Scaling-Up Reinforcement Learning using Parallelization and Symbolic Planning

Matthew Jon Grounds

Submitted for the degree of Doctor of Philosophy

The University of York

Department of Computer Science

April 2007

Abstract

Reinforcement learning (RL) methods are a family of techniques which allow an agent to improve its performance of a given task by *learning from direct interaction* with the environment it is situated in. Key to this approach is the notion of a *reward signal*, a numerical value observed by the agent which gives immediate feedback on the quality of its action choices. Using this signal, the agent can learn a policy which maximizes the total reward accumulated over time.

While many RL algorithms have theoretical convergence guarantees, achieving fast convergence to the optimum can be problematic in practice. There are particular problems with large-scale domains. As the learning environment becomes more complex and difficult to describe, the time required for an RL agent to learn an optimal policy grows very rapidly. This effect is known as the *curse of dimensionality*.

In this thesis, two different approaches to *scaling-up* RL are investigated. The first approach exploits parallel hardware to generate high-quality policies for simulated RL environments. An agent learns from simulated experience on each node of a parallel *cluster*. The agents periodically exchange weights from their approximate value functions. This allows a group of agents to converge more quickly than a single learner without compromising the final quality of the learned policy.

The second approach is a hybrid method combining *symbolic planning* and RL. A high-level knowledge base is used to generate a symbolic plan which provides *structure* for the learned policy. Abstract symbolic operators are implemented in terms of low-level actions using RL. This approach is shown to scale to much larger RL problems than is feasible with either standard or *hierarchical* RL algorithms.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would first like to thank my supervisor Daniel Kudenko for his guidance and collaboration during the years of research leading towards this PhD thesis. Daniel has helped me weather the highs and lows of the PhD process, encouraged me to explore a wide range of ideas during my time at the University of York, and has helped me develop the skills I will need for a future career in research. He has also provided a raft of suggestions that have improved both the content and polish of this document.

I gratefully acknowledge the financial support of QinetiQ Ltd which has made this research possible[1]. I would also like to thank Malcolm Strens of QinetiQ for his personal support during the development of this work. Through our conversations I have acquired a deeper understanding of my subject, while at the same time he has kept me aware of the problems that arise in industrial applications.

I would also like to thank my friends and colleagues at the University of York who have made my journey all the more interesting and enjoyable. Spiros Kapetanakis has been a good friend from the day I arrived, and has always been willing to hear about my work and to assist me in developing new ideas. Enda Ridge has helped me improve the quality of my statistics, and on many occasions has listened to my woes over a hot cup of coffee. I should also thank Joss Wright for teaching me how to play Go, which has been instrumental in helping me relax after a hard day at the office. Thanks also to Heather Barber, Thimal Jayasooriya and Joseph Marshall for reading chapters of my thesis and suggesting improvements.

Finally, I want to thank my wife Amelia for her love, understanding, humour and belief that I could finish! I couldn't have done it without you.

[1]This report was carried out under the terms of Contract CU004/26749 for the Director of Future Systems & Technologies Division, QinetiQ Ltd, Farnborough.

Declaration

I hereby declare that the research presented in this thesis is original work undertaken by myself, Matthew Jon Grounds, unless otherwise indicated in the text. This research was undertaken between October 2003 and November 2006. Where appropriate, previous results and techniques upon which this thesis builds are acknowledged with clear references to external sources. Some parts of this thesis have appeared previously in the papers listed below. In each of these papers, the major contributions were made by myself with input from my supervisor, Daniel Kudenko.

Matthew Grounds, Daniel Kudenko and David White. Parallel Reinforcement Learning by Merging Function Approximations. In *Sixth European Workshop on Adaptive and Learning Agents and Multi-Agent Systems*, Brussels, April 2006.

Matthew Grounds and Daniel Kudenko. Combining Reinforcement Learning with Symbolic Planning. In *Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*, Paris, March 2005.

Chapter 1

Introduction

This thesis focuses on techniques for *reinforcement learning (RL)*. RL methods allow agents to learn to choose actions effectively by observing the value of a *reward signal*. The reward signal gives an agent immediate feedback about the quality of each of its action choices. RL methods have been applied successfully in a wide variety of domains, but remain infeasible in many others. This is because when an agent's environment has a large number of possible configurations, standard RL algorithms are not able to find good action selection policies within a reasonable time.

In this work, two different sets of techniques are investigated which can be used to extend the applicability of RL to more difficult learning environments. The first set of techniques allows *parallel hardware* to be exploited so that a high quality policy for action selection can be learned much faster than would be possible on a sequential computer. The second set of techniques uses *symbolic planning* in combination with RL to provide a high level structure which constrains and accelerates the learning process. A wide-ranging empirical study is used to demonstrate the advantages of these techniques over standard RL methods.

This chapter begins with a high-level overview of reinforcement learning, followed by a description of the problems faced by RL in large-scale learning environments. The two principal topics of the thesis are then discussed: the use of parallelism in RL, and the combination of symbolic planning and RL. The introduction ends with a summary of the contributions of the thesis and an overview of the content of the remaining chapters.

1.1 Reinforcement Learning

One of the primary goals of Artificial Intelligence (AI) is the creation of *intelligent agents* (Russell and Norvig, 2003), which have the ability to *sense* external

stimuli, *perceive* the state of an environment, *reason* using knowledge about the environment, *learn* from past experience in the environment, and *act* to affect the environment according to internal goals. While each of these five abilities are considered to some degree in this work, the focus of the thesis is on *learning* in agents. There are many advantages of giving an agent the ability to learn from experience. If the dynamics of an environment are known to change over time, an agent can gradually modify its action choices to maintain a good level of performance. Adaptivity also adds some degree of robustness to an agent. If a situation arises that the agent's designer did not foresee, it may still be possible for the agent to learn an acceptable behaviour for this unforeseen situation. In *multi-agent systems* (Alonso et al., 2003), adaptivity can allow an agent to learn how to behave in a particular group or configuration. This is particularly important in *open* multi-agent systems, where new, unfamiliar agents may arrive at any time, and the agent must learn quickly to perform well in their presence.

A *reinforcement learning (RL)* problem (Sutton and Barto, 1998) is most easily characterized using the idea of an agent situated in an environment. The *state* of the environment can be observed by the agent[1]. At each time step in a *discrete* series, the agent must select an *action* to perform. After the action is performed, the environment enters a new state. In addition, the agent receives a *reward* or *reinforcement* for performing the action in that state. The reward indicates to the agent whether the choice of the action was good or bad, and as a *scalar* quantity it also indicates exactly *how* good or bad it was.

An important aspect of RL is reasoning about *future rewards*. It may be possible to achieve an extremely large reward in the future if the correct sequence of low-reward actions is followed. If this is the case, it is worth learning to follow this particular sequence, since the total reward accumulated over time will be greater. This leads to what is known as the *temporal credit assignment problem*. If after a long series of actions a large reward is received, it can be difficult to identify which of the actions in the sequence were instrumental in achieving the reward and which of the actions were not required at all.

Q-learning (Watkins, 1989), for example, is a popular RL algorithm which effectively solves the temporal credit assignment problem. If the sets of states and actions are finite, then under certain conditions Q-learning is guaranteed in theory to converge to an *optimal*[2] policy for choosing actions. Q-learning also performs well in practice if the sets of states and actions are not too large. Good performance with Q-learning can even be achieved in some cases where the environment is non-

[1]Sometimes only part of the state may be directly observed, with other parts remaining hidden.
[2]Various criteria for optimality are given in Chapter 2.

stationary (where the dynamics are changing over time) or non-Markovian (when there is some hidden state that cannot be directly observed).

1.2 The Curse of Dimensionality

Scaling-up reinforcement learning to more challenging learning environments is difficult because of the effect known as the *curse of dimensionality*, or alternatively as the *state space explosion*. Standard RL algorithms make the assumption that the learning environment can only be in one of a finite number of possible configurations. However, in most cases the state of the environment is naturally broken down into a set of *state variables*, each of which can be assigned a finite number of values. Complex learning environments tend to have many state variables, with each state variable having a wide range of possible values. As more complex environments are considered, the size of the overall state space increases rapidly. This in turn produces a rapid growth in the time required to learn a near-optimal policy with RL. Beyond a certain level of environmental complexity, RL algorithms which enumerate every possible state of the environment are simply not feasible.

Further complications arise in domains with continuous state variables (e.g. a robot situated in a three dimensional Euclidean space). While it is obviously possible to discretize these variables, even a coarse discretization of a few continuous state variables will create a large number of states. It would be preferable to use an algorithm which could deal directly with these continuous quantities.

In recent years there have been many techniques developed which allow modified RL algorithms to learn in some of these more complex domains. As part of this thesis, I present a comprehensive survey of these techniques in Chapter 3. Probably the most important addition to the standard RL algorithms is the use of *generalization*. Generalization is possible when states of the environment which have *similar* (but not identical) state features have a similar long term value. If this is the case, the exact table-based data structure used by algorithms such as Q-learning can be replaced with a *function approximation*. Since experience in one state will now affect the estimated values of lots of similar states, good policies may be achieved much more quickly. A good set of features for the approximator must usually be selected by hand, as is the case for most machine learning methods.

In this thesis, I examine two approaches to scaling-up reinforcement learning which have received relatively little attention: the use of parallel computing to reduce the time required to obtain a high-quality policy, and the use of symbolic planning techniques in combination with reinforcement learning.

1.3 Parallelization and RL

While there has been considerable progress in pushing forward the frontier of what is achievable with reinforcement learning, there remain many interesting problems which are of *borderline feasibility*. Standard RL algorithms may take several hours or even days of computation time to converge to a high quality policy for these problems. Given the computational effort required, it is reasonable to ask the question "can parallel computing hardware be used to obtain a high quality policy more quickly than is currently possible on a uniprocessor computer?"

Despite the significant computational requirements of RL algorithms, there has been very little research undertaken on parallel approaches to RL problems. This is somewhat surprising, considering that parallel approaches to the closely related problem of *planning* in *Markov decision processes (MDPs)* have been fairly well explored (Archibald, 1992; Wingate and Seppi, 2004). The lack of attention may be related to the fact that the essentially *sequential* interaction between a reinforcement learner and its environment does not yield directly to a natural parallelization.

The reason that parallelism has relevance for RL arises from the predominant use of *simulated learning environments* for the purpose of training RL agents. If an environment is simulated, it is relatively easy to situate a number of identical *instances* of the simulation on different nodes of a parallel computer. If a set of agents can interact with these instances *in parallel*, then by sharing intermediate results it is likely that the set of agents can converge towards a high-quality policy more quickly than a single agent learning in isolation.

The assumption of a simulated environment does exclude interesting cases such as an *embodied* agent situated in a real-life environment, or a software agent learning whilst deployed in an unpredictable open multi-agent system. In practice, however, generating experience in these non-simulated environments is usually expensive, and finding a high-quality policy for a large-scale RL problem will usually involve some degree of environmental simulation. A parallelization technique which requires a simulated environment will therefore be relevant for a wide range of existing problem domains.

Hence the first hypothesis to be investigated as part of this work is as follows:

> **Hypothesis 1**
>
> It is possible to exploit parallel hardware in reinforcement learning to achieve a speedup without sacrificing policy quality.

In this thesis, a series of increasingly efficient methods for parallel reinforcement learning are presented. Each of these methods uses a set of agents, where each

agent resides on one node of a *distributed-memory* parallel computer. Each agent interacts with a *local instance* of the simulated environment. The agents individually use standard RL techniques, including the use of *generalization*. Each agent learns an approximate value function which is represented using *linear function approximation*. Ensuring that the parallel method is effective *in combination* with generalization ensures that the method will have practical use for the most difficult RL problems, which are likely to be infeasible without some degree of generalization.

In the parallel method described in this work, the agents exchange information about their policies (in the form of *approximator weight values*) over the interconnection network of the parallel computing system. By using other agents' weight values to modify the local approximator weights, agents as a group are able to converge more quickly towards a high-quality policy. The agents are able to achieve this without each agent being restricted to a small area of the problem state space. All of the agents are able to explore the environment in an unrestricted manner.

The first parallel method presented in this thesis involves every agent broadcasting its entire set of weights periodically. The impact of the communication costs of this method means that a parallel speedup can only be achieved for a limited number of problem domains. Subsequent methods improve on the performance of the first method, by prioritizing the communication of weights which have recently undergone rapid change, and also with the effective use of *asynchronous message passing*.

This thesis includes a wide-ranging empirical evaluation of these methods using a cluster of Linux workstations. Five different example RL problems (some of which are well-known benchmark problems for RL algorithms) are used to illustrate the size of the speed-ups that can be achieved.

1.4 Symbolic Planning and RL

Symbolic planning (also known as *classical* planning), like reinforcement learning, is a mechanism for reasoning about *useful sequences of actions*. Unlike RL, symbolic planning is typically applied to *deterministic* domains where the only objective is to reach one of a set of goal states using the shortest number of actions. In addition, the outcomes of actions are known *a priori*, and do not have to be learned through trial and error.

Symbolic planning methods use a *relational* representation of state, which is generally based on a variant of first-order logic. Popular representations for symbolic planning include the STRIPS representation (Fikes and Nilsson, 1971) and

the situation calculus (McCarthy, 1963). Relational representations of value functions and policies have become increasingly popular for RL (van Otterlo, 2005) in domains involving objects and inter-object relationships.

In contrast to most other work in this area, this thesis is not concerned with relational versions of existing RL algorithms. Instead, one of the goals of the thesis is to investigate synergistic combinations of symbolic planning and RL in a *hybrid* approach. In the approach considered here, a symbolic plan forms the high level structure of a solution to the learning problem, with RL being used to fill in the low-level details of the solution. This approach is called PLANQ-learning within this thesis, and the second hypothesis to be investigated as part of this work is as follows:

> **Hypothesis 2**
>
> A hybrid planning-learning system based on a high-level STRIPS-based planner and low-level reinforcement learning will exhibit better scaling properties than both standard and hierarchical RL algorithms for goal-oriented learning problems.

To evaluate how well PLANQ-learning scales up to larger problem instances, a family of grid-world based learning problems is defined in this thesis. Progressively more difficult problems (with larger state spaces) can be created by increasing the value of a parameter which controls the size of the problem. This allows the performance of a learning algorithm to be assessed as a quantitative measure of problem scale is increased. In this work, it is shown that PLANQ-learning scales well to some extremely large problems in this family, where alternative approaches such as standard Q-learning and *hierarchical* reinforcement learning perform poorly.

1.5 Contributions

The work in this thesis focuses on the use of *parallelization* and *symbolic planning* as a source of techniques to *scale-up* reinforcement learning to large scale problems. The principal contributions of the thesis are as follows:

1. A novel approach to parallel RL, where a group of agents learning in parallel can quickly find a high-quality solution to a *single-agent* RL problem by *periodically exchanging approximator weights* over an interconnection network. In contrast to previous approaches, each agent may explore the entire state-space of the problem, not being restricted to a sub-region of this space.

2. Three novel methods for parallel RL which are based on the above approach. The *visit-count merge method* involves calculating a weighted average of the

agents' value function approximations to produce a *merged* value function. The *selective merge method* is based on broadcasting each agent's largest recent changes to its value function approximation. The *asynchronous selective merge method* achieves an extra boost in performance by removing the need for synchronization between the agents.

3. A wide-ranging empirical evaluation of the three parallel RL methods. The evaluation is based on the parallel speedups which can be achieved using different numbers of nodes in a cluster of Linux workstations. Five different learning problems are used in the evaluation. These learning problems vary in difficulty but also exhibit a range of characteristics, such as the level of stochasticity in the action effects, whether they are continuing or episodic problems, and whether they are *goal-oriented* problems.

4. The PLANQ-learning method, a novel combination of high-level STRIPS planning and low-level reinforcement learning. Empirical evidence is presented in the thesis to show that PLANQ-learning scales significantly better than both standard RL methods and hierarchical RL methods in learning problems where the high level solution structure can be modelled with a STRIPS knowledge base.

1.6 Thesis Structure

The remaining content of this thesis is structured as follows:

Chapter 2 presents an overview of basic reinforcement learning techniques. The key concepts of agent, environment, state, action and reward are described. The formalization of RL problems as *Markov decision processes (MDPs)* is discussed, and details are provided of some of the standard RL algorithms, namely Q-learning, SARSA, TD-learning and policy search. Terminology useful for describing the characteristics of particular RL algorithms is introduced. Readers already familiar with reinforcement learning techniques may prefer to skip this section.

Chapter 3 contains an extensive survey of existing methods for *scaling-up* reinforcement learning to larger, more difficult problems. The existing body of research is divided into the following five broad categories: efficient exploration, value function approximation, hierarchical reinforcement learning, symbolic representations, and parallel reinforcement learning. For each of these categories, common threads of existing research are grouped together, and within each category I will assess the potential of these techniques for reducing the impact of the curse of

dimensionality.

Chapter 4 begins with a motivation for the use of parallel hardware to perform RL, and goes on to state the assumptions underlying the work on parallel RL presented in the thesis. The basic operation of a parallel approach where a group of parallel agents *merge* their value function approximations is then presented. The criteria used to evaluate the parallel methods in the thesis are given, including a description of each of the single-agent learning domains which will be used to produce benchmark results. A number of different mechanisms to merge the approximator weights are proposed, and are first evaluated using a *simulation of parallel agents.* The most successful of these mechanisms, the *visit-count merge method*, is also evaluated in a more realistic setting, using a cluster of Linux workstations. As part of the evaluation, results are presented that show how communication between the agents is a vital component of the proposed parallel approach. The effect on performance of the choice of *how often* the agents exchange information is also examined.

Chapter 5 introduces a new approach to the use of communication in the parallel method. Rather than exchanging the absolute values of approximator weights over the network, agents instead broadcast the *recent changes* observed in their local weight value. In addition, agents no longer communicate information about all their weights, only the ones which have undergone the *greatest* recent change. This approach is known here as the *selective merge method.* Since each agent now only communicates partial information about how its weights have changed, a new mechanism is required for combining information received from other members of the group. This mechanism is known as a *combination function.* Several candidates for the combination function are proposed, and each is evaluated using the cluster of workstations. While different combination functions produce the best performance in different learning problems, the overall performance using *any* of the combination functions is much better than that achieved with the visit-count merge method in Chapter 4.

Chapter 6 presents a method which builds on the selective merge method defined in Chapter 5, and increases the parallel speedup that can be achieved by eliminating the synchronization penalty involved in the selective merge. The methods proposed in Chapters 4 and 5 have a distinct communication phase, where each agent broadcasts information to the other agents and waits to receive all the information before updating its local value function. The *asynchronous selective merge method* on the other hand has no distinct phase of communication. Instead, each agent can decide independently when to inform other agents of changes to its value function, and incoming messages can be used to update the local value

function as soon as they arrive. Since updates to the value function approximator must now be derived from *individual* incoming messages, a different mechanism is required for asynchronous updates. Several update functions are proposed for this mechanism, and their performance is evaluated using the cluster of workstations. One of these update functions is shown to produce the best aggregate performance over all the example domains, producing large improvements in performance over the previous synchronous methods.

With **Chapter 7** we leave the topic of parallelism, and begin an investigation of how *symbolic planning* can be combined with reinforcement learning to producing a hybrid method which exhibits good scaling properties. The PLANQ-learning method is defined, which combines high-level STRIPS planning with low-level Q-learning. A family of grid-world evaluation domains is presented, which can be scaled up quantitatively to more difficult problems by modifying one of the domain parameters. An initial comparison of this method with the standard Q-learning algorithm shows that PLANQ performs significantly better in smaller domains, but that this advantage decreases as larger domains are considered. An analysis shows that this effect is due to the lack of a *state-abstraction* mechanism. This mechanism is added to PLANQ, which is then compared with the hierarchical HSMQ-learning algorithm, which can exploit the same state abstraction. The results of the evaluation show that PLANQ always requires fewer environmental time steps than HSMQ to converge to a high-quality policy, and in addition that less total computation time is required by PLANQ once the learning domain exceeds a certain size. PLANQ is shown to remain feasible for much larger learning domains than HSMQ.

In **Chapter 8** the overall conclusions of this thesis are drawn. Both the successes and shortcomings of the techniques presented in this work are examined. The potential for future research to extend this work is also assessed, with some of the important remaining questions being sketched in some detail.

Chapter 2

Background: Reinforcement Learning

This chapter provides a basic introduction to *reinforcement learning (RL)*. As well as introducing standard RL concepts, the chapter also presents the formal basis of RL using the *Markov decision process (MDP)*. Algorithms for *planning* and *learning* in MDPs are presented, as well as the *terminology* required to describe different aspects of these algorithms. Finally, the concept of *partial observability* in RL is described.

A reader who is already familiar with these concepts may wish to skim through this material and proceed on to Chapter 3, which presents an extensive review of existing methods for *scaling-up* RL to large-scale problems. Alternatively, for a more comprehensive introduction to basic RL techniques, the reader should refer to either Sutton and Barto (1998), Kaelbling et al. (1996) or Bertsekas and Tsitsiklis (1996).

2.1 Basic Concepts

The concept of a reinforcement learning problem is easiest to describe by considering an *agent* situated in some *environment*, as shown in Figure 2.1. The agent can sense information about the *state* of the environment. The agent can also affect the environment by taking one of a set of *actions* available to it. After each action is taken, the agent receives a feedback signal from the environment called the *reward*, which determines how well the agent is performing the target task in the environment. The *goal* in a reinforcement learning problem is to learn which action to take in each state to *maximize* some measure of *optimality* based on the rewards received over time.

Formulating a learning problem in this way has a number of advantages. In

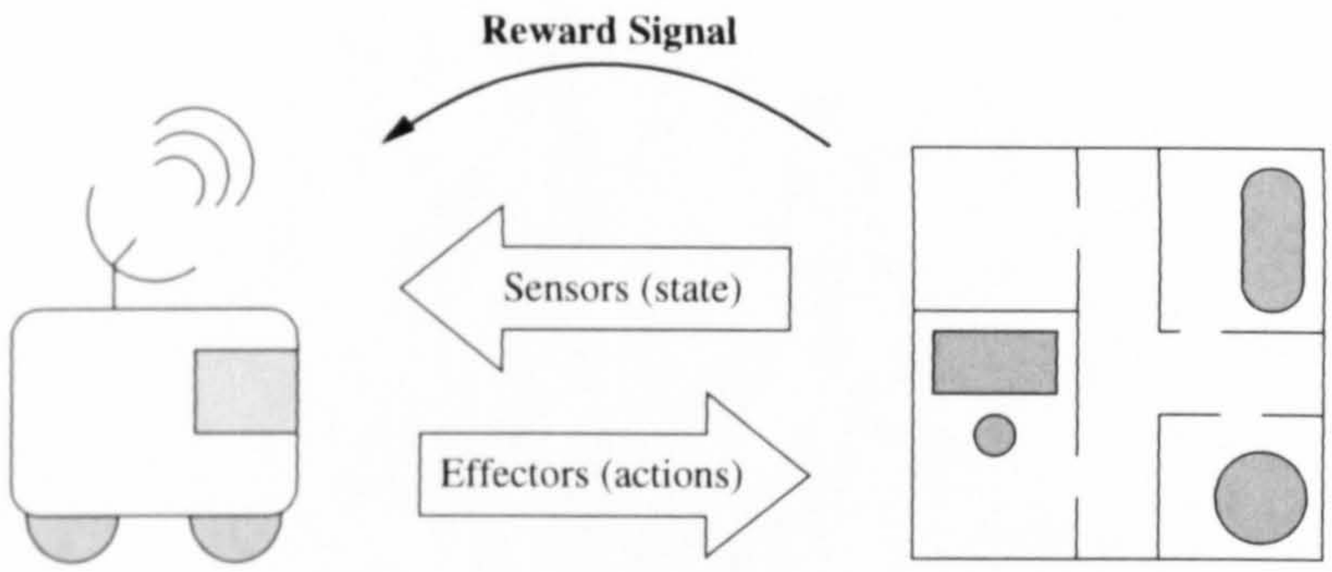


Figure 2.1: The basic components of a reinforcement learning problem.

contrast to a *supervised* learning method, it is not necessary to have a set of training examples annotated with the correct action for the agent. Initially the agent (and even the designer) can be completely ignorant of the best action for any state. It is also unnecessary to have an existing model of how actions affect the environment. As long as the reward function characterizes which situations are the most desirable for a given task, and the sets of possible states and actions are known, a reinforcement learning algorithm can find the optimal action choice for each state.

Reinforcement learning is a natural choice for agent-based problems in areas like autonomous robotics (Stone, 1998) and virtual environments (Guestrin et al., 2003). However, reinforcement learning can also be very useful in domains not typically characterised as agent problems, such as low-level motor control (Kirchner, 1998), dynamic channel allocation (Singh and Bertsekas, 1996), and search-control for scheduling problems (Zhang and Dietterich, 1995).

2.2 The Markov Decision Process

A given reinforcement learning problem can be formalized as a *Markov Decision Process* (Bellman, 1957) or MDP. An example of an MDP is shown in Figure 2.2. An MDP is described by a tuple $< S, A, T, R >$ where:

- S is the set of possible *states*.
- A is the set of available *actions*.
- $T(s, a, s') \rightarrow [0, 1]$ is the *transition function* defining the probability distribution $p(s'|s, a)$, the probability that taking action a in state s will result in a transition to state s'.

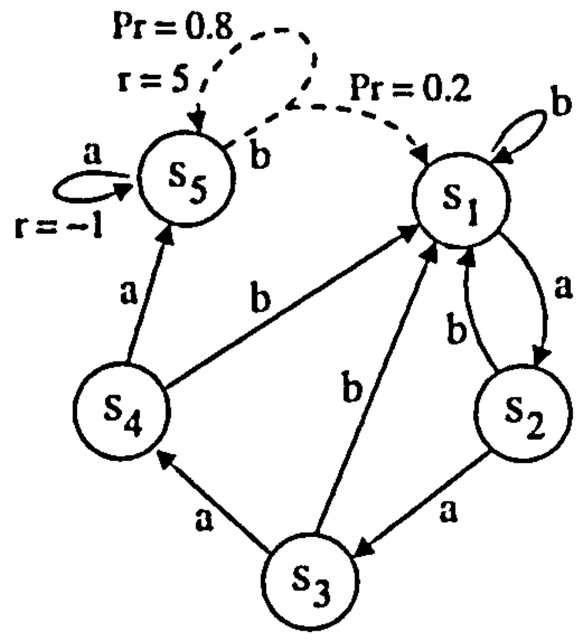


Figure 2.2: A Markov Decision Process with five states, two actions (a and b), and a single stochastic transition. Reward is assumed to be zero if not marked for a transition.

- $R(s, a, s') \rightarrow \mathbb{R}$ is the *reward function* defining the *expected* reward[1] received when such a transition is made.

A particular strategy for choosing actions in an MDP is known as a *policy*, and is specified formally as a function $\pi(s, a) \rightarrow [0, 1]$, which defines the probability $p(a|s)$ of selecting each action in a given state. Writing π_t for the policy at time t, if the policy changes over time ($\pi_{t_1} \neq \pi_{t_2}$) then the series $\{\pi_0, \pi_1, \pi_2, ...\}$ is said to be a *non-stationary* policy. A *stationary policy* π has the property that $\forall t.\pi_t = \pi$. A *deterministic* policy, usually written as $\pi(s)$, maps each state with probability 1 to a single action.

To compare different policies, it is necessary to define an *optimality criterion*, a measure of the quality of a particular policy. A number of different optimality criteria have been defined, of which the most common are given below. Here r_t is the reward received after taking an action on time step t. The notation $E_\pi\{\}$ indicates the *expectation* of the expression in the braces given that policy π will be used to select actions. For the third criterion we also require a *discount factor* $\gamma \in [0, 1)$.

- *Total Return over a Finite Horizon*

$$\text{optimality}(\pi) = E_\pi \left\{ \sum_{t=0}^{N-1} r_t \right\}$$

- *Average Return over an Infinite Horizon*

$$\text{optimality}(\pi) = E_\pi \left\{ \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{t=0}^{N-1} r_t \right\}$$

[1]Each transition can potentially have its own random distribution of rewards, so to fully specify the MDP, we should also specify these distributions. For defining optimality criteria and most algorithms, modelling the expected value for each transition is sufficient.

- *Total Discounted Return over an Infinite Horizon*

$$\text{optimality}(\pi) = E_\pi \left\{ \sum_{t=0}^{\infty} \gamma^t r_t \right\}$$

Of these, the *total discounted return over an infinite horizon* is the most common and well-understood optimality criterion, and this is the one that will be used from this point forward. For further information on average reward and finite horizon MDPs the reader is referred to Bertsekas (2001).

Here we define a *value function* $V^\pi(s)$ as the *expected total discounted return* when starting in state s and using policy π to choose actions (assuming some fixed value of γ). Intuitively, $V^\pi(s)$ represents the utility of a particular state of the MDP under policy π. The discount factor γ is used to determine the relative worth of future rewards in comparison to rewards available immediately in the current state. The value of γ is chosen to be less than 1 to give $V^\pi(s)$ a finite value for each state. The values of $V^\pi(s)$ at different states can be related using the transition and reward functions as follows:

$$V^\pi(s) = \sum_a \sum_{s'} \pi(s,a).T(s,a,s'). \left[R(s,a,s') + \gamma V^\pi(s')\right]$$

This formula, which forms the foundations of most of the algorithms for planning and learning in MDPs, relates the value of a state to the *expected immediate reward* in that state and the *value of the successor state(s)*.

An *optimal policy* π^* is a policy which, according to our optimality criterion, performs better in the MDP than any other policy π. More formally, the policy π^* satisfies:

$$\forall \pi \forall s. \left(V^{\pi^*}(s) \geq V^\pi(s) \right)$$

While the goal of MDP planning and learning is usually to find π^*, MDP solution methods are often based on a calculation of the value function for the optimal policy V^{π^*}, also denoted by V^*. Once V^* has been calculated, the parameters of the MDP can be used to calculate an optimal deterministic policy π^*:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right] \tag{2.1}$$

2.3 Planning in MDPs

If all the parameters of the MDP (S,A,T and R) are known, *dynamic programming* methods (Bellman, 1957) can be used to determine the optimal policy and value

function. An important algorithm for dynamic programming is *value iteration*, a method for calculating V^*. The algorithm represents (with a table of real numbers) a current estimate of $V^*(s)$ for each state s. We will write this estimate as $V(s)$. Before the algorithm begins the values in the table may be initialized arbitrarily. The algorithm is based on re-estimating each $V(s)$ based on the current value estimates for the successor states of s. Each re-estimating update to the table of values is known as a *Bellman backup*, and is defined as:

$$V(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right]$$

It can be shown that by repeatedly iterating over the set of states and performing the Bellman backup for each state in the table of values, each $V(s)$ value will eventually converge to $V^*(s)$.

Policy iteration is another important dynamic programming algorithm, which consists of alternate periods of *estimation* and *maximization*. Given a deterministic starting policy π_0, another form of value iteration is used to *estimate* the value function V^{π_0} for that policy. We use the following update rule:

$$V(s) \leftarrow \sum_{s'} T(s,\pi_0(s),s') \left[R(s,\pi_0(s),s') + \gamma V(s') \right]$$

By iterating over the set of states, each $V(s)$ value will converge to $V^{\pi_0}(s)$ using this update rule. Once the value function is sufficiently well estimated, a new improved policy π_1 is constructed by *maximizing* based on V^{π_0}. We make greedy choices at each state based on the values of successor states, using the following formula:

$$\pi_1(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_0}(s') \right]$$

Now we can go on to calculate V^{π_1} in the next estimation phase, and make greedy choices in this new value function to find π_2. These alternating phases of estimation and maximization are repeated until two subsequent policies π_n and π_{n+1} are unchanged, at which point the algorithm has converged.

2.4 Learning in MDPs

Reinforcement learning algorithms operate under different assumptions than algorithms for MDP planning. The only parameters of the MDP known at the start of learning are the state and action sets S and A. The transition and reward functions T and R must be estimated during learning by *interaction with the environment.*

Despite these differences, MDP planning and RL are closely related, and most RL algorithms are based on either a *value iteration* or a *policy iteration* approach.

To determine the optimal policy π^* for a reinforcement learning problem, it is *insufficient* to learn V^*, since Equation 2.1 cannot be applied if T and R are unknown. One way to calculate both V^* and π^* is to learn the related function $Q^*(s,a)$, defined as:

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

While $V^*(s)$ is the optimal value function defined over states, $Q^*(s,a)$ is the optimal value function defined over *state-action pairs.* From $Q^*(s,a)$ we can readily calculate both V^* and π^* as follows:

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

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

Q-learning

The *Q-learning* algorithm (Watkins, 1989) is a method for learning the Q^* function, and is probably the most well known reinforcement learning algorithm. It is popular both for its simplicity of implementation and its strong theoretical convergence results, and it exhibits good learning performance in practice.

Q-learning is similar to the *value iteration* dynamic programming method, in that a table of real numbers is used to store the current estimate $Q(s,a)$ of Q^* for each s and a, and that re-estimation is made on the basis of the estimates of successor states. Each value in the table is initialized arbitrarily, usually by setting it to zero or assigning it a small random value. An *experience tuple* $< s, a, r, s' >$ is a small excerpt from the trace of an agent's interaction with the environment. The full trace has the form $\{s_0, a_0, r_0, s_1, a_1, r_1, s_2, ...\}$. As experience tuples are generated through interaction with the environment, the value function is updated using the following rule:

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

The learning rate $\alpha \in [0,1]$ determines the extent to which the existing $Q(s,a)$ estimate contributes to the new estimate. The purpose of the learning rate is to allow each $Q(s,a)$ estimate to slowly converge to the *expected* future rewards in the face of stochastic MDP transitions (or a stochastic reward function). In theory,

this algorithm is guaranteed to converge to the optimal value function Q^* as long as each state-action pair is visited an infinite number of times in the limit and the value of α is decayed in the correct way (Watkins and Dayan, 1992). To achieve good results in practice, a careful choice of *exploration strategy* is required.

An exploration strategy is a mechanism for making a trade-off between *exploration* and *exploitation.* Exploration introduces randomness into the action choice, in order to explore the state space for rewards which have not yet been encountered. In contrast, exploitation is the choosing of actions which lead to the best rewards discovered so far. Usually a strategy will start off taking mainly explorative actions, introducing a greater proportion of exploitative actions as learning proceeds. A good choice of exploration strategy is a prerequisite for timely convergence in practice. Exploration strategies are discussed in more detail in Section 3.3.

SARSA

The SARSA algorithm (Rummery and Niranjan, 1994) is closely related to Q-learning. It uses the same data structure (a table of state-action values) and has a very similar update rule. However, while Q-learning converges to the optimal value function $Q^*(s,a)$, the SARSA algorithm converges to the value function $Q^\pi(s,a)$. Assume for the moment that π is stationary, i.e. the action choice in each state is *fixed*, and is not affected by the current value estimates or any exploration policy. The value of $Q^\pi(s,a)$ in this context is the expected return if we start in state s, execute action a, then use policy π to choose *all subsequent actions.*

The algorithm gets its name[2] from the letters used in the experience tuples generated during learning. If the agent takes action a in state s, receives reward r, and then proceeds in the next time step to take action a' in state s', the experience tuple $< s, a, r, s', a' >$ is generated. The rule to update the value function based on this tuple is:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s',a'))$$

The update rule differs from Q-learning in the way that the successor state value is estimated. In Q-learning the *maximum* value out of *all the actions* in the successor state is used as the estimate. In SARSA the value of the action *actually chosen by the learning agent* at the next time step is used instead. A

[2]Rummery and Niranjan (1994) actually called this algorithm *Modified Q-learning (MQ-L)*, but the alternative *SARSA* designation popularized by Sutton (1996) is the one which seems to have stuck.

slight modification to the proof of Jaakkola et al. (1994) can be used to establish the theoretical convergence of SARSA to the value function Q^{π}.

In addition, the SARSA algorithm can also be used to learn the optimal value function Q^{*}. This is achieved by relaxing the restriction that π has to be stationary. If the learner can take exploratory actions, but gradually tends towards greedy choices in the estimated value function, then the estimates will converge towards Q^{*} instead of Q^{π}. Singh et al. (2000) provide a theoretical proof that SARSA will converge to Q^{*} when an appropriate exploration strategy is employed. This approach also works well in practice, and we shall see later that it is preferable to the Q-learning algorithm in some specific situations.

TD(λ)

The reinforcement learning algorithms discussed so far, Q-Learning and SARSA, both update the value of state-action pairs based upon the estimated value of the state *one time step later*. An alternative to this approach is to also use the estimated values of states encountered *two or more time steps later* to re-estimate the original state's value. This is the intuitive idea behind the TD(λ) algorithm (Sutton, 1988).

TD(λ) learns the state value function V^{π} for the control policy π used by the agent to select actions. Because TD(λ) does not learn individual action values, it is most useful when we are only concerned with evaluating the quality of an existing policy, or if there is some external model of transition behaviour.

A parameter λ (where $0 \leq \lambda \leq 1$) is used to determine the degree to which the value of a state encountered n time steps later contributes to the value of the state being updated. The value of the state n steps later contributes a factor of λ^{n-1} less than the immediate successor state.

TD(λ) is usually implemented using an *eligibility trace*. The eligibility trace for a state s is a value e_s which determines the extent to which s should be updated using the value of the current state s_t. At every time step each of the e_s values is updated as follows[3]:

$$e_s \leftarrow \begin{cases} \gamma\lambda e_s & \text{if } s \neq s_t \\ \gamma\lambda e_s + 1 & \text{if } s = s_t \end{cases}$$

Once the eligibility trace values have been updated, the current estimate of each state value can also be updated:

[3]An eligibility trace updated in this way is known as an *accumulating trace*. An alternative approach is the *replacing trace* which sets the eligibility of the current state to 1 instead of incrementing it by 1.

$$\delta_t \leftarrow r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

$$V(s) \leftarrow V(s) + \alpha e_s \delta_t$$

The best value for λ varies depending on the problem being solved. TD(λ) with a good choice of λ generally converges after fewer steps in the environment than TD(0) (which only uses the value of the immediate successor state in each update). This improvement in convergence speed has resulted in eligibility traces being more widely applied in reinforcement learning algorithms. For instance, SARSA and Q-learning have been extended with eligibility traces to produce the SARSA(λ) algorithm (Rummery, 1995) and two different Q(λ) variants developed by Watkins (1989) and Peng and Williams (1996) respectively.

A theoretical proof of convergence of the TD(λ) algorithm is given in Jaakkola et al. (1994).

Policy Search

It is worth noting that not all algorithms for solving RL problems necessarily involve the manipulation of value function data structures. A number of researchers, notably in the area of autonomous robotics, have found that in some situations it is better to avoid value functions all together, retain only some form of policy representation, and *search* in *a space of policies.* In these methods, the policy is usually specified as a parameterized function $\pi(s, \vec{\theta})$, making the goal of the search to find a good set of parameters $\vec{\theta}$. The major advantage of this approach is that an agent designer's prior knowledge about what kind of structure good policies should have can be embedded into the parameterized function π, leaving the fine-tuning of the parameter vector $\vec{\theta}$ to the agent itself. This is a much easier task than trying to learn a non-structured value function for a complex structured task.

Each policy that is considered as part of the search must be *evaluated* to determine its quality compared to other policies that have been considered. The efficient use of sampled experience to compare the quality of policies is one of the topics studied by Peshkin (2001). Typically the search strategy is to determine the gradient in the policy quality with respect to the parameters $\vec{\theta}$ and adjust the parameters in the direction of the gradient's steepest ascent (Williams, 1992; Baxter and Bartlett, 2000). Alternative search strategies include exhaustive enumeration over a finite horizon (Pynadath and Tambe, 2002) and global search methods such as genetic algorithms and simulated annealing (Rosenstein and Barto, 2001).

2.5 Properties of Reinforcement Learning Algorithms

There are many other RL algorithms which will not be covered in detail here. However, it will be useful for the discussion in subsequent sections to define a number of features which can be used to classify different RL algorithms.

Online vs. Offline

If an algorithm is intended to interact directly with the environment and learn new information after each action is taken, it is termed an *online* algorithm. If instead the algorithm is designed to learn from an *execution trace* which records the states, actions and rewards which occurred during an episode interacting with the environment, it is termed an *offline* algorithm.

On-policy vs. Off-policy

Algorithms which learn a state-action value function from the experiences generated by an agent following a *control policy* π can be classed as one of two types. An *on-policy* algorithm learns the value function Q^π, i.e. the value function for the policy being followed by the agent. SARSA is an example of an on-policy algorithm.

An *off-policy* algorithm learns the *optimal* value function Q^* no matter which control policy π is followed[4]. The *control policy* π may be completely unrelated to the optimal policy π^*. Q-learning is an example of an off-policy algorithm.

On-policy algorithms can also be used to learn Q^*, but only if the agent gradually adapts its control policy towards greedy choices in the estimated value function. Off-policy algorithms allow a more flexible choice of control policy, but can be problematic in combination with function approximation (see Section 3.4).

Model-based vs. Model-free

A *model-based* reinforcement learning algorithm is one which builds an explicit model of an MDP which describes the learning agent's environment. The parameters of this model are estimated based on the experiences acquired by interacting with the environment. This MDP model can then be used either to *simulate* experiences for the learning algorithm, or to perform *Bellman backup* operations in the current (estimated) value function. In both cases, convergence to the optimal value function can be obtained after fewer experiences in the environment, at the

[4]As long as there is sufficient exploration of the state space. Policy π must visit every state-action pair infinitely often as time goes to infinity.

expense of more computation time per step in the environment. Prioritized sweeping (Moore and Atkeson, 1993) is a good example of a model-based algorithm. An algorithm such as Q-learning which builds no MDP model, and learns based on value function updates from experience tuples only, is known as a *model-free* algorithm.

Complexity Measures

The efficiency of RL algorithms in terms of various resources can be compared using the following complexity measures:

Memory Complexity The amount of memory required for data structures to learn and store a near-optimal policy.

Sample Complexity The number of experience tuples obtained from interaction with the environment required to learn a near-optimal policy.

Computational Complexity The computation time expended to process a single experience tuple after interacting with the environment.

Selecting a reinforcement learning algorithm for a particular domain often involves a trade-off between sample complexity and computational complexity. In domains where experiences in the environment are time-consuming or expensive, such as in autonomous robotics, minimizing sample complexity will be the primary concern. In other domains where simulated environments can be used to generate fast, cheap experience, a simpler method with a worse sample complexity may be preferred if this reduces the required learning time.

2.6 Partial Observability

An implicit assumption underlying the discussion so far is that the learning agent can detect with 100% accuracy the complete current state of the environment, and use this state to make the optimal action choice. In most real-world domains this assumption does not hold, and the true state of the environment is always *uncertain*. Autonomous mobile robotics is a good example of such a domain. Robotic sensors tend to be noisy, reporting imperfect information about the world. Robots are also *situated* at some location of the world, which means that the robot may only be able to observe events which take place at the same location. Events which occur at other locations may remain unknown for some time.

To formalise the notion of a problem which is only partially observable, we can extend the definition of an MDP (see Section 2.2) to define a *partially-observable*

Markov Decision Process, or POMDP. An example of a POMDP is shown in Figure 2.3. A POMDP is described by a tuple $< S, A, T, R, \Omega, O >$, where S, A, T and R have the same definitions as in the MDP. The first additional parameter Ω is a finite set of *observations* which represent the possible experiences the agent can have at each time step. The second parameter is a function $O(s', a, o) \rightarrow [0, 1]$ which defines the probability of making observation $o \in \Omega$ after taking action a and ending up in state s'. Note that one and only one member of Ω is observed on each time step.

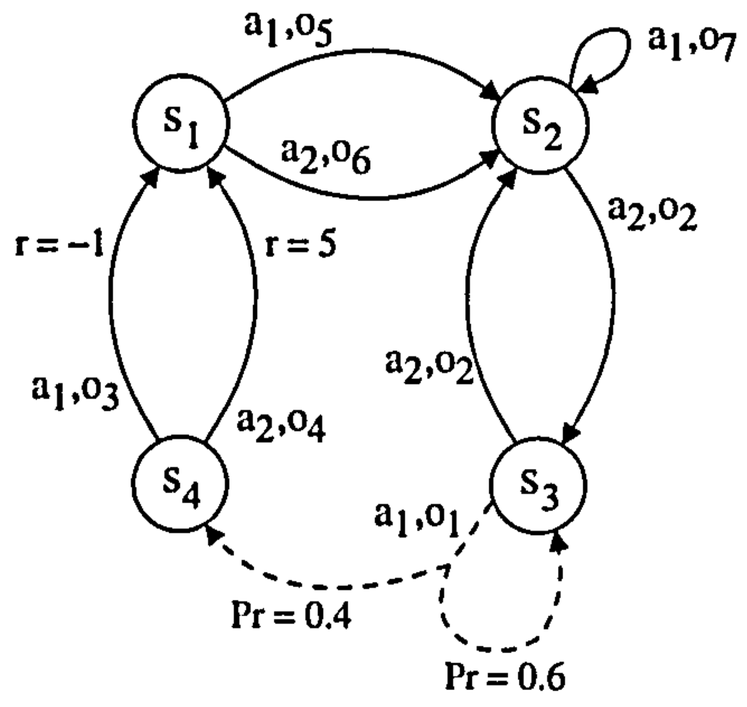


Figure 2.3: POMDP with four states, two actions, seven observations, and a single stochastic transition. Reward is assumed to be zero if not marked for a transition.

If all the parameters of the POMDP are known, and $|S|$,$|A|$ and $|\Omega|$ are all fairly small, then an *exact* solution of the POMDP can be found. A POMDP *induces* an MDP over belief states. Each element b_i of the current *belief state* $\vec{b}$ represents the probability that the current state is s_i. The state space of the induced MDP is the *continuous* space of beliefs $\mathbf{B}$. In addition, the optimal t-step value function $V_t(\vec{b})$ (the value of a belief state given that we can only take t further actions) has a *piecewise-linear* form. This means we can represent each V_t as a *finite set of vectors*. With this representation we can use value-iteration to calculate V_t for increasing values of t, gradually approaching the infinite horizon value function V^*. Some examples of exact POMDP algorithms are the *witness* algorithm (Kaelbling et al., 1998), *Incremental Pruning* (Cassandra et al., 1997) and the *linear support* algorithm (Cheng, 1988).

The complexity of exact POMDP algorithms is such that they are only appropriate for solving quite small problems. Pineau et al. (2003) present an approach similar to the exact methods, but limit the number of vectors that can be used to represent the intermediate value functions, resulting in a close approximation to the optimum for medium-sized POMDPs. For POMDPs with a large number of states, explicitly representing the belief state $\vec{b}$ and performing a full Bayesian

update each time step is infeasible, so some researchers have investigated approximate belief state representations, such as the approach by Roy and Gordon (2002) based on principal component analysis.

There are also a variety of approaches which avoid value iteration over belief states. Simmons and Koenig (1995) solve an POMDP problem as if the underlying states were fully observable, and use the resulting MDP solution as a heuristic to guide action choice in the POMDP . Policy search can also be used to exhaustively evaluate policies over a finite horizon (Pynadath and Tambe, 2002).

Empirical studies of POMDP planning have shown it to be much harder than MDP planning, a view which is supported by complexity results for the two planning problems (Madani, 2000). *Learning* in POMDP environments is possibly even more difficult, since a typical partially observable RL setting would involve experience tuples of the form $< a_t, o_t, r_t >$, with the agent having no prior knowledge of T, R, or O, and often not knowing the number of underlying states $|S|$. Estimating these unknown parameters is similar to the task of learning a *Hidden Markov Model* from observed data, but researchers who have modified the Baum-Welch algorithm (Rabiner, 1989) to learn POMDP models have found that this approach is computationally expensive (Chrisman, 1992; McCallum, 1996). An alternative approach is to learn a *predictive model* (Chrisman, 1992) to estimate the number and properties of the hidden states. This model may be based on a memory which stores the most recent actions and observations (Lin and Mitchell, 1992; McCallum, 1996), or use a more complex representation such as that of *TD Networks* (Tanner and Sutton, 2005) or predictive state representations (Wolfe et al., 2005). The predictive model can either be used to build an explicit POMDP model for planning, or combined with a model-free learning algorithm, using the predictive model only to identify the current hidden state.

In some circumstances it is possible to learn in a POMDP environment in a completely model-free fashion, with no attempt to identify the hidden state. For example, the *HQ-learning* algorithm (Wiering and Schmidhuber, 1997) is a hierarchical model-free approach to learning in goal-oriented POMDPs. HQ-learning is applicable when the task of reaching the goal is a linear sequence of sub-tasks, where each sub-task can be solved with a reactive policy mapping observations to actions[5]. HQ-learning relies on an *implicit* memory of past observations, since the active sub-task is an indicator of progress along the sequence. There are other model-free approaches (Littman, 1994; Cliff and Ross, 1994) which use explicit *memory bits* to record some part of the observation history. These approaches have a key advantage over a finite history window in that an agent can remember

[5]HQ-learning is described in more detail on page 71.

the important parts of the observation history compactly, even if the observations happened an arbitrarily long time in the past.

Partial-observability adds a further dimension of difficulty to many real-life problems. While much progress has been made in POMDP planning, state of the art algorithms remain computationally expensive, and representations for *learning* in POMDP environments are still evolving. Given the difficulty of planning and learning in POMDPs, modelling a large RL problem as a POMDP is impractical. It is likely in future that hierarchical environment models will limit the use of POMDP techniques to small sub-problems (see section 3.5 for a survey of existing hierarchical RL methods). Partial observability is a minor topic in this thesis, which will mostly be concerned with fully observable problems that can be modelled as MDPs.

Chapter 3

Background: Scaling-Up RL

In this chapter, the focus of attention is shifted to the problem of reinforcement learning in *large-scale* domains. The *state-space explosion* is presented as the primary challenge to overcome in order to scale-up RL. A wide range of techniques have been proposed for this purpose. A *categorization* of these techniques is defined in Section 3.2. Each category denotes a family of techniques which can be used to modify standard RL algorithms to allow them to be applied to a wider range of problems. This categorization is used to structure a *comprehensive survey* of existing techniques for scaling-up RL. At the end of the chapter, some broad conclusions are drawn from the complete survey.

3.1 The State Space Explosion

Reinforcement learning has been applied successfully in a variety of domains. It is an attractive approach when, for example, it is easier to define a good reward function than a full model of the environment, or when an environment is easily simulated but the principles behind an optimal policy for that environment are poorly understood. However, there remain many RL problems with no known optimal policy that are *infeasible* to solve with standard algorithms. Once the space of state-action pairs grows beyond a certain size, the time for standard algorithms to converge becomes too great, and in some situations there may not even be enough memory to store the entire table of state-action values. Standard algorithms are also based on a *finite* space of state-action pairs. There are many interesting learning problems where the state-action space is *infinite*, and usually in such cases the space is also *continuous*.

The key problem which arises when reinforcement learning is applied to large-scale problems is referred to as the *state space explosion*. It was also described by Bellman (1957) as the *curse of dimensionality*. The "flat" state space S used

by a traditional reinforcement learner can usually be expressed as the Cartesian product of n simpler state variables, $X_1 \times X_2 \times \cdots \times X_n$. Even if these were only binary state variables, $|S|$ would be equal to 2^n. As we scale-up to larger problems by increasing the number of state variables, the size of the state space S grows exponentially. Since the time required to learn an optimal policy grows at least as fast as the size of the state space for existing table-based RL algorithms (Strehl et al., 2006), the learning time will also grow exponentially as n is increased.

It is clear that in the fully general case of an arbitrary MDP with 2^n states (still assuming binary state variables), there is an inescapable limit on how large we can allow n to grow and still be able to find the optimal policy in a feasible time. Thankfully, real-life learning problems rarely exhibit the full generality of an unconstrained MDP. In a particular region of the state space, there may be only a few state variables which are relevant to the action choice. Alternatively, there may be a large group of states with similar state features which can be considered interchangeable in terms of state value and optimal action choice.

3.2 Categorization of Scaling-Up Techniques

In recent years, a wide range of techniques have been proposed to tackle the problem of scaling-up RL methods to solve larger and more difficult learning problems. To structure the survey of these existing techniques, they will be classified into the following five categories:

Exploration Strategy Since an RL agent begins with no knowledge of its environment, the agent must take *explorative actions* to discover the effects of each action and the states which contain large rewards. Once an environment has been well-explored, the agent normally tends towards *exploitative actions* which lead to the best rewards. To speed up the learning process, some researchers have focused on reducing the number of explorative actions required to learn the behaviour of the environment, meaning that the agent can move more quickly to exploit the rewards.

Value Function Approximation Many RL algorithms are based on a *value function* data structure. In its simplest form, a value function is a table of numbers which stores for each state an estimate of expected future reward. For large state spaces, representing an *exact* value function not only uses a lot of memory, but also causes many algorithms to converge more slowly. Replacing the exact table with an approximation means that the learning agent requires less memory and is better able to *generalize* from experience.

Hierarchical Reinforcement Learning In order to speed up learning, hierarchical RL methods employ a *divide and conquer* approach. An RL problem is decomposed into smaller sub-problems, and the results are combined to generate the overall policy. The problem decomposition can be carried out by creating an abstraction hierarchy of actions (*temporal abstraction*) and/or of states (*state abstraction*).

Symbolic Representations for Reinforcement Learning A symbolic representation of states and actions is often more compact than the extensional representation (which explicitly enumerates each state) used by standard RL algorithms. Symbolic representations also support mechanisms for reasoning using acquired knowledge, which can be used to accelerate the learning process.

Parallel Reinforcement Learning A number of agents learning in parallel can be used to find optimal policies for *single-agent* learning problems more quickly than a single agent learning in isolation. These methods can exploit the computing power of systems such as *multiprocessor* computers, *clusters* of computers and *grid computing* systems. The agents combine their results through a communication medium such as a *shared memory* or a network which supports *message passing.*

In the following five sections I will survey existing work in each of these categories to determine in each case the advantages and disadvantages of the general approach, the types of problem which will benefit most from the techniques in each category, and the limitations which are evident in each case.

3.3 Exploration Strategy

A key property of the standard reinforcement learning setting is that the agent initially has no information about the way the environment behaves. The dynamics of the environment (i.e. the transition and reward functions) can only be determined by performing actions in the environment and observing the results.

The goal in reinforcement learning is to find the optimal policy. But because the environment is initially unknown, there emerges a fundamental trade-off between choosing actions to gather information about the environment and choosing the actions which have (so far) proved to lead to the greatest rewards. This is usually described as a trade-off between *exploration* and *exploitation.*

Exploration Actions are chosen with the goal of discovering new information about the reward and transition behaviour of the environment.

Exploitation Actions are chosen which are likely to lead to the greatest rewards which have been discovered so far during learning.

Theoretical results in RL are usually based the notion of *asymptotic optimality*. An RL algorithm is asymptotically optimal if over an infinite learning time the algorithm is *guaranteed* to reach a point when all subsequent action choices are optimal. Applying RL algorithms in practice, however, requires that the learning time be both finite and feasibly short. In practice it cannot generally be guaranteed that an optimal policy will be learned. The best we can do is establish a *high probability* that a policy *close* to optimal will be learned in the available time.

Sometimes an RL task is formulated as entirely separate phases of exploration and exploitation. The initial exploration phase could simply be used to build as accurate a model of the environment as possible. This is known as *system identification*, an approach which essentially ignores the reward function during learning. Dynamic programming can then be used to determine a policy for the exploitation phase based on the learned model.

However, not all information about the environment is of equal value to the agent. During the exploitation phase the agent's goal is to accumulate rewards. Information about how large rewards can be obtained is more valuable to the agent than any other information about the environment. The specific problem of *exploration for future exploitation* was identified and investigated by Wyatt (1997). Note that during a separate exploration phase the accumulated reward is unimportant—the goal is simply to learn as much as possible about where the rewards are.

Separate exploration and exploitation phases are rare however, and usually some form of *exploration strategy* is used to choose actions. An exploration strategy encapsulates both exploration and exploitation in a single algorithm for action selection. It provides a trade-off between early exploration (to learn about the environment) and later exploration (to maximize reward). Exploration strategies which are partially exploitative are very effective in practice, regardless of whether the reward accumulated during learning is considered important[1]. This is because they tend to focus the exploration effort around paths in the MDP which lead to large rewards, as shown in Figure 3.1. This means that a suboptimal path leading to a reward can be quickly refined to the path which achieves that particular reward most quickly. However, random exploration away from such paths is still required to find larger rewards which have not yet been encountered.

[1]The difference in accumulated reward between the agent during learning and the optimal policy is known as the *regret*.

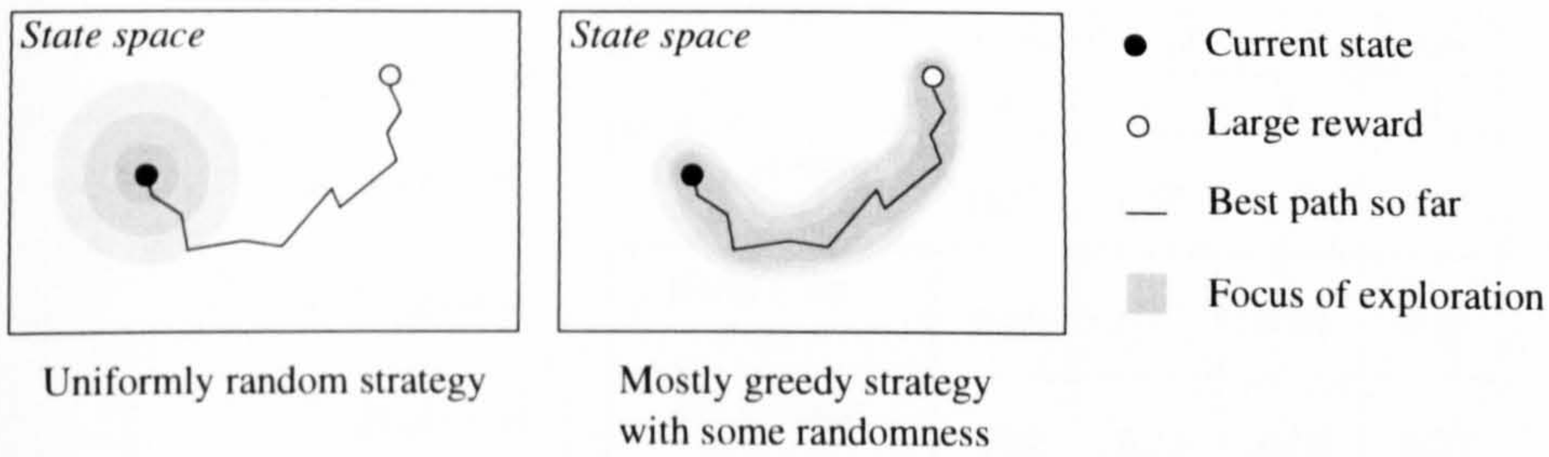


Figure 3.1: Exploration policies which are also partially exploitative focus the exploration effort around paths which are already known to lead to large rewards.

The choice of exploration strategy is critical to achieving timely convergence to the optimal policy. If there is insufficient exploration, the algorithm is likely to converge to a sub-optimal policy. If exploitation is delayed too long, the exploration effort will be spread too thinly over the state space and convergence will be slow.

3.3.1 Common Exploration Strategies

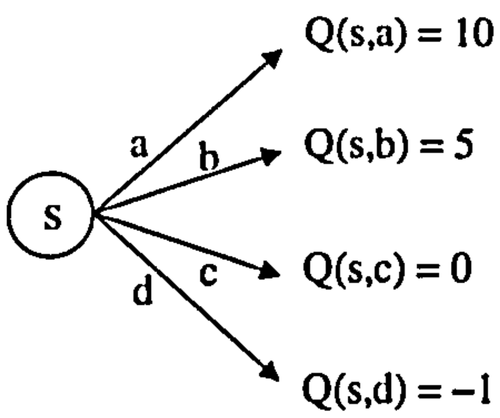
The two most commonly used exploration strategies both provide a mechanism to balance exploration between the two extremes of the *greedy* policy (which always picks the action with the largest $Q(s, a)$ value) and the *uniform random* policy (which assigns the same probability of selection to every action in a state).

ϵ-greedy Strategy

The *ϵ-greedy* or *semi-uniform random exploration* strategy (Watkins, 1989) is a simple mechanism for trading off the exploration of the uniform random policy against the exploitation of the greedy policy. There is a small probability ϵ at each time step of picking an action at random, otherwise the greedy policy is followed. With a good choice of the value for ϵ, the policy will quickly converge to one which selects the optimal action with probability $(1 - \epsilon)$.

Boltzmann Strategy

The *Boltzmann* or *softmax* exploration strategy (Luce, 1959) is a slightly more sophisticated strategy. It is based on the Boltzmann distribution, which has its origins in statistical mechanics, but also occurs in computer science in algorithms such as optimisation by simulated annealing (Kirkpatrick et al., 1983). While ϵ-greedy assigns equal probability to all actions when a random selection is performed, the Boltzmann strategy weights the probabilities using the $Q(s, a)$ values for the current state. So while the action choice is still random, actions leading to higher rewards will have a greater probability of being selected (see Figure 3.2).



	Probability of choosing action			
	a	b	c	d
ε − *greedy* $\varepsilon = 0.2$	0.85	0.05	0.05	0.05
Boltzmann $T = 2.5$	0.85	0.12	0.02	0.01
Boltzmann $T = 5.0$	0.62	0.23	0.08	0.07

Figure 3.2: The ϵ-greedy strategy chooses the greedy action with probability $(1-\epsilon)$ and otherwise chooses randomly using a uniform distribution. The Boltzmann strategy does not explicitly distinguish greedy and random actions. Instead the overall probability of choosing an action is weighted by the action's $Q(s,a)$ value.

How much the probabilities are affected by the state-action values is determined by the *temperature* parameter T. This allows us to choose from a spectrum of policies ranging smoothly from fully-random to fully-greedy, depending on our choice of value for T. High temperatures make the action choice more random, low temperatures encourage greedy behaviour. In state s, the probability of selecting action a_n is given by the distribution:

$$P(a_n) = \frac{e^{Q(s,a_n)/T}}{\sum_i e^{Q(s,a_i)/T}}$$

Decaying Parameter Values

Both the ϵ-greedy and Boltzmann strategies are compatible with the theoretical conditions for Q-learning to converge to the optimal value function. In addition, they both tend to focus the exploration effort along paths in the state space which have been shown to lead to good rewards, resulting in fast convergence. However, for *fixed* values of the ϵ and T parameters, the control policy will continue to select sub-optimal actions, even when the value function is arbitrarily close to the optimum $Q^*(s,a)$. In practice, once the value function is reasonably close to $Q^*(s,a)$ it is desirable to make *greedy* choices in the value function. This can be achieved by gradually decaying the value of parameter ϵ (or T).

For the SARSA algorithm, decaying the value of ϵ (or T) is even more important if we want to learn the *optimal* value function. Since SARSA is an *on-policy* algorithm, it will converge to the optimal value function only if the control policy tends towards greedy actions over time. Without decaying the parameter values SARSA will converge to the value function for the *control policy*, not the value function for the *optimal policy*.

Successful applications of the ϵ-greedy and Boltzmann strategies are strongly dependent on:

1. choosing a good initial parameter value to ensure enough exploration occurs.
2. decaying the parameter at the correct rate so that exploitation can take place once the value function is close to the optimum.

Unfortunately there is no analytic method to determine a suitable initial value and decay rate for a given problem. Suitable values therefore need to be determined by trial and error.

3.3.2 Directed Exploration Strategies

The ϵ-greedy and Boltzmann strategies require no extra state to be stored in addition to the table of $Q(s, a)$ values. There is therefore no explicit record of which areas of the underlying MDP have been explored. Instead we rely on the fact that if enough random actions are taken over a long time interval it is probabilistically likely that all areas of the MDP will be explored. This is what makes the choice of the initial value and decay rate of ϵ or T so vital to the success of these strategies. Since there is no way to detect when enough exploration has taken place these values must be selected by trial and error.

More complex exploration strategies have been developed which store additional information during learning to track the progress of exploration. This allows a more informed decision to be made as to when enough exploration has taken place. Thrun (1992) uses the term *directed* to describe such exploration strategies. In contrast, ϵ-greedy and Boltzmann are known as *undirected* exploration strategies. In addition, Thrun (1992) proposes a classification of exploration strategies based on which information influences exploration decisions:

Utility-based Strategies which are based on the estimated value of each state-action pair, i.e. value function information. Actions which lead to large rewards are explored more often. The ϵ-greedy and Boltzmann strategies are examples of utility-based strategies.

Counter-based Strategies which store a count of the number of times each state-action pair is visited. Actions which have small counter values are explored more often, which drives the agent towards areas of state space which are not well-explored. See Sato et al. (1988) for an example.

Recency-based Strategies which measure how much time has passed since each state-action pair was visited. Actions which have not been visited for some

time are favoured for exploration, which again drives the agent towards poorly-explored areas of state space. See Sutton (1990) for an example.

Error-based Strategies which measure how much the $Q(s, a)$ value of each state-action pair changes during updates. Actions which have recently undergone large changes in value are assumed to have larger error. These actions are favoured for exploration in order to reduce the error. See Thrun and Möller (1991) for an example.

Directed exploration strategies can also be either *local* or *distal* (Wyatt, 1997). *Local* strategies only use information about the current state (such as counters or recency information) to decide whether to explore. *Distal* strategies consider in addition the *long-term* exploratory benefits of actions. For instance, consider a state with two actions which have both been well-explored, but where the second of these actions would allow the agent to reach an unexplored state in several time steps. A local strategy in this state would not detect the exploratory benefit of the second action, whereas a distal strategy would. Measures of exploratory worth such as those described by Thrun (1992) can be back-propagated in distal strategies using dynamic programming or temporal difference updates. For a detailed discussion and an empirical comparison of local and distal strategies the reader is referred to Meuleau and Bourgine (1999).

Most directed strategies are heuristic approaches without any formal justification. They are generally inexpensive computationally, allowing them to outperform undirected strategies both in terms of sample complexity and computational effort. The main disadvantage of heuristic approaches is that they often require parameter selection and tuning for each new application domain.

Kearns and Singh (2002) describe a model-based algorithm utilizing a counter-based exploration strategy. This algorithm is interesting because its learning time can be polynomially bounded, whether the time is measured by environment time steps or computational operations. Given upper bounds on the mean and variance of the reward function in any state of an MDP, there is probability of $(1 - \delta)$ that their algorithm will learn a policy where the discounted return in all states is $\geq V*(s) - \epsilon$ in a time bounded by an expression polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$. This is mainly of theoretical interest, but it does suggest that in the future it may be possible to develop efficient directed methods which also have a formal basis.

3.3.3 Bayesian Approaches to Exploration

Asymptotic convergence results for Q-learning and SARSA make very few assumptions about the MDP in which learning takes place (Jaakkola et al., 1994; Singh

et al., 2000). The only major restriction is that the variance of the rewards received from each state-action pair must be finite. When such a wide range of environments are possible it is difficult to estimate the potential benefit of an exploratory action. Thankfully most of the problems we want to solve with RL do not require this generality. The mean and variance of the reward function can usually be bounded. In other cases the reward function may be completely known before learning begins, leaving only the transition probabilities unknown. When we have this kind of prior knowledge about the distribution of the underlying MDP model, a mathematically rigorous way to reason about exploration is to use a Bayesian statistical framework.

In Bayesian statistics, a *prior distribution* models the initial uncertainty. As new data is collected a *posterior distribution* can be calculated which reflects how the uncertainty has changed after observing the data. In RL the unknown parameters are the transition probabilities and reward distribution for each state-action pair. The transition probabilities define a *multinomial* distribution over successor states. The uncertainty in these probabilities can therefore be modelled with a *Dirichlet distribution*[2]. Rewards may be drawn from any underlying distribution, but are usually modelled using a *Gaussian distribution.*

Given a model of our uncertainty in the underlying MDP, how can we decide when it is worth exploring? Since integrating probabilities over the entire distribution of MDP models is unlikely to be feasible, approaches to date have been based on *sampling* the distribution of models. In Dearden et al. (1999) dynamic programming is used on the sampled models to generate estimates of the optimal $Q(s, a)$ values for the underlying MDP. A separate Gaussian distribution is then used to model uncertainty in each set of $Q(s, a)$ estimates, which is used to guide exploration based on an *information gain* criterion. Strens (2000) proposes a simpler scheme where a single sample from the distribution of models (a "hypothesis") is generated at the start of each of a series of finite length "trials." Dynamic programming is used to create an optimal policy for the hypothesis model. This policy is then used to select actions during the trial.

Approaches based on sampling from a distribution of MDP models are computationally expensive, since optimal policies for each sampled model must be found by dynamic programming. To avoid this expense, a number researchers have proposed Bayesian-inspired approaches on a smaller scale. One such approach is to adopt a local view each state of the MDP as a *bandit problem.* A bandit problem

[2]In most cases a state-action pair has only has a non zero transition probability for a few destination states. Therefore it is usually necessary for efficiency reasons to model the distribution with a representation suited to a sparse distribution. See Dearden et al. (1999) for further details.

(Berry and Fristedt, 1985) is a single state problem where the goal is to identify by trial and error which of the available actions has the highest expected reward. Optimal solutions (in a Bayesian sense) can be calculated for many bandit problems (Gittins, 1989). We can view each state of an MDP as a bandit problem where the "reward" for taking an action is the optimal discounted return $Q^*(s,a)$. Unfortunately, in RL the value of $Q^*(s,a)$ is initially unknown, and must be approximated by its estimate $Q(s,a)$. The local bandit problem for each state is therefore *non-stationary*, which makes some kind of *forgetting* mechanism necessary if $Q(s,a)$ is used for the bandit's reward.

The *interval estimation* method (Kaelbling, 1993b) is a non-Bayesian strategy which adopts the local bandit problem view. Uncertainty in the value of each state-action pair is modelled using a Gaussian distribution. Some small probability α is chosen, and for each action an upper bound is calculated so that the true value is below the bound with probability $(1-\alpha)$. The action with the highest upper bound is always chosen for execution. If the action turns out to be a poor choice, the upper bound will be reduced as the statistics are updated. If the action is a good choice, the upper bound will remain high and the action will continue to be selected in that state.

Meuleau and Bourgine (1999) present a method similar to interval estimation, based on a Bayesian technique for bandit problems using *Gittins indices* (Gittins, 1989). A related Bayesian approach which models the uncertainty of each $Q(s,a)$ value during the progress of Q-learning is presented by Dearden et al. (1998).

3.3.4 External Sources of Exploration

In the standard reinforcement learning setting the learning agent is *tabula rasa*. This means that the agent begins the learning process with absolutely no knowledge about how its environment behaves. In the Bayesian RL framework the tabula rasa assumption is relaxed, since the agent is provided with a *prior* distribution which models and quantifies the agent's uncertainty of the behaviour of the environment. In both cases the responsibility for making intelligent exploration decisions rests entirely with the agent.

In particularly complex environments with sparse rewards, the learning time can be shortened significantly by using an *external source* to perform the initial exploration (Smart and Kaelbling, 2000). The external source could be a human controlling the system, or a hand-coded policy. This introduces an element of *teaching* into the RL process, making the initial phase of learning similar to *behavioural cloning* (Sammut, 1996). The advantage of this approach is that sparse rewards can be quickly uncovered by the external source, but the reinforcement

learner can go on to learn a policy which improves on the performance of the external source. Bentivegna et al. (2004) use such techniques to develop robotic systems capable of playing air-hockey and a marble-maze game.

The disadvantage of this approach is that the quality of the learned policy is strongly dependent on the quality of the external source. If the external source is very sub-optimal, it is possible that exploration will be insufficient to avoid converging to a local optimum. In the worst case, if there is no known reasonable hand-coded policy for a domain, and the task is beyond a human controller, this approach is probably inapplicable. Conversely, Driessens and Džeroski (2002) discovered that it can be problematic if the external source is too close to the optimal policy, since a learner observing only optimal actions may not be able to distinguish between good and bad action choices.

3.3.5 Limitations of Improving the Exploration Strategy

Extensive previous research and continued interest in exploration techniques reflects the fundamental nature of the exploration-exploitation trade-off in RL. A good exploration strategy is vital for learning the optimal policy in a reasonable time. A variety of both simple and complex strategies were surveyed in this section. The more complex strategies require fewer explorative actions at the expense of greater computational effort. Simple strategies such as ϵ-greedy and Boltzmann tend to be preferred when the environment is simulated, since environmental experience is cheap and plentiful. Directed and Bayesian strategies become most useful when experience in the environment is limited or expensive to obtain.

Despite the gains that the directed exploration strategies afford us, there is a limit to how far they can make large reinforcement learning problems tractable. The difficulty of large reinforcement learning problems is primarily due to the exponential growth of the state space as the number of state features is increased. While it may not be necessary to visit all of these states if there is a known bound on the reward function, it remains likely that a large subset of the state space must be repeatedly visited to establish the optimal policy within some reasonable error bound. So even if we had access to a perfect exploration strategy, the sample complexity would still increase exponentially. To tackle this problem, we need either to use some technique to reduce the size of the state space, or to generalise between similar states so that we do not need to visit all state-action pairs. Techniques for dealing with the state space explosion are discussed in Sections 3.4, 3.5 and 3.6.

3.4 Value Function Approximation

In many situations, representing the function $Q(s, a) \rightarrow \mathbb{R}$ explicitly as a table of real numbers will result in some degree of redundancy. If two states are *similar in terms of state features*, it is likely that a given action will have similar value if taken in either of the two states. To take advantage of this property, a representation is required for $Q(s, a)$ which will allow us to *generalise* between similar states. There are a number of advantages to generalisation:

- By removing redundancies in the table of $Q(s, a)$ values, the function can be represented more compactly in memory.
- Each update from experience affects more than one state, which can accelerate convergence to the optimal value function.
- Values of states which were not encountered during learning can be estimated.
- Learning can take place in domains with *continuous* state spaces.

This form of generalisation in reinforcement learning is known as *value function approximation*, since the goal is to create an approximation of the entire value function from a limited number of examples. Numerous existing techniques for function approximation can be used from the fields of inductive concept learning, pattern recognition and statistical curve fitting. In this section the application of such techniques to reinforcement learning is considered.

Historical Remarks

Recent research has focused on applying function approximation techniques to the popular Q-learning (Watkins, 1989) and TD(λ) (Sutton, 1988) algorithms, which were both originally presented with the assumption of an exact tabular representation. However, learning an approximate value function is a concept which originated in early game-playing programs. In his seminal work on the design of chess programs, Shannon (1950) first suggested that a program could learn from the outcome of each game it played by changing the coefficients of the evaluation function used to rate positions of the board. Samuel (1959) implemented such a program to play the game of checkers, using among other techniques an ad hoc method for re-estimating the value of board positions based on the estimated value of a board position encountered several moves later, a technique with many similarities to the more general TD(λ) algorithm. The evaluation function in Samuel's program is a linear combination of numerical features of the board position.

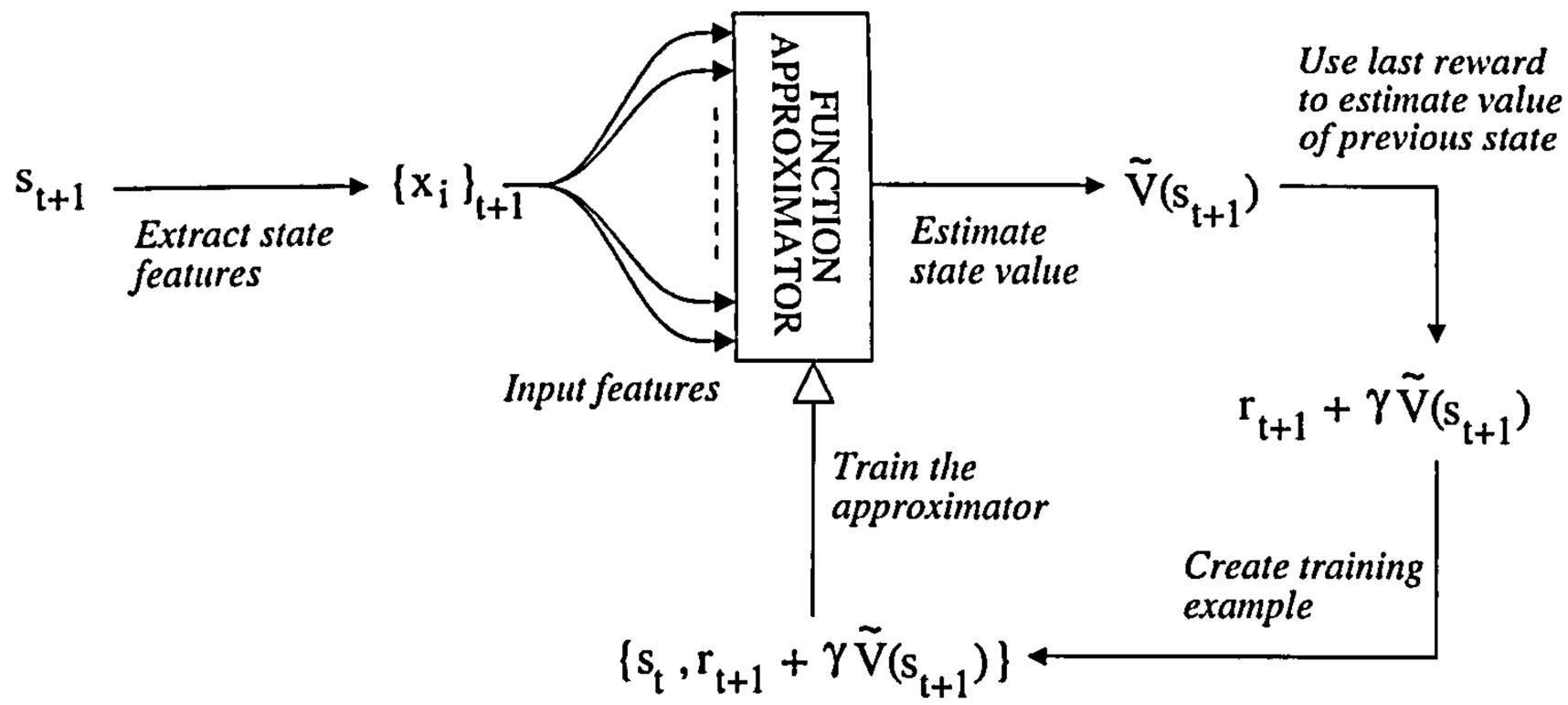


Figure 3.3: Instead of performing backup operations in a table data structure, experience can be used to generate training examples for a function approximator.

There has also been a wide range of research on combining function approximation with dynamic programming (see Section 2.3), using approximators such as orthogonal polynomials (Bellman and Dreyfus, 1959) and splines (Daniel, 1976). However, most of these approaches assume that an exact model of the environment is available for use in calculations, and so they are not directly applicable to reinforcement learning.

3.4.1 Fundamentals of Approximation

Consider using an algorithm such as TD(λ) (Sutton, 1988) to calculate the value function V^π for some policy π. The estimated value function at time t is written as V_t. When function approximation is combined with the TD(λ) algorithm, V_t is no longer represented as an exact table, but in a compact form which approximates the table. The approximator is often a *parameterized function*, which represents V_t using a fixed size parameter vector $\vec{\theta_t}$. Other approximators are based on a finite database of experiences recorded during training. In either case, the number of parameters or experience data points is usually much smaller than the total number of states.

Each TD(λ) backup, which would usually update individual values in the table representation, can now be used as a *training example* for the function approximator, as shown in Figure 3.3. For example, in the simplified case of TD(0), the value of the state encountered at time t, $V_t(s_t)$, is re-estimated using the value of the subsequent state as $r_{t+1} + \gamma V_t(s_{t+1})$. To improve the function approximation, we use the tuple $(s_t, r_{t+1} + \gamma V_t(s_{t+1}))$ as a training example for the approximator.

This approach allows us to choose from a wide range of existing *supervised learning* algorithms for purposes of function approximation. Not all supervised

learning techniques are equally appropriate however. The distribution of training examples will appear non-stationary until V_t becomes a good approximation of V^π. Some neural-network methods require multiple passes over a static training set, and would be poorly suited for reinforcement learning.

Note that although much of the discussion in this section is framed in terms of learning the state value function $V^\pi(s)$, these methods can apply equally well to approximate the state-action value function $Q^\pi(s,a)$.

3.4.2 Comparing Function Approximation Techniques

How can we compare the performance of two function approximation methods? Since many supervised learning methods seek to minimize the *mean squared error (MSE)* over the training examples, a criterion often used to compare function approximations in a reinforcement learning context is:

$$MSE(\vec{\theta_t}) = \sum_s P(s)[V^\pi(s) - V_t(s)]^2$$

Here $P(s)$ is a probability distribution which weights the error values according to the likelihood of arriving in a particular state. Note that for *on-line* reinforcement learning, $P(s)$ is dependent on the policy being used to explore the environment. If we change the policy over time (e.g. to approach the optimum) then $P(s)$ will also change over time. This is a source of instability when function approximation is combined with on-line learning.

It is arguable whether this is the best criterion with which to grade approximations, since the greedy policy derived from a parameter vector $\vec{\theta_t}$ which minimizes $MSE(\vec{\theta_t})$ can often be outperformed by a greedy policy derived from some other value of $\vec{\theta_t}$. However, on the basis that a value function which minimizes the MSE will result in good performance, there are two important properties of our method to be determined:

- How close to the optimum is the V_t which minimizes $MSE(\vec{\theta_t})$? In other words, what is the most accurate value function *representable* in our function approximator?
- Does the combination of the reinforcement learning and function approximation methods selected guarantee *convergence* to this minimal $MSE(\vec{\theta_t})$ approximation?

It is the second of these properties which is most problematic, and is discussed in more detail in Section 3.4.7.

3.4.3 Linear Approximation Methods

Function approximators based on a linear combination of basis functions offer a number of advantages. They are backed by strong mathematical theory. There are also efficient algorithms for performing gradient descent on the parameters of a linear approximator. These algorithms converge to a *global optimum* which minimizes the MSE over a static training set.

The functional structure of a linear approximator is illustrated in Figure 3.4. Given a set of n basis functions $\{\phi_i(s)\}$ and a vector of n parameters $\vec{\theta}$, we can express the linear approximation of the value function as:

$$V_t(s) = \sum_{i=1}^{n} \theta_i \phi_i(s)$$

The correct choice of the basis functions $\{\phi_i(s)\}$ is an important factor in determining the success of a linear approximator. In making this choice, we can exploit any prior knowledge we may have about the learning problem to select input features which are good discriminators for the value function. *Feature selection* is a vital stage for most supervised learning methods, both linear and non-linear.

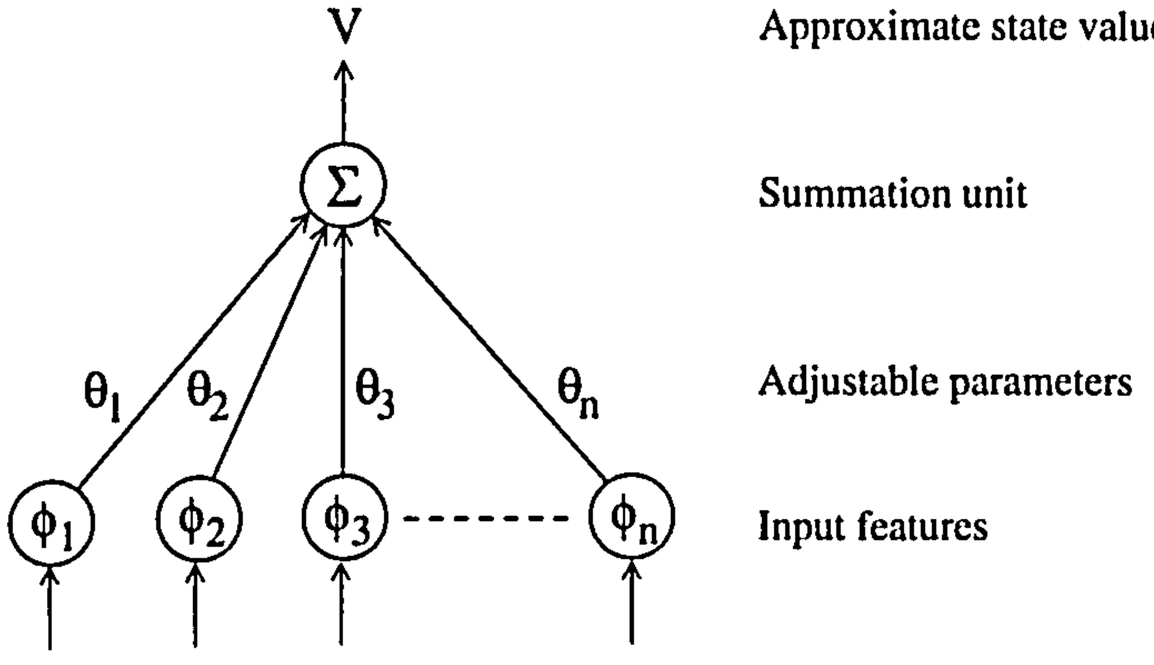


Figure 3.4: Linear approximation architecture for learning a value function.

Coarse Coding

We could use the unmodified state variables (whether discrete or continuous) as the basis functions, but this is unlikely to be a successful approach, given the limited representational power of a linear approximator. A more suitable set of basis functions can be constructed using *coarse coding* (Hinton et al., 1986). Each feature in coarse coding is defined as a region of the state space, and the basic approach is to use a large set of *overlapping* features which between them cover the whole state space. Coarse coding generally uses binary features, which have value 1

if the current state lies within the receptive field of the feature, and otherwise have value 0. It turns out that using relatively coarse features with significant overlap is more effective in most situations than using fine-grained disjoint features, although if features are too coarse then it becomes difficult to represent fine-grained changes in the true value function.

Tile Coding

Tile coding[3] (Albus, 1981) is a form of coarse coding which has proved to be a popular function approximation technique for reinforcement learning (Watkins, 1989; Lin and Kim, 1991; Sutton, 1996). In tile coding, we define sets of features, each set being an *exhaustive partition* of the state space. Each of these sets is known as a *tiling*, and each feature in the set is called a *tile*. The multiple tilings are each offset by a different amount in the state space (see Figure 3.5), which improves the generalisation achievable by the approximator. The tilings need not be uniform grids, an arbitrary partitioning strategy can be used. Tile coding is generally combined with *hashing* techniques to reduce memory requirements. This compresses a large tiling into a smaller set of tiles, each tile being composed of several non-contiguous regions spread randomly over the state space.

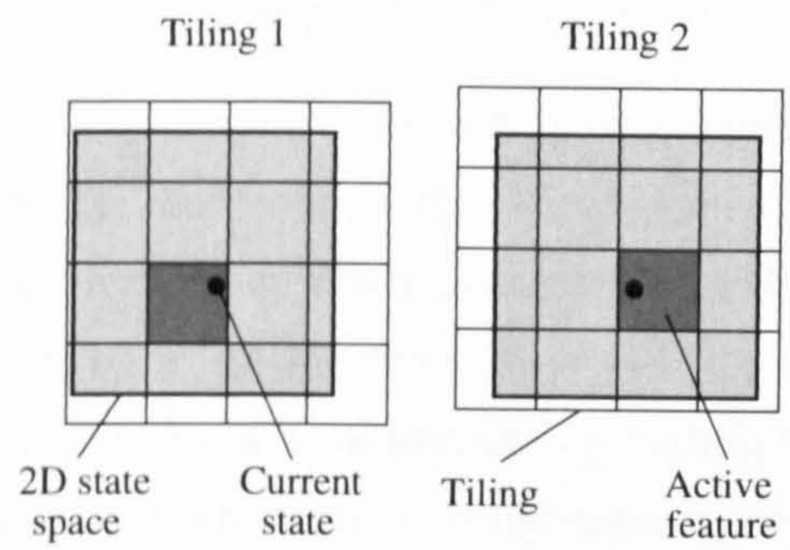


Figure 3.5: *Tile coding* uses coarse binary state features, arranged into a number of overlapping tilings.

3.4.4 Memory-Based Approximators

Memory-based approximators are a family of approximation methods which are not based on a parameterized functional model. Instead a finite number of training examples are simply stored in memory for later use. After training, the values of new states are approximated using a subset of examples whose state features are most similar to the new state. The set is determined as and when a new state

[3] A tile coding approximator is sometimes known as a *Cerebellar Model Articulation Controller (CMAC)* since this was the original name used by Albus (1981).

value needs to be estimated, i.e. at *query time* rather than at *learning time.* This kind of approach is sometimes known as *instance-based learning* or *lazy learning.* The *k-nearest neighbour* algorithm (Cover and Hart, 1967) is an example of a memory-based approximator. Using this algorithm, the approximate value of a state would be the mean value of the k most similar states encountered during training, where k is some constant. Sheppard and Salzberg (1997) use a variant of k-nearest neighbour as a value function approximator for Q-learning, resulting in a method which they term *lazy Q-learning.*

Kernel Methods

Kernel methods (Shawe-Taylor and Cristianini, 2004) are powerful memory-based machine learning methods that have only recently been used for reinforcement learning. The core idea in these methods is to map examples into an *implicit* feature space $\phi(\mathbf{x})$ (where $\mathbf{x}$ is the vector of state variables and ϕ is a mapping to a rich space of state features). Since $\phi(\mathbf{x})$ can contain a large (or even infinite) number of features, or be expensive to evaluate for other reasons, we avoid evaluating $\phi(\mathbf{x})$ explicitly by defining a *kernel function* k:

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$$

The value of $k(\mathbf{x}, \mathbf{x}')$ is the *inner product* of two states mapped into the rich feature space, which can be seen informally as a measure of the *similarity* of the two states. Note that evaluating ϕ is not necessary to calculate k. ϕ remains an *implicit* feature space arising from the choice of kernel function k. Kernel methods not only benefit from the computational saving of avoiding the evaluation of ϕ, but also allow powerful *domain-independent* kernel-based algorithms to be developed, which can then be tailored to a specific application with a *domain-specific* kernel function. Kernel methods can generalise very effectively from only a very small number of stored training examples.

Some researchers have begun assessing how kernel methods can be used for value-function approximation in RL. Ormoneit and Sen (2002) demonstrate the robustness of a kernel-based reinforcement learning algorithm in a theoretical context. The empirical performance of reinforcement learning algorithms based on *Support Vector Machines* (Dietterich and Wang, 2002) and *Gaussian processes* (Rasmussen and Kuss, 2004) has also been investigated. A limitation of these approaches is that they all rely on offline processing. Developing effective online kernel-based reinforcement learning algorithms is an active area of research.

3.4.5 Decision Tree Approximators

A number of researchers have developed function approximation approaches which draw their inspiration from *decision tree*[4] learning algorithms such as ID3 (Quinlan, 1986). Chapman and Kaelbling (1991) presented the *G-algorithm*, which incrementally builds a tree-structured value function. In the G-algorithm, a state is represented as a binary string, with each bit in the string mapped to a binary state feature. As experience is gathered, a standard statistical test known as the *Student's 't' test* is used to determine which of the bits in the string are relevant to making optimal action choices, and the decision tree is split each time a relevant bit is discovered. Note that this approach will only have good performance in situations where the applicability of each action depends only on a small subset of the state features. This work was later extended in scope by Pyeatt and Howe (1998), who removed the assumption of binary state features. They also compared the performance of four different statistical tests for splitting the decision tree nodes. An example of a Q-function represented as a decision tree is shown in Figure 3.6.

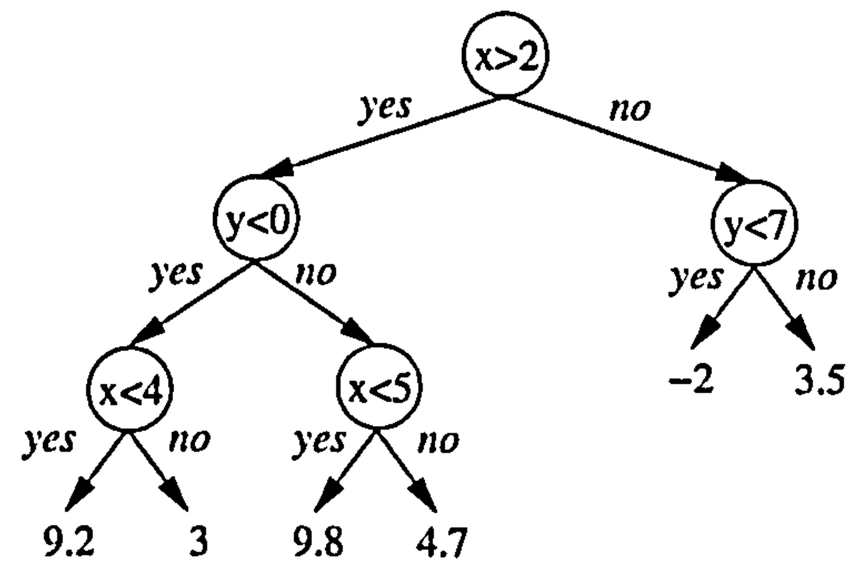


Figure 3.6: A decision tree representing the value function for a single action in a continuous state space of two dimensions.

The *kd-tree* data structure (Moore, 1990) has similarities both to decision trees and to the quad-tree data structure used in computer graphics. Function approximation using a kd-tree is a variable resolution method—the state space is represented at a fine grain resolution only in the areas where it is needed for optimal decision-making. The kd-tree approach is most useful for non-linear approximation in problems with continuous state variables and high dimensionality. This approach eventually developed into the Parti-game algorithm (see Section 3.5.2).

[4]In the statistics community, a decision tree mapping each input to one of a finite set of classes is known as a *classification tree.* A decision tree mapping each input to a real-valued output is known as a *regression tree.* While these names are more descriptive in some contexts, in this thesis I follow the convention of Pyeatt and Howe (1998) and use *decision tree* to refer to both kinds.

3.4.6 Neural Network Approximators

The *back-propagation neural network* (Rumelhart et al., 1986) is a popular function approximator in the wider machine learning community. Many researchers have investigated its usefulness for reinforcement learning. An example of a multi-layer neural network is shown in Figure 3.7. Anderson (1987) employed a multi-layer back-propagation network as a function approximator for RL with the *Adaptive Heuristic Critic (AHC)* algorithm (Barto et al., 1983), using this approach to solve the pole-balancing problem. Lin (1992) used neural network approximations with both the AHC and Q-learning algorithms, examining the performance of learning agents in a grid-world with food, enemies and obstacles. Tesauro (1995) trained the highly successful TD-GAMMON backgammon program using the TD(λ) algorithm and a neural network. Zhang and Dietterich (1995) learned strategies for job-shop scheduling using a similar approach with TD(λ).

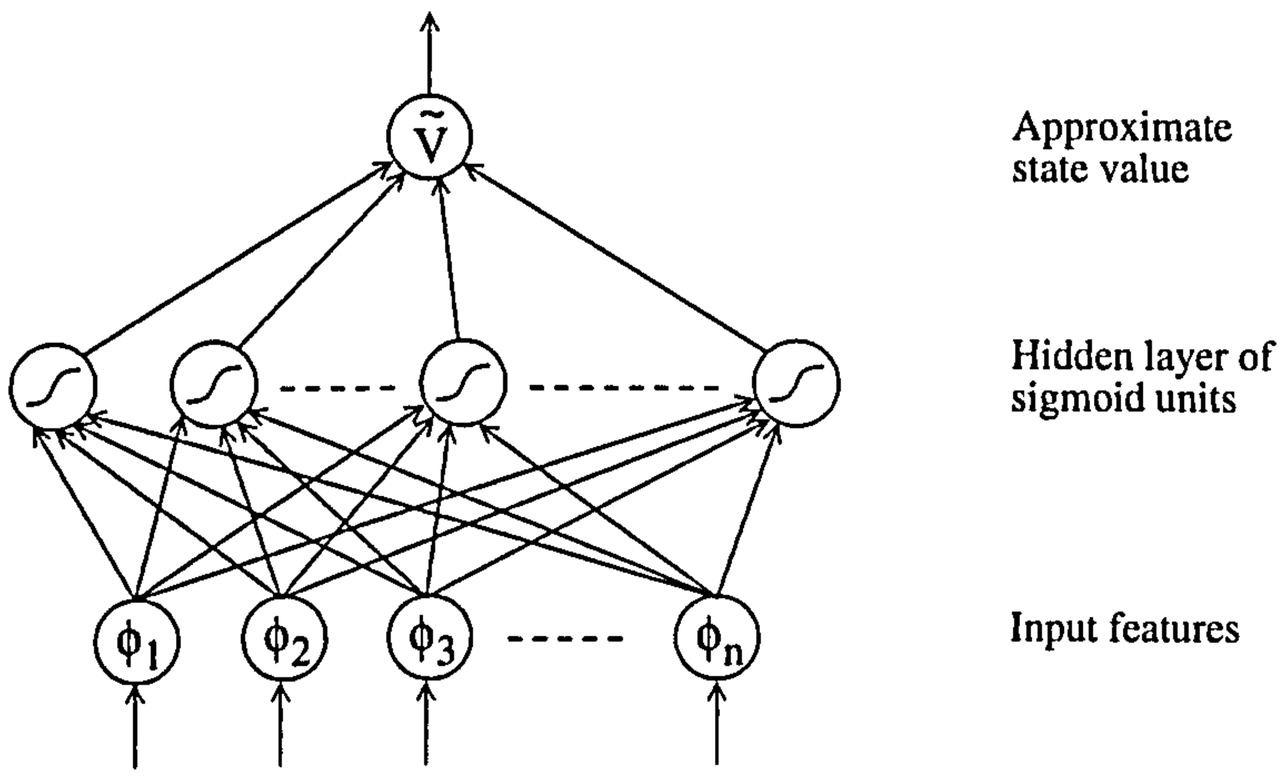


Figure 3.7: A neural network approximator which could be used for learning a value function.

Despite the positive results listed above, the combination of RL and neural networks can result in slow learning, convergence to a sub-optimal policy, oscillation or even divergence. Anderson (1986) used such a combination to solve the pole-balancing and "Towers of Hanoi" problems, and notes the slow convergence in both cases. Shepanski and Macy (1987) used neural network approximation in a simplified driving simulator. The initial learning phase (after the network weights were randomly initialised) exhibited great instability, and often failed to converge. When convergence did occur, the weights exhibited oscillatory behaviour for some time before becoming stable. A more complex driving simulation was used as the basis for experiments conducted by Barreno and Liccardo (2003) with neural network approximation for RL. These experiments exhibited convergence problems, while similar experiments with a *linear* approximator produced good results.

3.4.7 Convergence Problems and Guarantees

The mixed success of the above experiments is symptomatic of more general problems with using function approximation in RL. With most of these algorithms there is no formal guarantee that they will converge to a policy whose value function approximation has a small $MSE(\vec{\theta}_t)$ error value (see Section 3.4.2). It is also fairly easy to construct particular examples to demonstrate empirically that the approximation does not always converge, and in some cases even diverges (Bradtke, 1993; Baird, 1995; Tsitsiklis and van Roy, 1997).

Boyan and Moore (1995) evaluated quadratic regression, locally weighted regression, and neural networks as possible function approximators for dynamic programming (using the value iteration algorithm). Each of these approximation methods was shown to exhibit divergent behaviour in one of several simple grid-world problems. However, using the same set of problems, Sutton (1996) showed that an RL approach using the SARSA algorithm (Rummery and Niranjan, 1994) and linear tile coding (Albus, 1981) could learn robustly and efficiently in all of the problems. These results indicate that the approximation architecture must be carefully matched to both the learning algorithm and the domain if good performance is to be achieved.

How can we determine which function approximator will work well with a particular RL algorithm? Deriving a theoretical guarantee of convergence for the algorithm would be a useful first step. However, the formal guarantees for Q-learning (Watkins and Dayan, 1992), SARSA (Singh et al., 2000) and TD(λ) (Jaakkola et al., 1994), which were cited in Section 2.4, were all derived based on the assumption that an exact tabular representation was used, not a function approximator. Some researchers have begun extending these results to include the use of function approximation (Gordon, 1995; Tsitsiklis and van Roy, 1996; Gordon, 2001). Perhaps the most significant result proved so far is that of Tsitsiklis and van Roy (1997) where it is shown that, under a number of assumptions, the TD(λ) algorithm combined with a *linear* function approximator will converge to a near-minimal MSE solution $\vec{\theta}_\infty$ with probability 1. The near-minimal solution is related to the minimal solution $\vec{\theta}^*$ by the inequality:

$$MSE(\vec{\theta}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\vec{\theta}^*)$$

Note that the parameter λ strongly affects the quality of the theoretical bound for TD(λ). If $\lambda = 1$ (equivalent to Monte Carlo policy evaluation) then the algorithm will eventually converge to the minimum MSE solution $\vec{\theta}^*$. However, in practice much faster convergence can be achieved with smaller values of λ.

A key assumption in the derivation of this bound is that the distribution of training examples is the *on-policy distribution*, i.e. training examples are generated by following the policy being evaluated by TD(λ). The success of this approach suggests that an *on-policy* algorithm such as SARSA(λ) may form a more successful combination with *linear* approximation than the *off-policy* Q-learning algorithm. In fact, for purposes of *estimating* the value function of a fixed policy, a modification of the proof of Tsitsiklis and van Roy (1997) may be used to prove convergence of SARSA(λ) using linear approximation. This means that SARSA(λ) can be used as the estimation step of an *approximate policy iteration* approach which has strong convergence guarantees (Perkins and Precup, 2002). However, the theoretical basis for SARSA(λ) with linear approximation is much weaker when the control policy uses an exploration strategy which is greedy in the limit (Gordon, 2001). In spite of this, SARSA(λ) with linear approximation has proved to be very successful in practice (Sutton, 1996) and remains a popular approach to generalization in RL.

A different formal approach is necessary to derive guarantees for using function approximation with an *off-policy* algorithm such as Q-learning. Thrun and Schwartz (1993) suggest that using approximation with value iteration methods such as Q-learning is inherently dangerous because errors in the value function due to generalization can interact poorly with the "max" operator used in the definition of the value function.

How can we avoid the instability which arises in approximate value iteration methods? One approach is not to use approximators which *extrapolate* from the observed training examples, as polynomial regression does. Gordon (1995) defines a class of function approximators called *averagers*, which includes local-weighted averaging, k-nearest neighbour and Bézier patches. It is shown that when the transition and reward functions are known in advance, approximate value iteration with an averager is guaranteed to converge to a function whose *max norm* error has a bounded difference from the minimum max norm error solution representable by the averager.

Another possible approach is to change the form of the error being minimized from mean-squared error to *mean-squared Bellman error* or *residual error* (Baird, 1995), which can be expressed as:

$$\sum_s P(s) \left[E_\pi \{ r_{t+1} + \gamma V_t(s_{t+1}) | s_t = s \} - V_t(s) \right]^2$$

In practice these methods tend to converge more slowly than the simpler function approximators, and the formal convergence guarantees which have been proved so far are often inapplicable to *off-policy* algorithms such as Q-learning.

3.4.8 Limitations of Function Approximation

It is evident that many problems still exist for function approximation in reinforcement learning. Successful generalization can often be achieved with careful choice of algorithm and approximator, but the field still lacks a coherent formal framework to ensure stable convergence to a near-optimal policy.

Function approximators of the kind discussed in this section also have two key limitations:

- Function approximators are heavily reliant on *prior knowledge* to provide a set of good input features for learning.
- Function approximators are most effective when the target value function has no sharp discontinuities between similar states.

If there are only a few discontinuities in the target value function, a decision tree could be an effective approximator if an algorithm with an appropriate inductive bias is used to build the tree. A memory-based approximator could also be effective if enough data can be concentrated in the region of each discontinuity. For highly-discontinuous value functions, however, both of these methods require too much memory to store enough data points (or decision nodes) to accurately approximate the target value function.

Sections 3.5 and 3.6 examine approaches to reinforcement learning using techniques such as *hierarchical decomposition* and *symbolic AI methods.* These approaches can be used for solving learning problems where function approximation is inappropriate because of the limitations mentioned above.

3.5 Hierarchical Reinforcement Learning

One way to approach the problem of scaling-up reinforcement learning is to adopt a *divide and conquer* strategy. Rather than attempting to solve an MDP for the whole problem at once, a *decomposition* is performed to create a *hierarchical* structure of sub-problems. Usually the structure of the decomposition is supplied as prior knowledge to an algorithm. This has resulted in a family of closely related methods known as *Hierarchical Reinforcement Learning* methods.

Many hierarchical reinforcement learning methods are not designed to find the true optimal policy. By using decomposition to constrain the scope of the learning which takes place, some solution quality is sacrificed to reduce the learning time required. By relaxing the requirement of true optimality, a policy with good performance can be found in orders of magnitude less time. Dietterich (2000b)

defines two forms of restricted optimality to describe the solution quality of some of these methods:

Recursive Optimality A *recursively optimal* policy is obtained when the policy at each node in the learning hierarchy is optimal given the policies to which its child nodes have converged.

Hierarchical Optimality Only a subset of all possible policies can be represented with a particular learning hierarchy. The *hierarchically optimal* policy is the best policy in this limited subset of policies.

A hierarchically-optimal method will return the best policy from a given policy space, so if the true optimal policy happens to lie in this space, it will be returned as the solution. However, this is very unlikely—the hierarchy is constructed to allow the problem to be solved in less time, not to preserve optimality. In general, the true optimal solution can only be found if we solve the problem in the flat state space, which is intractable for large problems.

The focus of this section is on methods which decompose a learning problem and use reinforcement learning as the *only* learning technique in the resulting hierarchy. It is worth mentioning that reinforcement learning is also a common component technique of *hybrid learning architectures.* These hybrid approaches are often based on a combination of high-level *deliberation* and low-level *reactivity*, as proposed by Gat (1997). *Layered learning* (Stone, 1998) is one of the most successful hybrid approaches of this kind. Stone uses reinforcement learning to learn low-level behaviours in the layered learning architecture for a successful RoboCup team (Kitano, 1998).

3.5.1 Parallel Decomposition

Consider a reinforcement learning problem where each action can be interpreted as a number of component actions executed *in parallel.* If the state space of the problem can be expressed as a set of features, where each feature is only affected by one of these component actions, then a *parallel decomposition* is possible. Essentially, this means that choosing an optimal action in the original is equivalent to choosing the optimal action for each of a set of sub-problems running in parallel.

A parallel decomposition for the MDP representing the problem results in a set of smaller MDPs. Each of these smaller MDPs represents one of the parallel sub-problems which characterise the original problem. The original state space is now represented by the Cartesian product of the state spaces of the smaller MDPs. Similarly, the original action space is represented by the Cartesian product of the action spaces of the smaller MDPs.

A traditional reinforcement learning algorithm can now be used to obtain a policy for each of the smaller MDPs, which are much less complex than the original MDP. These policies are used to select the optimal action for each sub-problem, and the combination of these component action choices (which can also be described as a *joint action*) is the action choice in the original MDP. This process is illustrated in Figure 3.8.

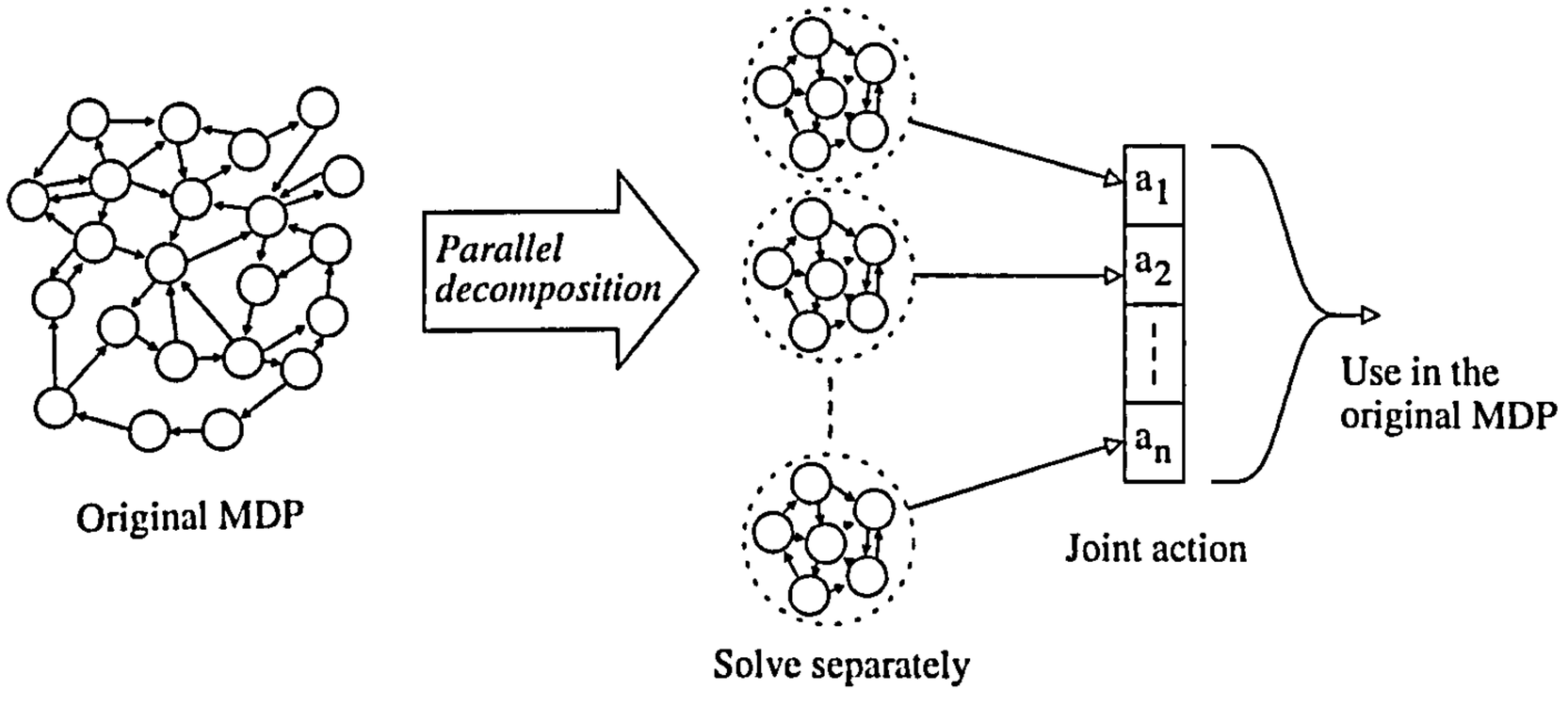


Figure 3.8: Some MDP problems can be decomposed into parallel sub-problems, which can be solved separately and their policies combined into a joint action.

Parallel decomposition is particularly appropriate for multi-agent problems. In many such problems each agent will spend a large proportion of its time reasoning and acting independently of other agents. If this the case, an MDP representing the entire multi-agent problem can be decomposed into an MDP for each agent. However, non-trivial problems rarely allow the agents to be assumed totally independent. There may be some shared resource that is consumed by all the agents. Alternatively, the coordination of individual actions may be necessary to get the largest pay-offs. If there are only a limited number of such interactions between agents, the parallel decomposition may still be applicable.

Parallel decomposition therefore has limited applicability—it is generally only suitable for problems where there are several parallel processes which are either independent, or have weak interactions such as shared resource constraints. If interactions exist, the optimal actions for two sub-problems may be mutually exclusive. Such conflicts must be resolved when the joint action is constructed. Meuleau et al. (1998) present a method for exactly this type of problem, where value functions learned for the smaller MDPs are used as heuristics for combining conflicting action choices into a joint action for the larger MDP.

Note that parallel decomposition is concerned with decomposing problems with *parallel structure* to make them easier to solve. In Section 3.7 *parallel reinforcement*

learning methods are surveyed, which exploit parallel hardware to quickly find solutions for RL problems which have no parallel structure.

3.5.2 State Aggregation and State Abstraction

State aggregation is a more generally applicable hierarchical approach. It is based on the grouping of sets of the original low-level MDP states into a number of high-level *abstract states*, as shown in Figure 3.9. State aggregation exploits the fact that very similar low-level states can be collected into a single abstract state, and can then all be considered identical with regard to the current learning problem. We can define a high-level MDP over the abstract states, which will be much easier to solve because the number of states is much smaller.

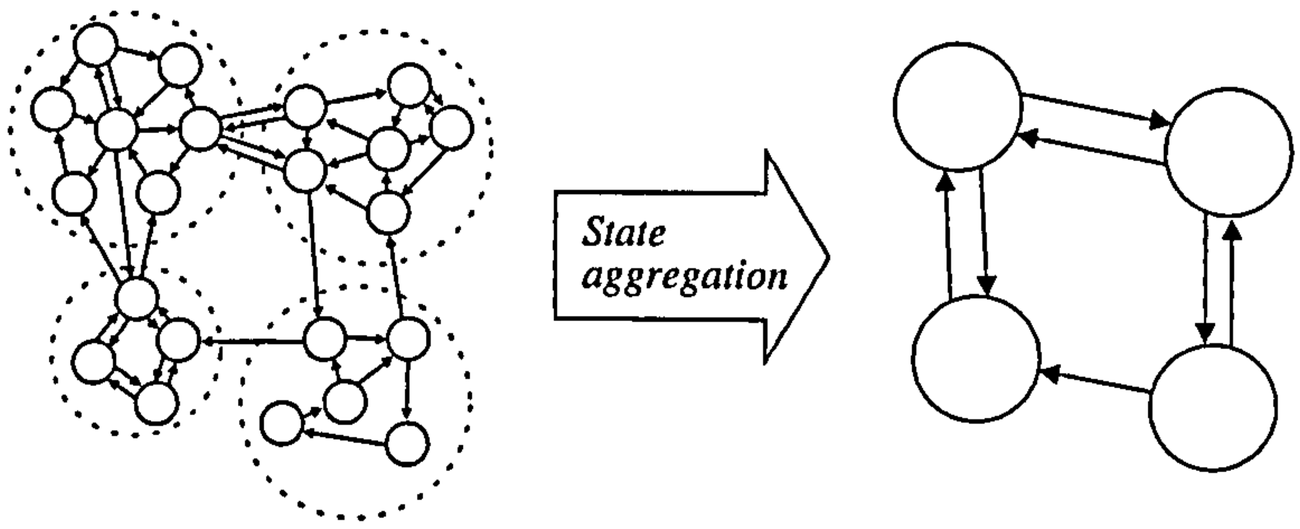


Figure 3.9: State aggregation groups together low-level MDP states, resulting in a much simpler MDP to be solved over high-level *abstract states*.

State abstraction is a particular form of state aggregation, which arises in situations where the state consists of a number of state features. If it is known that the value of a particular state feature is not needed for selecting the optimal action, we can reduce the size of the state space by aggregating together states which only differ in terms of this feature. This is a very effective way to reduce the size of the state space, which grows exponentially with the number of state features. However, determining without prior knowledge whether a state feature is relevant to a given problem is difficult, so we will generally need to supply suitable state abstractions to an algorithm using our background knowledge.

Hierarchical Distance To Goal

Early research exploiting hierarchy in reinforcement learning was principally based around state aggregation. The *Hierarchical Distance to Goal* or HDG algorithm (Kaelbling, 1993a) solves large navigation tasks by distributing landmarks over the space to be navigated, and aggregating states according to their closest land-

mark. Learning paths between landmarks is much less complex than learning paths between arbitrary states. This approach was later extended with the concept of *airport states* (Moore et al., 1999), which constitute a hierarchy of landmarks for learning routes at different levels of abstraction.

Feudal Reinforcement Learning

Feudal reinforcement learning (Dayan and Hinton, 1993) is another early state aggregation algorithm applied to navigational learning problems. While there are no explicit landmarks used in this algorithm, states are similarly aggregated according to spatial proximity. Aggregations are themselves aggregated together to create a hierarchy of abstraction levels. Feudal reinforcement learning uses the concept of a *manager*, an independent reinforcement learning agent which has responsibility over an aggregation at some level of the hierarchy. Each manager is subordinate to a manager at the level above, creating a feudal hierarchy of independent learners. All but the lowest layer of managers take action by ceding control to a sub-manager to achieve some sub-goal. The lowest layer of managers performs actions directly. When a sub-manager achieves its goal, it receives a reward from its manager for doing so, and control is returned to the manager. In this way, each manager gradually learns which sub-manager to choose to take action in each of the abstract states over which it has responsibility. A large state space is thereby decomposed into a set of independent learning problems, each with a state space bounded by the maximum number of elements in an aggregation.

Parti-Game

The *Parti-game* algorithm (Moore and Atkeson, 1995) is a *variable resolution* state aggregation algorithm. Instead of creating state aggregations of similar size over the entire state space, the algorithm begins with a single large aggregation, and splits it into smaller ones in regions of the state space which require fine-grained discrimination to produce a good policy. For each aggregation, the algorithm chooses the action which has proved from experience to make the best progress towards a goal region. If taking this action always results in the same transition to a new aggregation, there is no need to split the source aggregation—an effective policy choice has been identified for that area of state space. However, if an occasion arises where an unexpected transition occurs and the action fails to make good progress towards the goal, there must be a state within the aggregation which requires a different action choice, so the aggregation is split. This variable resolution approach can reduce the state space considerably, and has been applied

successfully to problems with high dimensionality and continuous state spaces. However, the Parti-game algorithm makes a number of restrictive assumptions, and is only applicable to deterministic problems with an explicit set of goal states. More recently, Munos and Moore (2002) have evaluated the performance of several different *splitting criteria* for a closely related variable resolution method.

3.5.3 Temporal Abstraction

State aggregation is an effective technique for reducing the state space of a reinforcement learning problem, but in the general case it may not be obvious how to travel from one abstract state to another. The state aggregation algorithms above were mostly applied to navigation problems, where moving between abstract states can be simple (e.g. take low-level actions heading towards the landmark which defines the destination abstract state). To apply hierarchical techniques to more general problems, we need policies to move between abstract states, which leads to the idea of *temporal abstraction.*

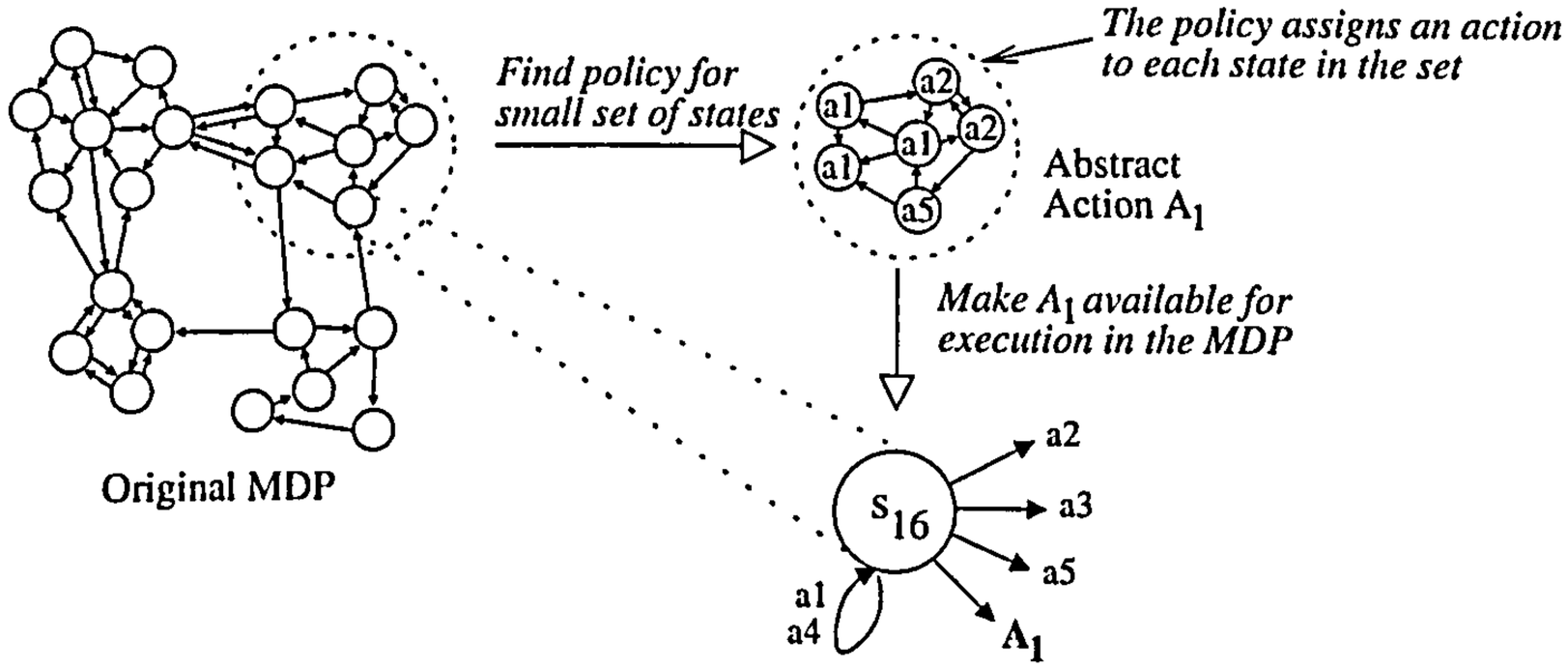


Figure 3.10: Temporal abstraction uses a partial policy as an abstract (macro) action, which can be executed to make fast progress towards regions with good rewards.

Temporal abstraction is intuitively the decomposition of a task into a set of sequential subtasks which can be used to complete the task (see Figure 3.10). This form of decomposition is similar to the use of subroutines in a procedural programming language. Perhaps because of this similarity, temporal abstraction has emerged as the principal technique for exploiting hierarchy in reinforcement learning problems in recent years. Given a reinforcement learning problem, we learn a policy for a small set of low-level states (perhaps the set of states in some aggregation). This policy is then made available as an *abstract action* in addition to the concrete actions which define the problem. This abstract action may simply be used at the lowest level of abstraction to make fast progress towards a goal, or

may be used at a higher level as a mechanism for moving between abstract states.

H-DYNA

The H-DYNA system (Singh, 1992) is an early example of temporal abstraction in reinforcement learning. Based on ideas from hierarchical planning, Singh introduces the notion of a variable temporal resolution model (VTRM) for reinforcement learning. A VTRM uses different temporal resolutions in different parts of the state space to reduce the complexity of the problem being solved. Extending the DYNA architecture developed by Sutton (1990), the H-DYNA architecture models the reinforcement learning problem at various levels of temporal abstraction. At the lowest level, policies are learned for taking the optimal path between a pair of states. These policies are then available to the high levels as abstract actions to achieve sub-goals effectively. Like many hierarchical reinforcement learning algorithms, H-DYNA is more useful for problems with an explicit goal state than those without.

Hierarchies of Abstract Machines

Hierarchies of Abstract Machines (Parr and Russell, 1997) or *HAMs* are similar in structure to H-DYNA, consisting of a hierarchy of learning machines, with a low-level machine representing an abstract action available to a high-level machine. Each machine is defined by a *partial program*, which constrains the range of policies the machine can learn. In the case of HAMs, the partial program is a finite state machine augmented with non-deterministic *choice points*. Reinforcement learning is used to determine the optimal choice at each of these points, and learning can take place at many levels in the hierarchy simultaneously. The HAMs approach was later extended by the ALisp language (see Section 3.5.4) which provides a much more expressive language for partial programming.

Options

Sutton et al. (1999) use the notion of *options* to formalise temporally-extended actions. An option is defined by a tuple $\langle \pi, I, \beta \rangle$—a policy π defined over a subset S_o of the full state space S, a set of input states $I \subseteq S_o$ in which the option can be initiated, and a function $\beta : S_o \rightarrow [0, 1]$ which determines the probability the option will terminate in a given state. Options are made available for execution in addition to the primitive actions, and act as effective macro-actions for making progress towards a goal region, accelerating the reinforcement learning process. When an option is initiated for execution, the policy π is used to choose actions,

and after each action the option will probabilistically terminate according to the value of β for the resultant state. On termination, control is returned to the agent which initiated the option, so that a new option or primitive action can be selected for execution. A scheme of *hierarchical options* of the form $\langle \mu, I, \beta \rangle$ can be constructed by replacing the policy π for choosing primitive actions with a policy μ which can select other options for execution as well as primitive actions.

Semi-Markov Decision Processes

The *semi-Markov decision process* (SMDP) (Howard, 1971) is commonly used as a formal model of temporal abstraction in reinforcement learning. In this model, the number of time steps between one decision and the next is a random variable, and this is interpreted as the system remaining in the current state for a random waiting time, then making an instantaneous transition to the next state. The transition behaviour of the model is defined by a joint probability distribution $P(s', \tau | s, a)$, the probability that choosing action a in state s will result in a transition to state s' after τ time steps. It is straightforward to derive forms of the Bellman equations and the Q-update rule in this model, and to prove convergence results for some of the temporal abstraction algorithms described above. Barto and Mahadevan (2003) describe SMDPs in more detail, and show how the options, HAMs and MAXQ methods (MAXQ is discussed in Section 3.5.4) may be formalised using an SMDP model.

3.5.4 Combining Temporal Abstraction with State Abstraction

Temporal abstraction has proved to be a very effective way of decomposing large reinforcement learning problems into smaller, more tractable pieces. However, while temporal abstraction simplifies the learning problem, it does not address the state space explosion in the way that state aggregation techniques do. So while a subtask may be comparatively simple to learn, a huge state space will still render the problem intractable.

Using the full state space for learning a subtask will include many irrelevant state features. These features are not needed to make policy choices within the subtask, and increase the size of the state space exponentially. State abstraction is a technique for tackling exactly this sort of problem. For each subtask we need only consider the smallest number of state features necessary to construct an effective policy (see Figure 3.11). Hierarchical reinforcement learning offers the greatest gains when temporal abstraction is combined with state abstraction.

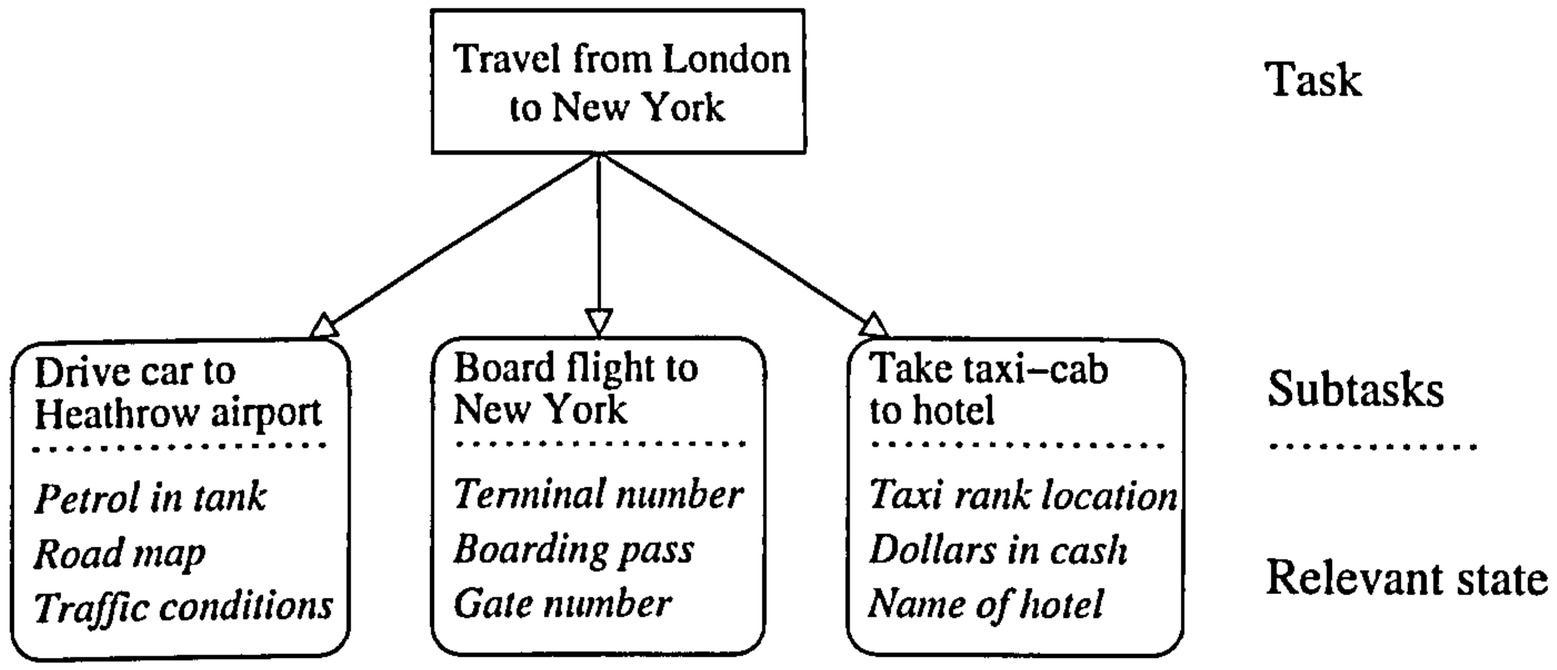


Figure 3.11: In this example, temporal abstraction is used to break the overall task into subtasks. State abstraction is then used to annotate each subtask with the relevant state variables for solving it.

MAXQ

The MAXQ decomposition method developed by Dietterich (2000b) was one of the first ways to combine these two forms of abstraction. MAXQ has many similarities to the options framework (Sutton et al., 1999), but places a greater emphasis on the use of hierarchy during the process of learning. Within each task in the hierarchy, the goal is to learn values for $Q(p, s, a)$, the expected return when task p is completed if we choose to execute subtask a in state s. The MAXQ value function decomposition partitions $Q(p, s, a)$ into two parts: $V(a, s)$, the expected return while executing subtask a, and $C(p, s, a)$, the expected return from completing p *after a has finished executing.* Partitioning the rewards in this way allows powerful state abstractions to be used which can greatly reduce the effective state space for the problem.

Combining the MAXQ decomposition with Q-learning for each policy subtask produces the MAXQ-Q algorithm. Using this algorithm, learning can take place at all levels of the hierarchy simultaneously, with the algorithm eventually converging to a *recursively optimal* solution. Dietterich derives the formal conditions for safely employing state abstraction in MAXQ-Q (in other words, using state abstraction without compromising the quality of the final policy).

ALisp

ALisp (Andre, 2003) is another recent method combining state abstraction with temporal abstraction. ALisp extends the *partial programming* approach of the HAMs (Parr and Russell, 1997) method, and is essentially the Lisp language augmented with non-deterministic choice points (where the reinforcement learning

takes place) and subroutine calls to lower levels of the hierarchy. The sum of rewards is partitioned in a similar way to the MAXQ method, but the algorithm converges to a *hierarchically optimal* solution, which is an improvement over MAXQ. State abstraction is achieved by the user annotating the choice points in an ALisp program with the names of the state features which are relevant to that learning choice. Andre (2003) also presents the formal conditions for safe state abstraction in the ALisp framework.

3.5.5 Learning Sub-Goal Hierarchies

All the hierarchical methods described so far assume that a hierarchy is supplied by the user before learning begins. There has been some research into determining useful task decompositions without prior knowledge. This obviously makes the learning problem much more difficult, and there has been very limited success in this area.

The SKILLS algorithm (Thrun and Schwartz, 1995) searches for abstract actions using a *description length* argument, which specifies the number of states each abstract action should cover. The algorithm starts with a single state for a skill, and grows the set of states defining the skill while learning a policy which maximizes reward in that region. While effective, the algorithm is slow. In the example grid navigation domain, the time required to find useful skills is an order of magnitude greater than the time needed to find a near-optimal policy.

In more recent research, McGovern and Barto (2001) use the concept of *diverse density* to identify states which occur *somewhere* in every successful episode, but not at all in failed episodes. Reaching such a state becomes a sub-goal, and an *option* (Sutton et al., 1999) is created for each of them. Şimşek and Barto (2004) present a similar approach based on the *relative novelty* of states. The novelty of a state decreases the more times it is visited. The relative novelty of some state in an experience trace is defined as the ratio of the novelties of the states immediately preceding and following the state. States with high relative novelty are candidates for sub-goals.

There have been several approaches which are based on properties of the *graph* formed by the underlying MDP's states and transitions. Şimşek et al. (2005) use a graph built from a recent experience trace, and search for a *cut* (a small set of edges to remove) which would divide it into densely connected subgraphs. States on either side of the cut are candidates for sub-goals. Mannor et al. (2004) pursue a bottom-up approach based on state aggregation, where small clusters of states are gradually combined together to minimize inter-cluster edges.

It is worth mentioning several algorithms for POMDP learning (see Section 2.6)

which are based on automatic construction of a hierarchy. *HQ-learning* (Wiering and Schmidhuber, 1997) is based on decomposition of a goal-oriented POMDP into a sequence of Markovian sub-tasks. Each sub-task is defined by an observation which must occur for the sub-task to end. A number of Q-learners are arranged in a fixed sequence. Each learner gradually determines which sub-task to achieve and how to achieve it by mapping observations to actions. When a sub-task is completed, control is always passed to the next Q-learner in the sequence. Sun and Simmons (1999) extended this approach so that each Q-learner can be activated more than once in the sequence. Since the hierarchy in these two approaches is essentially flat they are unlikely to scale well to larger problems.

McCallum (1996) introduces a POMDP learning technique called *Utile Distinction Memory (UDM)* which constructs a hierarchical state abstraction during learning. It based on a statistical test which estimates whether distinguishing two states (based on a finite observation history) will allow an improved policy to be represented. The technique is similar to decision tree function approximation (see Section 3.4.5) but in addition to estimating whether state/observation variables are relevant for optimal decision making, the technique estimates when the *recent history* of observations is also relevant.

3.6 Symbolic Representations for RL

A fundamental feature of standard reinforcement learning algorithms, and a key factor contributing to the state space explosion, is the *extensional* representation of states. In an extensional representation each state $s_i \in S$ is *explicitly named*, and important data structures such as the value function are based on this explicit naming scheme. Algorithms based on this extensional representation are very efficient for small state spaces, but have the disadvantage of an exponential growth in learning time as the number of state variables is increased.

In traditional AI disciplines such as classical planning, an *intensional* representation is much more common. In an intensional representation states (and more importantly, *sets of states*) are represented by a set of *state features*. For instance, if the state S is the Cartesian product of state variables $X_1 \ldots X_n$, we can use the feature $X_1 = 3$ to describe the set of all states which have value 3 for state variable X_1, whatever values the other $n-1$ variables take. If each state variable can have i possible values, $X_1 = 3$ represents a set of i^{n-1} states—hence the intensional representation of some sets of states can be exponentially smaller in n than a representation which explicitly enumerates each of the states in the set.

An intensional or *symbolic* representation allows us to represent sets of states

in a more compact fashion. However, algorithms for symbolic reasoning are usually based on searching for solutions in a search space of size exponential in the number of state variables (Bylander, 1994), so a trade-off emerges between representational complexity and time complexity.

It is worth noting that some of the methods discussed in Sections 3.4 and 3.5 derive part of their usefulness from an implicit symbolic representation of state. For instance, each node of the decision tree function approximator (Pyeatt and Howe, 1998) uses a symbolic state feature as a decision criterion. Also, the state abstraction offered by the ALisp language (Andre, 2003) is based on identifying the relevant symbolic state features for each procedure in the partial program.

This section surveys recent approaches which use symbolic representations to make reinforcement learning feasible in domains where there are a large number of state variables to consider. Representation techniques from other AI disciplines such as *classical planning*, *Bayesian networks*, *probabilistic planning*, and *logic programming* are among those evaluated for their suitability for use in reinforcement learning.

3.6.1 Classical Planning and Reinforcement Learning

A classical AI planning problem uses a restricted first order representation of state to provide a basis for reasoning efficiently about sequences of actions to achieve a given goal. In this section, we will consider planners which use the STRIPS representation (Fikes and Nilsson, 1971), although much of the discussion applies equally well to more complex representations such as ADL (Pednault, 1989), probabilistic STRIPS (Kushmerick et al., 1995) and the situation calculus (McCarthy, 1963).

Planning With STRIPS

In STRIPS an individual state is represented by a set of *positive ground literals*, and the *set* of goal states by a *conjunction of positive literals*. Each STRIPS *operator* (or action) is defined by the *changes* it makes to the set of positive literals which constitute an applicable state. This is usually compactly encoded as three elements, a set of *preconditions*, an *add-list* and a *delete-list*. A solution to a STRIPS planning problem consists of a sequence of operators which transform the *initial state* to one of the set of goal states.

A planning problem and a reinforcement learning problem share some key structural components. In both paradigms we have a current *state*, which can be transformed into a new state by means of an *action* (or operator). Both are

also concerned with identifying *useful sequences of actions.* This suggests a close relationship between the two problems, and that we may be able to use planning techniques to improve the performance of reinforcement learning algorithms. However, there are also key differences between the two problems. The effects of planning operators are *deterministic*, and the operator effects are given as *prior knowledge* to the planner. Also, in the planning problem we are primarily interested in reaching one of a set of *goal states.* In contrast, the goal in reinforcement learning is to maximize some optimality criterion such as the total discounted reward. It is not always possible to express this maximizing goal as a set of goal states.

STRIPS Planning vs. Dynamic Programming

The relationship between classical planning and *dynamic programming* is discussed in some detail by Boutilier et al. (1999). This relationship can be illustrated by transforming a STRIPS problem into an MDP problem for solution by a dynamic programming algorithm. This transformation can be defined as follows:

- The state space of the MDP enumerates all the sets of positive ground literals which represent valid situations in the problem domain.
- The actions of the MDP are the ground instances of each STRIPS operator.
- The reward function of the MDP is 1 in all the goal states of the planner (which are terminal), and 0 everywhere else.
- Taking an action in a state which does not satisfy the preconditions of the corresponding STRIPS operator results in a self-transition.
- Taking an action in a state which satisfies the preconditions results in a transition to a new state determined by the add and delete lists of the STRIPS operator.

The value function for this MDP can be obtained by dynamic programming. This determines for each state the action which leads to the nearest goal state. Therefore, from the value function it is possible to read out the shortest plan for *any* initial state. A planner's solution only applies to a *single* initial state. Despite this advantage, the state space explosion means that solving any significant planning problem in this way is generally intractable. But the close relationship between the two problems suggests that representations from AI planning are likely to be important in developing dynamic programming and reinforcement learning algorithms which can solve larger problems.

Richer Planning Representations

The reverse transformation, from an arbitrary MDP to a STRIPS planning problem is not possible, since neither actions with stochastic effects, nor reward functions which are not goal-oriented can be represented in the STRIPS framework. However, more expressive representation languages for AI planning have been developed since STRIPS, some of which can express one or more of these qualities. Probabilistic planning algorithms such as BURIDAN (Kushmerick et al., 1995) can represent the stochastic effects of actions, but the systems which have been built so far have exhibited very poor performance compared to deterministic planners. In terms of reasoning about rewards and the relative *quality* of several plans to achieve a goal, decision theoretic planning systems such as DRIPS (Haddawy and Suwandi, 1994) may provide insight into which intensional representations could be useful for reinforcement learning.

Macro-Operators and Hierarchical Planning

Some techniques developed to solve large-scale planning problems turn out to have natural analogues in reinforcement learning. The strongest influences are evident in the *hierarchical* reinforcement learning methods previously discussed in Section 3.5. A *macro-operator* (Korf, 1987) is a useful sequence of planning operators which is considered as an atomic unit for the purposes of state space search. Macro-operators have influenced methods like the *options* framework (Sutton et al., 1999) which are based on temporal abstraction. A similar influence can be observed in the state abstraction hierarchies of methods such as *ALisp* (Andre, 2003), which have many conceptual similarities to hierarchical planning algorithms such as ABSTRIPS (Sacerdoti, 1974).

Methods Combining Planning And Reinforcement Learning

So far, there has been relatively little research using symbolic planning representations to augment the capabilities of reinforcement learning methods. One exception is the technique developed by Boutilier et al. (1997) to solve MDPs with reward functions expressible as an *additive* combination of sub-goals. The technique uses *partially-ordered plans* as the intermediate representation of solutions for each of the sub-goals. These plans are then used as the basis for creating an overall solution of the MDP by merging the sub-goal solutions and prioritizing the sub-goals with the largest associated rewards.

RACHEL (Ryan, 2002a,b) is a hybrid system which combines RL with techniques from *teleo-reactive planning* (Nilsson, 1994). At the centre of this approach is the

concept of a *reinforcement-learned teleo-operator* (RL-TOP). An RL-TOP uses symbolic preconditions and effects to define the intended outcome of a behaviour, while leaving the implementation of the behaviour to be learned by RL. This means that the operator can be used in a planning algorithm, but it also means that a reward function for learning the operator behaviour can be automatically generated from the operator's symbolic preconditions and effects.

The RACHEL system uses what is known as *semi-universal* planning. A *universal* plan contains a path to the goal for every possible symbolic state, but is typically too costly to calculate and store in memory. A *semi-universal* plan is typically much smaller, and is generated by storing in memory all the failed paths generated during a search for a valid plan. In addition, if during plan execution a state is encountered which the plan does not cover, the plan is extended with a path from this new state to the goal. The use of semi-universal plans in the RACHEL system reduces the cost of replanning when an operator fails. In addition, the teleo-reactive approach allows the system to exploit shorter plans which unexpectedly become possible due to exogenous events.

3.6.2 Factored Representation of MDPs

One intensional approach which has become popular for solving large dynamic programming problems is to describe an MDP with a *factored representation* (Boutilier et al., 1999). Each state variable which is part of the overall MDP state can be referenced by a symbolic name, and is termed a *factor* of the MDP. Based on these factors, compact representations can be defined for the effects of actions, the reward function, the value function, and other elements of the MDP. Some of these representations are described in this section. Although much of this work assumes that the parameters of the MDP are known (and usually that the factored representations of actions and reward function are also known) it is likely that such representations will be useful in the future for scaling-up reinforcement learning methods.

A factored representation of an *action* compactly encodes the following properties:

- Factors in the current state which *affect the result* of the action.
- Factors which *can change* in the next state if the action is applied.
- The *conditional probability* of each change, given the current state.

Dynamic Bayesian Networks

If we have a factored representation for every action, this represents a compact encoding of the transition function of the MDP. One way to encode this information is to use a *Dynamic Bayesian Network* (Dean and Kanazawa, 1989), as shown in Figure 3.12. The form of the network used here is sometimes referred to as a 2TBN or *two-stage temporal Bayesian network.* The example in Figure 3.12 uses only binary state variables, but the approach applies equally well to multi-valued state variables. The *conditional probability table* (CPT) for each node at time $t+1$ determines the probability that the state variable has a particular value at this time, given the values of relevant state variables at time t. Any state variables not relevant to the action are not represented. If there are only a few relevant state variables, this is much more compact than a flat representation of the transition probabilities.

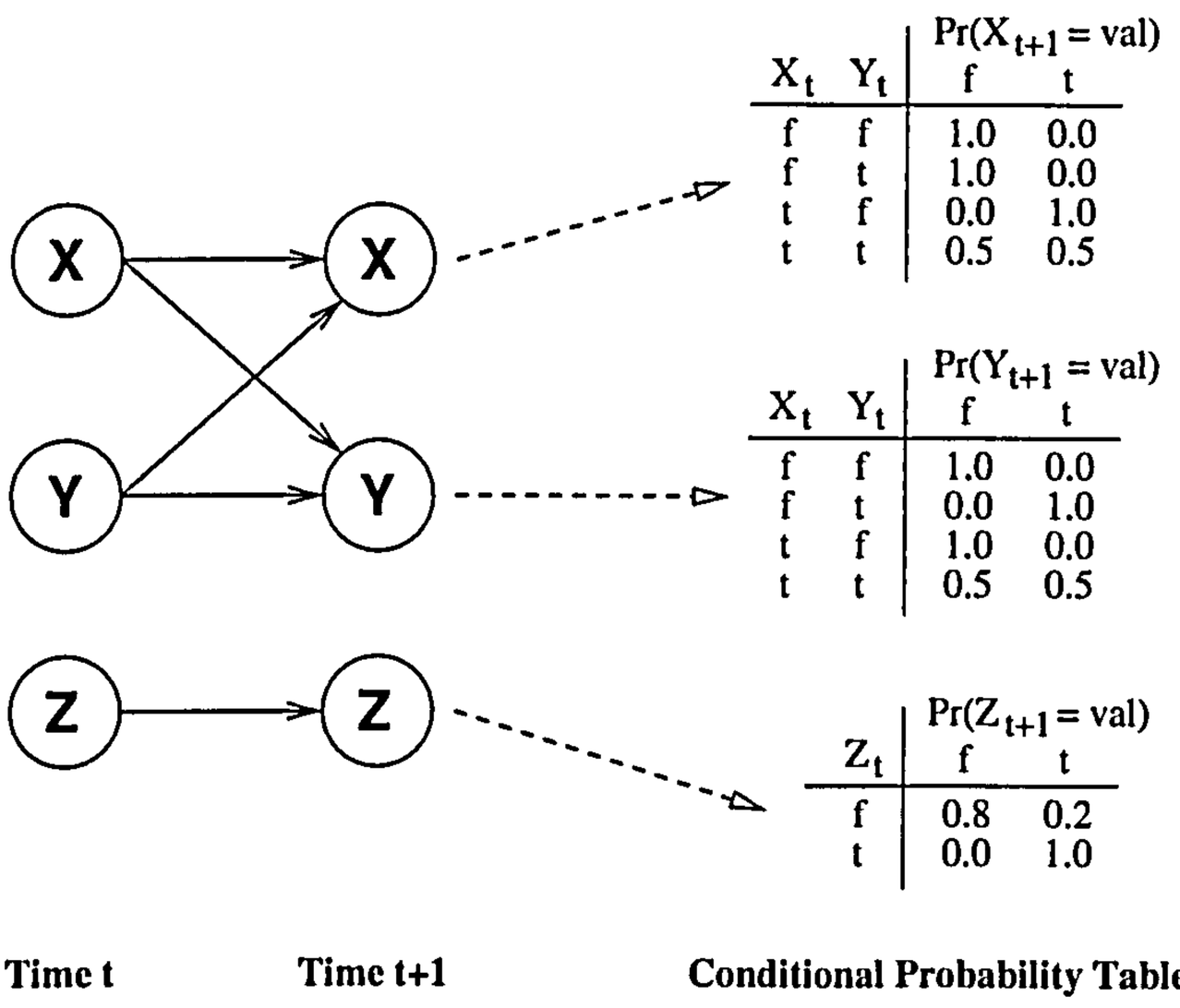


Figure 3.12: Factored action representation based on a dynamic Bayesian network.

Probabilistic STRIPS Operators

An alternative factored action representation is based on the concept of a *probabilistic STRIPS operator* (Kushmerick et al., 1995). Like a regular STRIPS operator (Fikes and Nilsson, 1971), the effects of the action are represented with an *add list* and a *delete list.* However, the probabilistic STRIPS operator has a number of such lists—the one to be used depends on the values of relevant variables in the current state. This is encoded as a decision tree, as shown in Figure 3.13. Each leaf node in the decision tree is a set of possible effects lists, each tagged with a

probability. This probability determines how likely it is that this list of effects will occur if the action is applied in a state corresponding to a particular path through the decision tree. This is good for representing compactly situations where the stochastic effects on several state variables are *correlated*. If, on the other hand, there are several *independent* stochastic effects on different state variables, the 2TBN representation is likely to be more compact.

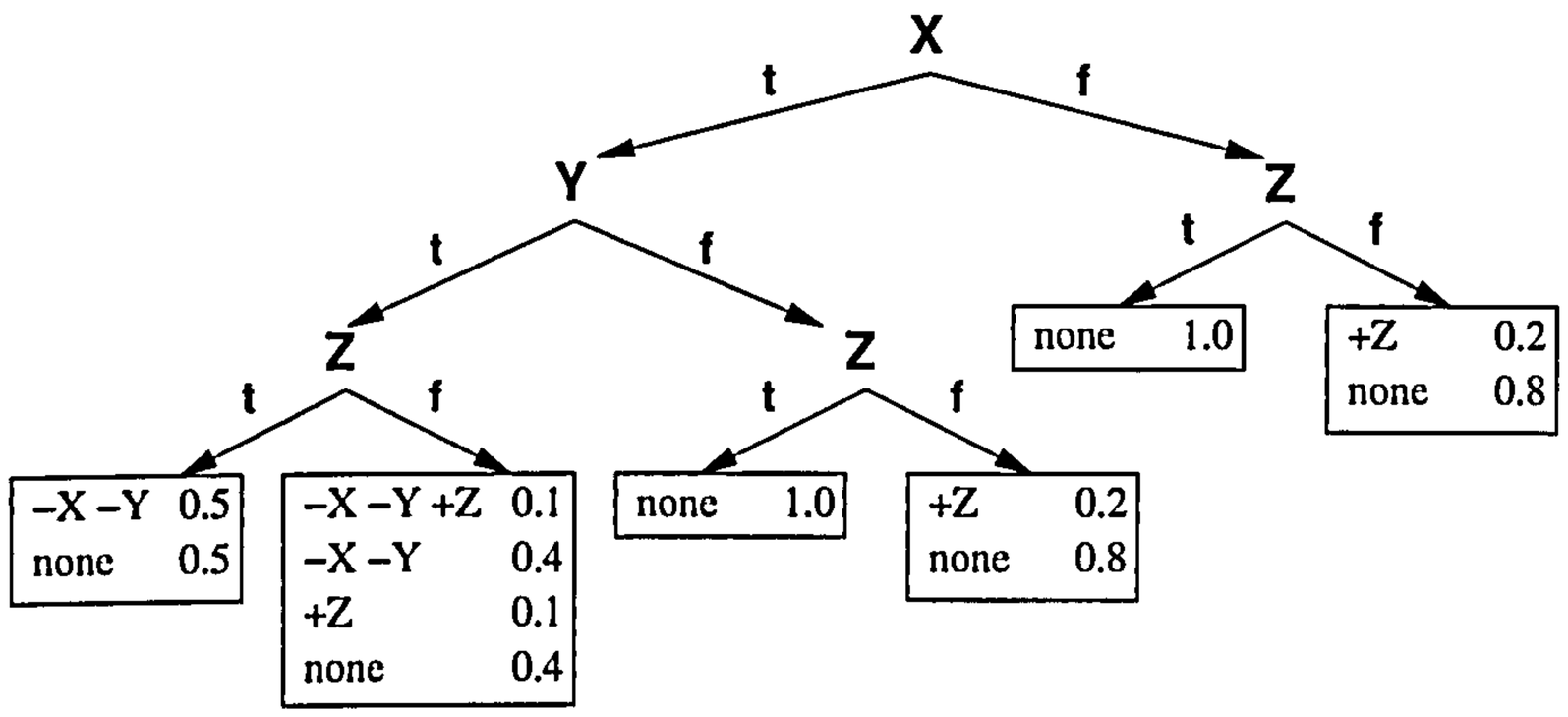


Figure 3.13: A probabilistic STRIPS operator representing the same action.

Factored Reward Functions

A factored representation can also be defined for the reward function. Like the action representations, there are several ways to do this, but one of the simplest is to use a decision tree data structure, and store at the leaf node the reward associated with states corresponding to that path through the decision tree (see Figure 3.14). If rewards are associated with state-action pairs rather than states, we will require one of these trees for each of the actions.

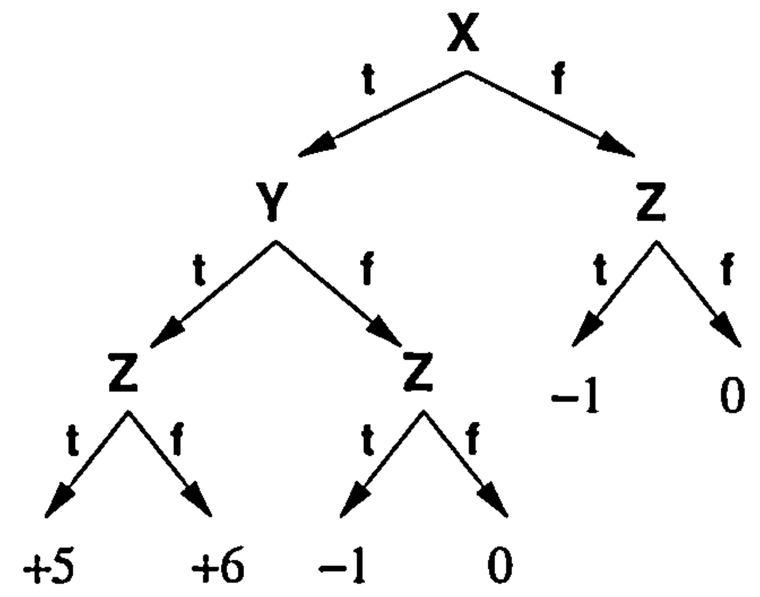


Figure 3.14: A factored representation of the reward function of an MDP.

Model Minimization

Now that we have represented the MDP compactly, the next challenge is to develop algorithms which can take these factored data structures and calculate similarly compact representations of the optimal policy and the value function. One approach by Dearden and Boutilier (1997) uses data from the factored representation to rank the state variables according to their degree of influence on the reward function. Given this ranking, a subset of the most relevant variables can be determined, and an abstract MDP with a smaller state space can be solved using this subset. This solution may then be used to "seed" a solution in the original MDP, reducing the time to convergence. A similar approach by Dean and Givan (1997) uses a factored representation to aggregate states between which it is not necessary to distinguish in order to act optimally. This technique is used to progressively build up a *minimal model*, the solution of which induces an optimal solution of the original MDP.

Factored Policies and Value Functions

Both the above algorithms only use the factored representation to reduce the state space that need be considered by a standard dynamic programming algorithm. The *Structured Policy Iteration* (SPI) algorithm developed by Boutilier et al. (2000) is one of the few algorithms which maintains a factored representation through to its eventual output. This may be significant if we need to solve a problem where the subset of potentially relevant state variables is still very large. The SPI algorithm employs factored representations based on decision trees for both value functions and policies. The value function for a specific policy is constructed starting from the tree for the reward function. A transformation of the tree based on the Bellman backup is repeatedly applied. This transformation uses the factored action representations to extend branches of the tree and update the state values at the leaves of the tree. Policies are successively improved by building the value function tree for one policy, then building a new policy tree with greedy choices in the value function.

The SPUDD (*Stochastic Planning using Decision Diagrams*) algorithm (Hoey et al., 1999) extends the decision tree representation of the SPI algorithm, using algebraic decision diagrams (ADDs) to represent conditional probability tables and value functions. SPUDD is a *value iteration* style algorithm which can be used to solve factored MDPs.

3.6.3 Relational Representations for RL

The factored approaches in the previous section use intensional representations based on state variables to solve structured MDPs efficiently. These representations are all inherently *propositional*—they can only express the possible values for each of the state variables. If the learning domain involves many *objects*, and *relations* between the objects are part of the state description, then a propositional representation is likely to be inefficient. This has led some researchers to consider using *first order* representations of state for reinforcement learning.

Symbolic Dynamic Programming

Symbolic Dynamic Programming (Boutilier et al., 2001) is a first order approach based on the *situation calculus* (McCarthy, 1963). It does assume the MDP parameters are known, but the representations involved may still prove useful for reinforcement learning. The preconditions and effects of each action in the MDP are represented in an extended version of the situation calculus, thus allowing *universal* ($\forall$) and *existential* ($\exists$) quantification to be used to express sets of states very concisely. To express stochastic effects, the calculus is extended with the *choice* operator, which represents a stochastic choice between two deterministic effects. When a stochastic action is applied, the system decides probabilistically which of the two deterministic effects to apply—so while the symbolic reasoning is restricted to deterministic situations, the addition of the choice operator allows the correct calculation of state values in stochastic domains. The reward function is expressed by partitioning the state space into sets of states using situation calculus expressions, and annotating each set with a reward value. A version of the Bellman update rule is then defined over the first order representation, and is repeatedly applied to obtain a value function compactly represented by situation calculus expressions (in a similar way to the reward function).

A disadvantage of using the situation calculus is that computationally expensive theorem proving techniques are needed to apply the inference rules. The relational Bellman operator, or REBEL (Kersting et al., 2004), is an alternative symbolic dynamic programming algorithm based on a STRIPS-like representation.

Relational Reinforcement Learning

Another technique for learning in first-order worlds is *Relational Reinforcement Learning* (Džeroski et al., 2001). In this approach, formulae from a STRIPS-like representation are the basis of a *decision tree* which compactly represents the value function for a *parameterized action* such as `move(A,B)` (where `A` and `B` are first or-

der variables). An example of a decision tree learned by relational reinforcement learning is shown in Figure 3.15. The decision tests in the nodes of the decision tree are based on relations, which can reference the variables of the parameterized action, e.g. `clear(A)`. Džeroski et al. (2001) evaluate several different algorithms for building the decision tree, based on existing learning techniques such as TILDE (Blockeel and De Raedt, 1998). The input to the decision tree learning algorithm is a set of situations encountered during a learning episode, each annotated with the action taken and the total rewards accumulated in the rest of the episode. While the representation is much more restrictive than the situation calculus, it still allows us to represent very compactly the effect of an action on a large number of objects. It can also take advantage of a fast decision tree learning algorithm, rather than having to use a theorem prover to calculate the value function. Relational reinforcement learning can be seen as the application of *inductive logic programming* techniques to the reinforcement learning problem.

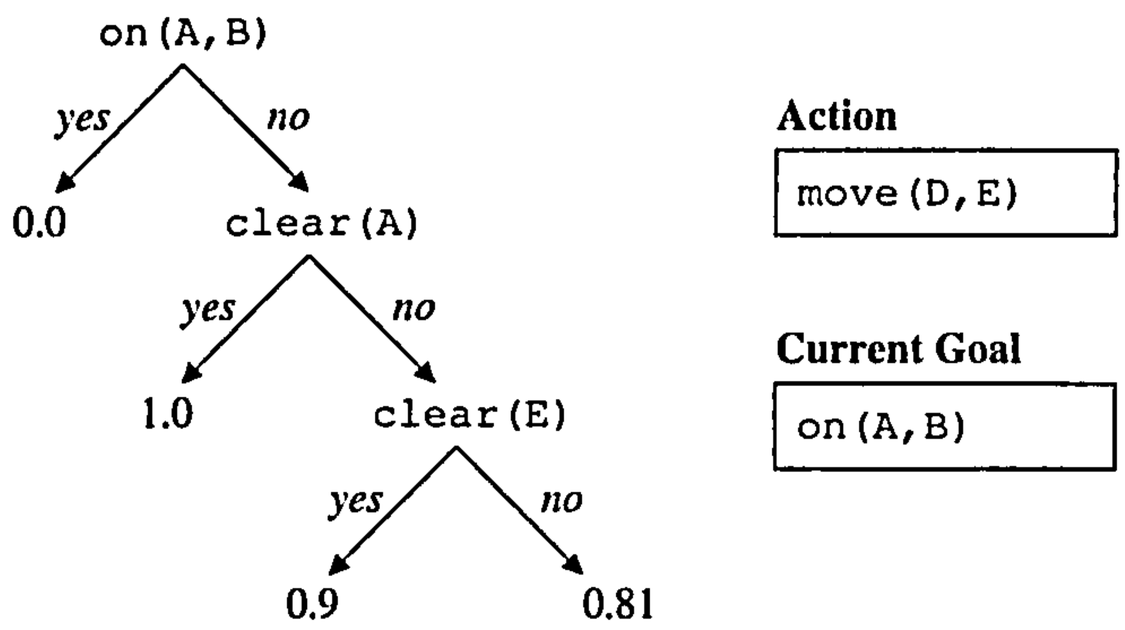


Figure 3.15: Example of a value function learned by the Relational Reinforcement Learning method (Džeroski et al., 2001).

Although relational reinforcement learning is a relatively new technique, it is becoming increasingly popular, and many researchers are seeking to improve and extend the approach. Alternatives to the decision tree approximator used by Džeroski et al. (2001) are being investigated, such as the method using *instance-based regression* developed by Driessens and Ramon (2003). Another successful approach encodes state-action pairs as graphs, then uses a kernel function (see Section 3.4.4) over the graphs as the basis of the approximation (Gärtner et al., 2003). Relational reinforcement learning has become one of machine learning's "hot topics" in recent years, resulting in a range of new approaches which combine ideas from classical planning, probabilistic logic learning, and the broader machine learning community. A comprehensive survey of the state of the art in relational RL and its relationship to research in other areas can be found in van Otterlo (2005).

3.7 Parallel Reinforcement Learning

In the preceding four sections, a wide range of techniques have been surveyed which can be used to apply RL to large-scale problems. While there has been substantial progress towards this goal, there remain many problems of *borderline feasibility* which can require many hours or even days to learn a high-quality policy. In these situations it is reasonable to ask whether a *parallel computing architecture* could be used to generate the policy more quickly. It is not immediately clear that this should be possible, since the characteristic interaction between an agent and its environment in RL is essentially a *sequential* process. However, where it is possible to *simulate* the target environment, the benefits of a parallel approach become evident. Parallel techniques for generating RL policies are surveyed in this section, and this broad approach will be referred to as *parallel reinforcement learning.*

The section begins with an overview of concepts from parallel computing which will be used throughout the thesis. Since limited space precludes an in-depth discussion of this topic, the reader is referred to Hwang and Xu (1998) for more details on the theory and practice of parallel computation.

3.7.1 Overview of Parallel Computing

There has been such a variety of parallel computing systems built over the years that it is important to try to classify parallel systems in a way which captures some of their key properties. *Flynn's taxonomy* (Flynn, 1972) can be used to classify a parallel computer according to the number of instruction streams and data streams:

SISD – Single Instruction Single Data A sequential computer with no parallelism.

SIMD – Single Instruction Multiple Data A machine with a single control unit which controls several subordinate processing units, each with its own data stream.

MISD – Multiple Instruction Single Data For example, pipelined architectures where each piece of data proceeds in turn through a sequence of processing units.

MIMD – Multiple Instruction Multiple Data Consists of a number of processors, each of which can run a different program and operate on its own data stream.

Flynn's taxonomy only captures the most basic dimensions of a parallel system, and in particular there are a wide variety of MIMD machines which require further differentiation. Bertsekas and Tsitsiklis (1989) list the following dimensions for classifying parallel systems:

Type and number of processors *Massively parallel* systems have thousands of processors. *Coarse-grained* parallel systems have more of the order of 10–20.

Presence/absence of a global control mechanism This roughly corresponds to the number of instruction streams in Flynn's taxonomy.

Synchronous vs. asynchronous operation Is there a global clock shared by all the processors which keeps them in lock-step?

Processor interconnections How do the processors exchange information? The two main alternatives are *shared-memory* and *message-passing* architectures.

From the wide variety of possible parallel systems, two kinds of MIMD systems are particularly popular.

A *symmetric multiprocessor* (SMP) computer uses a set of identical processors, where each processor has its own on-chip cache. These processors are connected to a *shared-memory*, either using a high speed bus or a crossbar switch. Maintaining *cache coherence* (consistency between the on-chip caches and the shared global memory) in such systems is a key architectural challenge. The processors are *symmetric* in that they have equal access to shared memory and any I/O devices attached to the computer. The SMP architecture is illustrated in Figure 3.16.

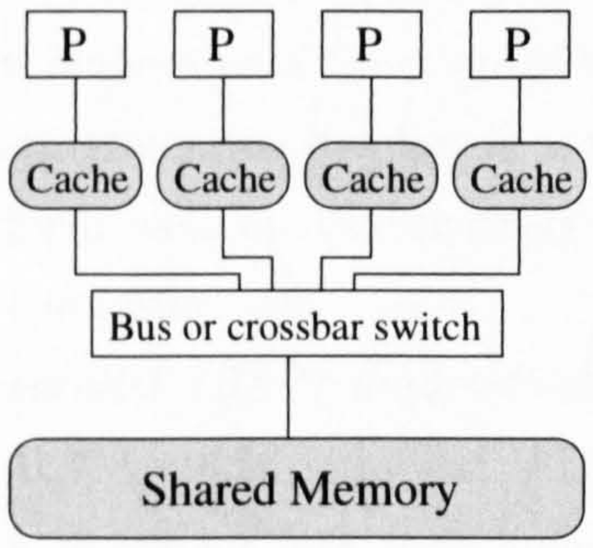


Figure 3.16: The architecture of a symmetric multiprocessor (SMP) computer. Each processor is marked with a P.

A *cluster of workstations* consists of a number of nodes, where each node is a computer in its own right. Each node has one or more processors, a local memory and usually also a local hard disk. The nodes are connected using either a low-cost switched Ethernet network or a high-speed interconnect designed specifically for

building clusters. A cluster of workstations is a *message-passing* or *distributed-memory* parallel system — messages are passed between the nodes over the network. The architecture of a cluster of workstations is illustrated in Figure 3.17.

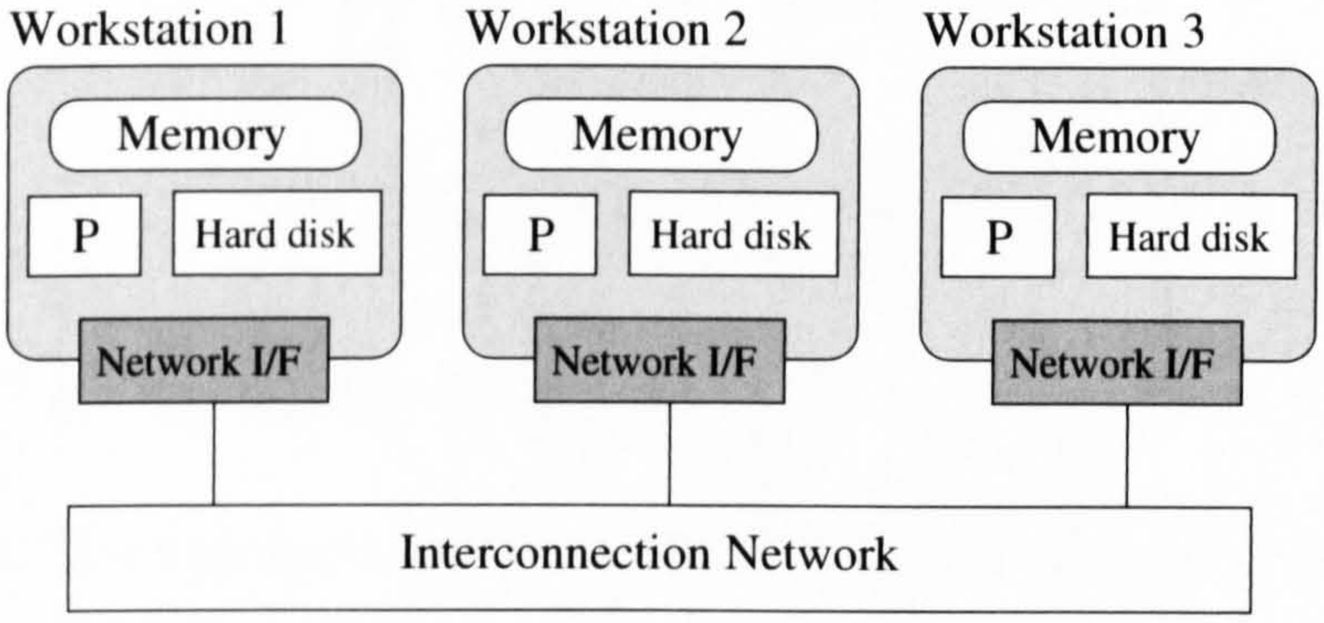


Figure 3.17: The architecture of a cluster of workstations. Each processor is marked with a P.

Abstract Models of Parallel Computers

In order to analyse parallel algorithms without needing to reference a particular parallel system, it is useful to define an *abstract model* of parallel computation to facilitate formal analysis.

The most popular such model is the *PRAM* model (Fortune and Wyllie, 1978), in which a *parallel random-access machine (PRAM)* consists of n processors which have access to a shared memory. Each processor can execute a single instruction at each time-step, or *cycle*. The processors are tightly synchronized, and communicate by reading and writing to shared variables in the memory. The complexity of a PRAM algorithm is usually defined as a function of the problem size N and the number of processors n. Communication overheads are not modelled, which means that algorithms which perform well on the abstract PRAM will not always be practical on a real parallel computer.

The *bulk synchronous parallel (BSP)* model (Valiant, 1990) addresses some of the problems that the PRAM model exhibits. A BSP computer consists of n nodes (each of which has a processor and local memory) that are linked using a communication network. A BSP computation proceeds in phases, as shown in Figure 3.18. At the start of a phase, each node performs a local computation that lasts at most w cycles. There follows a phase of communication between the nodes, where each node sends no more than h messages and receives no more than h messages. This phase takes no more than gh cycles. Finally there is a *barrier synchronization*, lasting at most l cycles, to ensure that all communications are finished. The entire phase, consisting of the three stages, is known as a *superstep*.

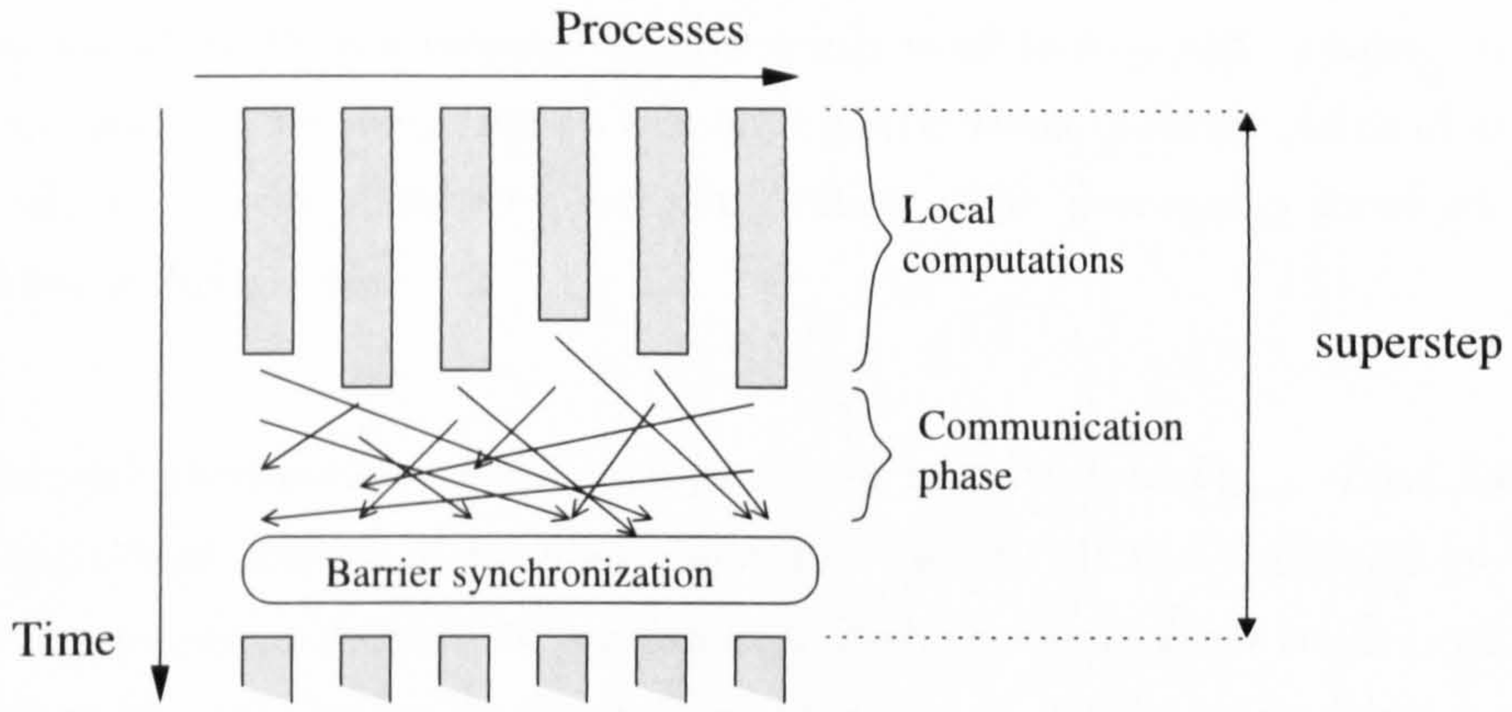


Figure 3.18: The bulk synchronous parallel (BSP) model of parallel computation.

The time for a superstep can be estimated as $w + gh + l$. The parameters g and l can be chosen to reflect the properties of a particular parallel system, which allows the performance of an algorithm on that system to be predicted. The BSP model can produce good predictions for both shared-memory and message-passing systems by accurately modelling communication overhead in either case.

Note that these models can only be used to analyse algorithms which are synchronous at either the cycle or superstep level. To analyse algorithms which proceed almost completely asynchronously, a more complex model for analysis must be used (Cole and Zajicek, 1989).

Properties of Parallel Algorithms

It is useful to define a number of quantitative properties which can be used to compare parallel algorithms for a given problem. Suppose that for some problem and algorithm the time to solve the problem sequentially is T_1, but the problem can be solved in time T_n using n parallel processors. These times could be in terms of cycles in an abstract model, or they could be actual timings in seconds of a real parallel system. The *parallel speedup* S_n achieved by the n processors is defined as:

$$S_n = \frac{T_1}{T_n}$$

It is generally the case that $0 < S_n \leq n$. If $S_n = n$ the algorithm is said to achieve *linear speedup*. It is more often the case that communication and other overheads prevent full linear speedup from being achieved. If $S_n < 1$, the parallel algorithm actually takes *more* time to solve the problem.

A related measure is the *parallel efficiency* E_n, which is a number between 0 and 1 which indicates how close we can get to a linear speedup using n nodes.

$$E_n = \frac{S_n}{n} = \frac{T_1}{nT_n}$$

The *workload* W is a measure of how much work is required to solve the whole problem, and can be measured in machine instructions, floating-point operations, or in terms of more abstract processing units. The *processing speed* P_n of the algorithm is defined as:

$$P_n = \frac{W}{T_n}$$

The *peak processing speed* of each processor is written as P_{peak}. *Load balancing* is an important aspect of parallel algorithm design. If the workload is divided equally between the processors, we can keep each processor close to its peak speed. In most cases, unbalanced loads and communication delays ensure that each processor spends some of its time *idle*. The utilization U_n is a measure of how well an algorithm minimizes the processor idle time.

$$U_n = \frac{P_n}{nP_{peak}}$$

Approaches to Parallel Programming

There are three primary approaches to parallel programming. These are *shared-memory* programming, *message-passing* programming, and *data-parallel* programming.

Shared-memory programming allows processes to communicate by reading and writing *shared variables*. Typically a global address space shared by the processes is divided into regions of private local memory for each process, and regions of shared memory accessible by multiple processes. A problem in shared memory programming is that variables can become inconsistent when two processes attempt simultaneous access. To prevent this, a *mutual exclusion* mechanism (often supported by hardware) is required to protect some sequences of instructions. Such a sequence is called a *critical section*, and these sections are the source of communication overheads in shared memory programs. While modern operating systems provide support for shared-memory programming, there is currently no widely accepted platform-independent standard. The POSIX Threads (Pthreads) library is a popular choice for shared-memory programming on Unix-like systems. *OpenMP* is the most promising platform-independent standard to emerge.

While shared-memory programming maps well to SMP systems, programming using *message-passing* is usually more appropriate for cluster computing. At its most basic, message-passing involves one process calling a `Send` function to transfer a sequence of bytes to another process calling a `Receive` function. These functions are usually provided in a library that can be used in an existing language such as C++. In *synchronous* message passing, the `Send` function will *block* until the remote process calls the corresponding `Receive` function. In *asynchronous*

message passing, *non-blocking* Send and Receive functions are used, with incoming messages placed on a queue until they are consumed by a Receive function call. Messages may be copied to a *buffer* in system memory, or be transmitted directly from an area of user memory. While message-passing programs can be implemented using platform-dependent libraries such as Windows *sockets*, there are two very popular platform-independent standards: the *Message-Passing Interface (MPI)* and *Parallel Virtual Machine (PVM)* standards. An implementation of the MPI standard is used as the basis for implementing the parallel algorithms in this thesis.

Data-parallel programming, which has little relevance to this thesis, is typically supported with additional language features rather than a library, and can concisely express procedures such as arithmetic operations on large vectors of numbers.

3.7.2 Parallel Dynamic Programming

Parallel approaches for planning in MDPs are typically based on *partitioning* the state space[5]. If there are n processors available, the MDP's state space S is divided into disjoint subsets $S_1, S_2, ..., S_n$. It is preferable that these subsets are all the same size, to balance the load on the processors. Processor i uses a value iteration update to calculate new estimates for all the states in S_i. The new updates are then broadcast to all the other processors. Each processor maintains a buffer to store the most recently received estimates received from other processor, which are used in the value iteration updates. In the simple synchronous version of this algorithm, each processor must wait to receive estimates from *all* the others before the next set of updates can be calculated.

Variations can be made to this approach to improve performance. Archibald et al. (1993) present a *pipelined* version of this algorithm. Each partition S_i is further partitioned into f sets $S_{i,1}, S_{i,2}, ..., S_{i,f}$. The updating and communication phases can now be overlapped, with processor i broadcasting updates to $S_{i,j}$ while concurrently updating $S_{i,j+1}$.

An asynchronous version of the basic algorithm is presented along with a convergence proof in Bertsekas and Tsitsiklis (1989). The asynchronous algorithm no longer requires updates and broadcasts to occur at the same rate. Multiple updates of each state in partition S_i can occur between broadcasts, and the most recent estimates from another processor can be used without needing to wait for estimates from any other processor. This asynchronous algorithm is shown to converge for infinite-horizon discounted MDPs under the assumption that there exists $T > 0$

[5]It is also possible to partition the action space, but most problems of interest have such a small number of actions that this kind of parallelization is not effective.

such that for all processors i and all time intervals of length T, processor i performs at least one update of every state in S_i and performs at least one broadcast.

There are many MDP problems where, for all states s, there are non-zero transition probabilities for only a small set of neighbour states $N(s)$, where $|N(s)| \ll |S|$. These are problems with a strong element of *locality* to the state. Solving these problems with the above algorithms can use significantly fewer communications if estimates of state s are only sent to processor i if there exists some $s' \in S_i$ where $s \in N(s')$. The memory requirements for each processor i are also reduced, since each only needs to buffer estimates for the set of states $\{s' | s \in S_i, s' \in N(s), s' \notin S_i\}$. The use of neighbour states in this way was proposed by Bertsekas (1982). Of course, the number of neighbouring states for each partition depends greatly on how the partitions $\{S_i\}$ are constructed. If each partition consists of states which inhabit a local region of state space, the number of neighbouring states can be kept small. An algorithm which constructs and exploits partitions of this kind is presented by Wingate and Seppi (2004).

3.7.3 Parallelizing Reinforcement Learning

In comparison to the above work in dynamic programming, parallelization of RL techniques has received little attention. This is surprising given that much existing RL research is carried out using *simulated* environments. A simulation can easily be replicated for each processor in a parallel computer. Therefore it should be possible for each processor to run a learning algorithm, and for intermediate results to be exchanged between them. More often, the attention of researchers has been directed towards *multi-agent* reinforcement learning, where several agents learn *different* but *related* tasks in either a *cooperative* (Kapetanakis and Kudenko, 2005) or *competitive* (Littman, 2001) setting.

The parallel RL and multi-agent RL settings have somewhat different assumptions and goals. Parallel RL is primarily concerned with finding policies for (simulated) *single-agent learning problems* more quickly by exploiting parallel hardware. Multi-agent RL involves agents which are situated in the *same* environment (producing a *multi-agent learning problem*) where *interactions* between the agents complicate the achievement of an effective cooperative/competitive policy. There is some degree of crossover between the two areas though, with multi-agent techniques such as advice exchange (Nunes and Oliveira, 2003) having some relevance for parallel RL.

Whitehead (1991) investigated RL in a restricted class of MDPs (deterministic, goal-oriented k-dimensional grids). One of the methods in this paper, *Learning By Watching* has several agents learning in identical environments, which are able

to "watch" the other agents, and use these watched experiences to update their value functions. The expected time (in environment time steps) for a population of n agents is shown to be $\Omega(1/n)$. Although a parallel implementation is not the focus of this work, it does describe a technique appropriate for parallel RL: *experience broadcast*. While in this paper watching other agents is essentially free, in most situations communicating experience tuples will have a cost. Experience broadcast is most useful for parallel RL when the cost of generating experience is large compared to the cost of updating a value function.

Tan (1993) also investigates experience broadcast in the context of multi-agent RL. The performance of experience broadcast in a *predator-prey* domain is compared with two other approaches. *Policy-averaging* involves a set of agents combining their value functions by setting each value to the mean of all the agents' estimates. *Same-policy updating* on the other hand requires that all the agents share a single value-function data structure, such that the effects of an update made by one agent are immediately visible to all the other agents. It should be noted that these results are not based on a true parallel implementation. The costs of communication are ignored in the results, although Tan does try to broadly characterize the network bandwidth consumed by each of the methods. Ahmadabadi and Asadpour (2002) extend the policy-averaging approach of Tan (1993) by introducing a number of measures for the *expertness* of an agent at a given task. A weighted average of policies can then be defined, favouring the most expert agents.

Parallelism is considered more explicitly by Kretchmar (2002) for the purpose of solving *bandit problems* (Berry and Fristedt, 1985). These are essentially single-state MDPs, where the goal is to explore the available actions in order to converge quickly to the one with the optimal return. In Kretchmar's approach each parallel agent stores an estimate of the return for each action, as well as a record of how many times each action has been tried. After an action is taken by each of the agents, they combine their estimates with a weighted average. Each agent's action value estimate is weighted by the number of times the agent has tried the action. Excellent parallel speedups are shown in the empirical evaluation of this method, which is based on a *simulation* of parallel agents. There is no cost assigned to inter-agent communications.

A parallel version of the TD(λ) algorithm was proposed in Maillard et al. (2005), which was implemented using the MPI message passing standard and evaluated using a cluster of workstations. The state space of each agent is represented using a neural network, and the changes in each agent's weights are periodically added together by exchanging weights over the network. This work is currently at a very early stage, but is conceptually similar to some of the new methods developed

in this thesis.

Note that *policy search* methods for RL (see Section 2.4) are in many ways more naturally parallelizable than value-function based approaches. It may be for this reason that there have been no specific studies of parallelism in policy search—parallelism is simply used in some researchers' implementations. There are three main ways to use parallelization. The *policy search space* can be divided among the agents for exhaustive search, or searched simultaneously using a genetic algorithm. The calculation of a *policy gradient* can be parallelized for gradient-based methods. Finally, the *policy evaluation* required to compare candidate policies can be parallelized by agents simulating episodes in parallel.

3.8 Conclusions

In this section, some broad conclusions are drawn from the areas of research surveyed in this chapter.

Exploration Strategy

The choice of a good *exploration strategy* remains an important part of applying reinforcement learning to any given problem. Without a good strategy reinforcement learning will almost certainly perform poorly. The more complex strategies can reduce the required number of explorative steps in the environment by an order of magnitude (at the expense of computation time and space). However, improvements in the strategy do not address the key problem of the *state space explosion*, and exploration time is likely to remain closely linked to the size of the state space.

Value Function Approximation

Function approximation enables reinforcement learning to be applied in domains with a large set of states, or even in domains which have a continuous state space. It also allows the reinforcement learner to *generalise* and make good decisions in states that have not yet been encountered during learning. There have been great successes in applying these methods to some domains, but it is often hard to reproduce this success in similar domains.

The relatively strong theoretical basis which has been established for reinforcement learning using *linear* and *memory-based* approximators should stimulate more research which applies these methods to the most difficult problems. If a similar theoretical basis cannot be established for neural networks, it may still be possible to determine empirical guidelines for using this approximator for RL. Ultimately

though, the successful application of function approximation in RL relies most strongly on choosing a good set of input features for learning.

Hierarchical Reinforcement Learning

Using *hierarchical reinforcement learning* methods to constrain the learning effort required for a given problem has proved useful for learning in large domains where function approximation is ineffective. Methods which make use of both *temporal abstraction* and *state abstraction* are the most successful. Hierarchical approaches are limited by the quality of the hierarchy supplied to the algorithm. A great deal of prior knowledge about the problem structure can be encoded in the hierarchy, resulting in fast learning and policies with good performance. If there is little prior knowledge about the problem, constructing a hierarchy can be difficult. A poor hierarchy can result in many of the best policies being excluded from being learned.

While there has been some research on automatic construction of hierarchies, building a hierarchy from scratch is likely to remain a slow process for the largest problems. But being restricted to a fixed hierarchy means that the best result that can be obtained may only be optimal with respect to the chosen hierarchy. This may be some way short of the true optimal. This suggests that future improvements to hierarchical methods may be based on *extensions* or *transformations* of an existing hierarchy supplied as prior knowledge to an algorithm.

Symbolic Representations

Reinforcement learning methods which exploit *symbolic representations* are an alternative approach to learning in domains that exhibit significant internal structure. Algorithms based on *factored* representations of MDPs allow prior knowledge about the internal structure of an MDP to be compactly encoded. *First-order* representations can be employed for domains with many objects and/or many inter-object relations.

Relational reinforcement learning is a technique which has been developed quite recently, and is likely to be widely applied in the future to suitable domains. The successful combination of *inductive logic programming* techniques with reinforcement learning in this approach suggests that other symbolic learning methods such as *explanation-based learning* could be used for a similar purpose (see Dietterich and Flann, 1997). There is also the potential in this area for *symbolic reasoning* to be combined with relational RL to create powerful hybrid planning/learning systems.

Parallel Reinforcement Learning

In comparison to the other techniques for scaling-up, parallel RL methods have not been widely investigated. Mechanisms such as *experience broadcast* and *policy averaging* have been proposed and evaluated with simulated agents. Very few studies have examined implementations on real parallel hardware, where message passing and shared-memory access both have costs which must be taken into account to achieve good performance. Almost all of the existing research assumes that a table-based value function is in use. Parallel methods which allow function approximation to be used would broaden the applicability of parallel RL. Measures of an agent's *experience* (or *expertness*) appear to be valuable for combining information from several agents. Policy search RL methods are often more naturally parallelizable than value-function based approaches.

Discussion

Developing methods to solve large RL problems remains a vibrant area of research. A huge variety of techniques have been proposed, as this survey has shown. However, while there has been substantial progress, many learning problems remain infeasible for the current generation of algorithms. In addition, many of the techniques only perform well in special cases of learning problems.

The primary focus of this thesis is the area of *parallel reinforcement learning*, which has received little attention compared to some of the other techniques. One goal of this thesis is to examine the possibility of parallel RL in the presence of *value function approximation*, since most previous work on parallelism has assumed a table-based value function. Another goal of the thesis is to develop methods which are practical for implementation on real parallel hardware, as many existing methods do not take into account the cost of inter-agent communication. This material is covered in Chapters 4, 5 and 6.

A secondary topic in this thesis is the use of *symbolic planning* to provide hierarchical structure for RL policies. This is a hybrid approach which draws on techniques from both the *symbolic representations* and *hierarchical RL* sections of the above survey. The purpose of this part of the thesis is to show that symbolic planning is an effective way to automatically construct hierarchies for learning, which is an important goal in hierarchical RL. This material is covered in Chapter 7.

Chapter 4

Merging Approximate Value Functions

In this chapter I present a family of methods for parallel reinforcement learning (see Section 3.7), based on the notion of a set of agents which learn in parallel and *periodically merge* their *value function approximations*. After motivating this approach and describing the core method, several instantiations of the method which use different *merging functions* are defined. These instantiations are evaluated and compared using a *simulation* of parallel agents. I examine the effect on parallel speedup of both the number of agents and the period between merges. A comparison is also made with a baseline method where agents learn in parallel, but in isolation (i.e. with no merging). Finally, the method is implemented on a cluster of workstations to assess its practicality for speeding up learning in a realistic parallel setting.

4.1 Motivation and Assumptions

In Chapter 3 a wide variety of techniques were surveyed which can be used to extend the feasibility of RL to larger and more complex learning environments. While a great deal of progress has been made in this area, there remain many RL problems of interest that are only borderline feasible, requiring hours or days of learning time to converge to the optimal policy. In such borderline cases, it is reasonable to ask whether *parallel hardware* could be used to reduce the time required to find the optimal policy. The hardware available may take a variety of forms: a computer with a *multi-core* processor, a *multi-processor* server, a *cluster* of workstations or even a *grid computing* infrastructure.

It is not obvious that parallel methods can be applied to RL problems. RL agents are typically characterized as learning from a *sequential* stream of *experience*

tuples generated (using some exploration strategy) by *direct interaction* with some environment. If we assumed that this was the only source of experience available, it would only be possible to parallelize the computation expended on each experience tuple. This could be used to update a large set of neural network weights more quickly, or to perform extra planning in the manner of DYNA (Sutton, 1990). If generating experience is relatively cheap, this approach is unlikely to improve greatly over a simple sequential Q-learning agent.

However, suppose it were possible to situate two or more agents in *separate* but *identical* environments. If each of the environments behaves as if it has the same underlying MDP then experience in any of the environments is interchangeable. The environments are in effect parallel sources of experience. If in addition there is a way the agents can communicate with each other, there is the potential for them to *share experience* in some manner. Identical environments of this kind are possible if the environments are *simulated* on a computer.

The parallel architecture we assume is available is shown in Figure 4.1. There are a number of parallel processes, each of which supports an agent learning in parallel using an identical simulation of a *single-agent* problem. During learning each agent updates its own private value function. There is a channel for inter-process communication, which may be based on either shared-memory or an interconnection network. Using this channel, information can be exchanged between agents, allowing one agent to exploit information learned by another. This can accelerate convergence to an optimal policy. The exact form of the information being exchanged will be discussed later in this chapter. Note that it is important that agents have *different* experiences in the simulation, which is simple to achieve if each agent's exploration strategy selects a small proportion of random actions.

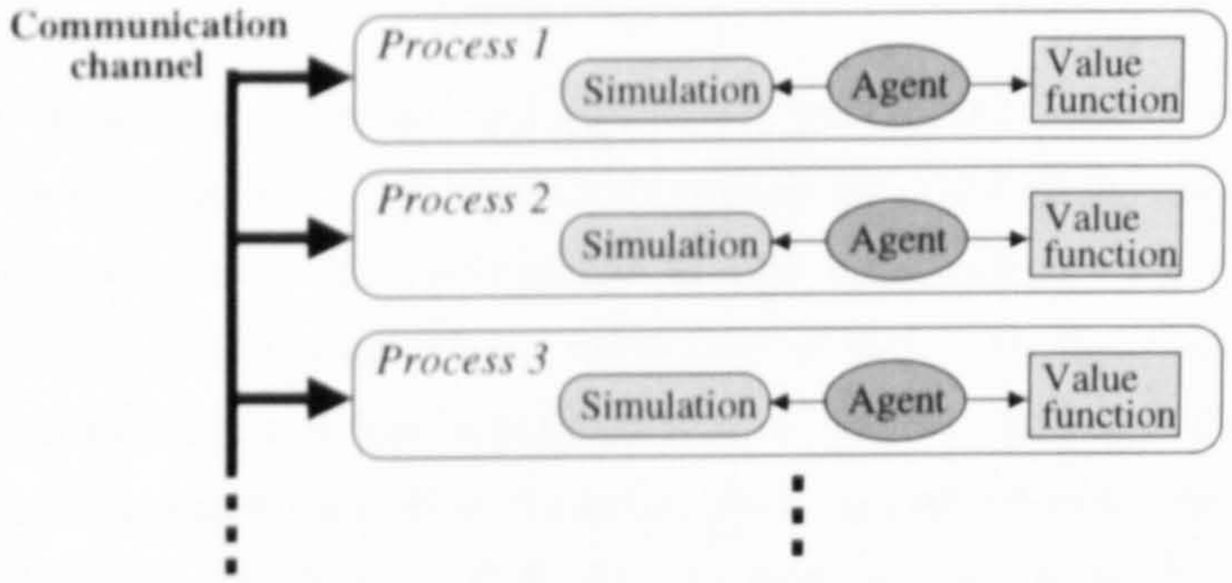


Figure 4.1: The parallel architecture allows several agents to learn in parallel using identical simulations of a single-agent environment.

Adopting this architecture implies that our method can only be used for *simulated* RL environments. One could argue that this excludes the interesting case of

embodied RL agents—autonomous robots which interact directly with some real-world environment. In practice, the cost in time and difficulty of collecting a large volume of robot interaction data usually makes some degree of environmental simulation inevitable. It is therefore reasonable to adopt the assumption of a simulated environment.

Note that the value function learned by each agent predicts values over the *entire* state space of the simulation. This is quite different from the traditional approach in *parallel dynamic programming* (see Section 3.7.2) where the state space is *partitioned* into a number of regions, with each agent only updating the values of states in its assigned region. There are a number of reasons why a partitioned approach may not be appropriate for parallel RL, where the transition and reward functions are not known:

- A lot of effort in RL is focused on *sampling* the environment's transition and reward behaviour. Speeding up the sampling process by combining results from several agents is only possible if the agents learn in the same areas of state space.

- A key strength of RL is that value function updates are focused on states with a high visitation probability under the current policy of the learning agent. Restricting each agent to learning within a partition would change the overall distribution of updates.

- Since the transition function is initially unknown, a suitable partition (with a small number of inter-partition transitions) would either have to be derived from external problem knowledge, or would require modification during learning as the state space was explored.

Because the agents are not restricted to disjoint partitions, they do not each have an exclusive specialization. This means there may be some duplication of effort in the population of agents. The advantage of this approach is that *all* the agents can focus on those states with a high visitation probability, and that the effort of sampling reward and transition behaviour can be divided amongst the agents.

A final point to mention is that there are great benefits to adopting an *approximate representation* (see Section 3.4) for the value functions used by the agents. Consider the goal of using parallel RL to speed-up learning in problems of borderline feasibility. Restricting a parallel method to table-based value functions would allow the method to be applied only to the most simple problems. Many RL problems of interest will be infeasible without value function approximation, with or without parallel methods. Allowing a parallel method to be *combined* with

value function approximation gives the method a much wider applicability. In addition, the generalizing power of function approximation broadens the effect of each piece of information exchanged between the agents, and the compactness of an approximate representation may reduce the overall bandwidth required for the communication channel.

4.2 A Merging Method

Based on these motivating ideas, I will now present the general form taken by the methods studied in this chapter. In essence, the core idea is for parallel learning agents to periodically *merge* their (approximate) value functions to *accelerate convergence.*

Method Overview

Each individual agent in this method uses *standard* RL techniques to learn from experience in its local simulation instance. A precise description of the techniques used is given later in this section.

The novel aspect of the merging method is the way in which the agents *exchange information* to improve their collective performance. This information exchange takes place by *merging* value function approximations (VFAs). Intuitively, a merge takes several VFAs and combines their content to form a new VFA. The new VFA should ideally preserve information known by all the agents, but also incorporate recent changes to the value function discovered by individual agents.

The method alternates between a *learning phase* and a *merging phase.* In the learning phase, each of the learning agents operates in isolation, interacting with a local simulation instance and updating its private VFA. There is no communication in this phase. The learning phase lasts for p simulation time steps (i.e. each agent collects p experience tuples). This quantity p is known here as the *merge period* of the method.

The merging phase is illustrated in Figure 4.2. Separate from the learning agents, a single *manager agent* takes responsibility for the merge operation. The phase begins with each learning agent communicating the weights of its private VFA to the manager agent. The manager agent computes the merged VFA using these weights. It then broadcasts the weights of the merged VFA to the learning agents. Each learning agent updates the weights of its private VFA to those of the merged VFA. This means that there is no diversity in the agent population at the start of each learning phase. Diverse experiences are achieved only through the randomness of explorative actions.

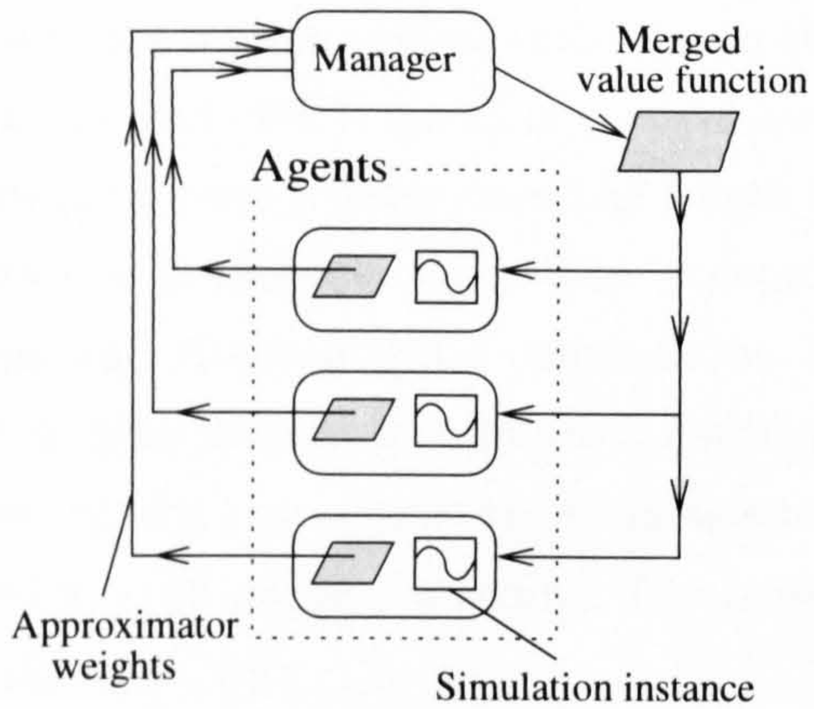


Figure 4.2: The *merging phase* of the core method. Each learning agent sends its approximator weights to the manager agent. The manager agent creates a *merged value function* from the weights and broadcasts the result back to the learning agents.

Method Detail

Each learning agent uses *linear function approximation* (see Section 3.4.3) to represent an approximate value function. The features used by the linear approximator are generated using *tile coding* (see Section 3.4.3). The SARSA(λ) algorithm with a *replacing eligibility trace* (see Section 2.4) is used to update the approximate value function as experience is acquired from the environment. An efficient implementation of the eligibility trace is used which maintains a size-limited list of *non-zero* trace values (see Sutton and Barto, 1998, section 7.9 for further details).

This combination of linear approximation and SARSA(λ) has been shown to be strongly convergent when used purely for estimation of value functions (Perkins and Precup, 2002). However, in this work a SARSA(λ) learner follows an exploration policy which in the limit tends towards greedy choices in the approximate value function. Under these conditions there does not currently exist a proof of convergence for SARSA(λ), although some limited progress has been made towards such a proof (Gordon, 2001; Singh et al., 2000). In spite of the lack of theoretical guarantees, the combination of SARSA(λ) and linear tile coding has been shown to work well in practice for a wide variety of domains (Sutton, 1996). These techniques are therefore currently the most reliable basis for a parallel RL method which uses an approximate value function representation.

Each agent chooses actions using an ϵ-*greedy* exploration strategy. It is important that $\epsilon > 0$ so that there is some diversity in the experiences of individual agents.

The division of the method into local computations in the learning phase and

global communications in the merging phase is similar to the notion of *supersteps* in the *bulk synchronous parallel (BSP)* model of parallel computation (see Section 3.7.1). The choice of merge period p determines the length of each learning phase. Since the merging phase is potentially quite time consuming, both in terms of communicating weights and the manager's computation time, p must be large enough for the agents to learn enough new information to warrant the expense of a merging phase. However, if p is too large then the agents will not communicate often enough to achieve a good parallel speedup. The influence of parameter p is investigated further in Section 4.8.

Underlying this approach is the key assumption that all the agents use the same set of learning features $\{\phi_i\}$. Since these features are generated using tile-coding, this implies that the number, resolution and offsets of the tilings are the same for each of the agents. This has some important advantages:

- No mechanism is required for projecting from one set of basis functions onto another.
- A weight θ_i has the same meaning for every agent.
- The only weights from other agents that are relevant for adjusting one agent's value for θ_i are the other agents' values for θ_i.
- A set of weights can be communicated using either the full vector of weights $\vec{\theta}$, or using a sparse set of index-value tuples $\{(i, \theta_i)\}$.

Under these assumptions, the i^{th} weight of the merged approximator, θ_i^m, will only depend on each learning agent's estimate for θ_i. This allows a *family* of merge methods to be defined, each based on a function f. The function f is used by the manager agent to calculate each weight of the merged VFA as follows:

$$\theta_i^m = f(\theta_{1,i}, \theta_{2,i}, ..., \theta_{n,i})$$

Here $\theta_{j,i}$ is the i^{th} weight of agent j, and there are n agents in total. The function f will be known here as a *merge function*. A number of different merge functions that can be used with this method are investigated in Section 4.4. The general form of the merge method (given a specific merge function f) is shown in Algorithms 1 and 2.

4.3 Evaluating Parallel Learners

At this point, in order to evaluate a number of possible choices for the merge function f, it is necessary to define the criteria by which the success of the merging

Algorithm 1 Pseudocode for learning agent.

while time elapsed $< t_{end}$ **do**

for $step = 1$ to p **do**

Execute a simulation step.

Update weights $\{\theta_i\}$

end for

Send weights $\{\theta_i\}$ to manager agent.

Receive merged weights $\{\theta_i^m\}$ from manager agent.

for all i **do**

$\theta_i \leftarrow \theta_i^m$

end for

end while

Algorithm 2 Pseudocode for manager agent.

while time elapsed $< t_{end}$ **do**

for $j = 1$ to n **do**

Receive weights $\{\theta_{j,i}\}$ from agent j.

end for

for all i **do**

$\theta_i^m \leftarrow f(\theta_{1,i}, \theta_{2,i}, ..., \theta_{n,i})$

end for

for $j = 1$ to n **do**

Send merged weights $\{\theta_i^m\}$ to agent j.

end for

wait until next merging phase.

end while

method will be judged. In this section I will describe the measurements which will be recorded as part of an empirical evaluation. In addition, I will give details of two implementations of the method which will be used to generate results. Finally, several learning problems will be defined which will be used to evaluate the implementations of the method.

The goal of the work described in this thesis is to *speed up* the learning of policies for single-agent RL problems using parallel hardware. However, there are a number of properties of RL which must be addressed in order to apply concepts such as *parallel speedup* and *parallel efficiency* (defined in Section 3.7.1). A key property of RL algorithms is that they *converge* to an optimal (or near-optimal) value function (or policy) rather than *calculating* the optimal value function. RL algorithms based on value functions gradually reduce the error in the estimate of the optimal value of each state. In many cases, the true optimum may never be reached—the best we can do is make the error arbitrarily small. To directly compare the time used by two RL algorithms, it is therefore necessary to specify some error bound ϵ for the final value function. It is also informative to compare the *rate of convergence* of two RL algorithms, since this will allow us to make predictions for a range of different values for ϵ.

The other key property of RL algorithms is that they are *probabilistic*. Since most algorithms take some number of random exploratory actions, and many environments have stochastic state transitions, it is not possible to guarantee that a particular error bound ϵ can be achieved in a fixed time t. However, as $t \rightarrow \infty$ it is possible to have a probability of achieving the error bound that is arbitrarily close to 1. Therefore, in order to directly compare the time taken by two RL algorithms, it is not only necessary to specify an error bound ϵ but also a probability δ of achieving the error bound. These ideas are similar to those used in the *Probably Approximately Correct (PAC)* model of learning (Valiant, 1984).

Error in the value function is by no means the only way to judge the success of an RL algorithm. A value function with large errors may be very effective for choosing actions if the ranking of actions is similar to the optimal value function. This idea leads to metrics which measure the *performance* or *quality* of a policy as learning proceeds. One such measure is the *average reward* collected over some interval, which characterizes how well a learning agent seeks out the highest rewards in the domain. This measure changes depending on the reward function used. To examine whether different reward functions allow desirable behaviours to be learned more quickly, an *external* measure of performance is required which does not depend on reward.

The evaluation domains used in this work (see Section 4.3.1) are all *episodic*,

and the desirable behaviour in each of these domains is to either *maximize* or *minimize* the number of simulation steps in each episode. This allows the use of an easily measured indicator of performance, the *mean number of steps per episode* over a given interval. This measure will allow us to vary both the RL algorithm and the reward function to achieve the best performance in a given domain.

The term *experiment* will be used here to denote a single run of the parallel method, with n agents starting with the same initial VFA weights, performing learning episodes in parallel and communicating every p time steps. Each experiment ends after a fixed time limit, with time defined using one of the three quantities to be defined shortly. Since the agents' experiences are different, the performance measure of mean steps per episode will vary across the group of agents to some degree. It could be argued that to measure the performance of a parallel RL algorithm, some *collective* measure of the overall performance achieved by the *group* is needed. However, since each agent learns a VFA to cover the whole state space, and information exchanged between the agents accelerates convergence for *all* members of the group equally, it is sufficient to nominate *a representative agent* from the group and record the performance of this agent over time. The fact that each merge results in the agents having identical VFAs means that over time it is highly unlikely that any agent's performance will differ greatly from that of the representative agent.

Because the outcome of an experiment is probabilistic, results must be collected over a number of different experiments, each starting with a different random seed. The expected performance of the algorithm is found by taking the mean of the performances achieved in each of the experiments, i.e. the mean over the set of experiments of the mean steps per episode achieved in a given time interval. The standard deviation and standard error of the mean over experiments will be examined to establish confidence intervals to compare the performance of two algorithms. The variance of the episode length *within* a single experiment will not be examined in detail.

To determine the speedup (or rate of convergence) achieved by an RL algorithm, it is necessary to measure the performance achieved after a given time. But how should the learning time be measured? There are three key quantities for measuring the elapsed learning time:

Simulation Episodes The number of episodes completed in the simulation by a *single* parallel agent (*not* the total over all the agents).

Simulation Steps The number of time steps completed in the simulation by a *single* parallel agent (*not* the total over all the agents).

Real Time The number of seconds elapsed since the parallel experiment was started.

For the most part the use of episodes as a measure of time will be be avoided. This is because the goal in our evaluation domains is to maximize or minimize episode length. This causes the length of episodes to vary significantly from the start to the end of an experiment, which tends to distort the reported rate of convergence if time is measured in episodes.

Simulation steps and *real time* both have their advantages. Measuring the simulation steps consumed by each parallel agent allows an assessment of *how efficiently* a parallel method *combines data* from *multiple experience sources.* The cost of communication is not included in this parallel measure, which can be viewed as an analogue of the *sample complexity* of a sequential RL algorithm.

The elapsed real time includes not only the time spent simulating the environment, but also the time required to update the VFA and the time used for communication. It is a measurement which captures more of the costs associated with the parallel method, and is good for assessing the performance achievable in practice. However, measuring real time has the disadvantage that any results will be specific to both the parallel system in use and the implementation details of the parallel method. The processor speed, memory size and speed, and the network latency and bandwidth are the system properties which will most significantly affect the result. In addition, if the parallel system is shared between a number of users, variations in the load of the system can adversely affect the results. In this work, the real time measurements were collected during periods of very light system load, with the parallel method consuming almost all of the available processing and network resources.

I will now describe the two implementations which were used to collect results.

A Simulation of Parallel Agents

The first implementation is a simple simulation of parallel agents, which requires no parallel hardware to be available, and does not even require operating system support for threads or inter-process communication. The simulation was written using C++, and runs within a single process. Suppose we use merge period p for a particular experiment. During a learning phase, the agents *in sequence* each execute p time steps in the environment. The VFA for each agent is stored separately in memory. After the learning phase is finished, the effects of a communication phase are calculated. Note that sending of individual messages between the agents is not modelled. Instead, each weight θ_i^m of the merged VFA is calculated using the

agents' VFAs which are stored in memory. Once the merged VFA is calculated, the weights are copied into each of the agents' VFAs, modelling the effect of a broadcast from the manager agent.

Results from the simulation are reported using *time steps* in the simulated environment to measure time. After each learning and communication phase, the elapsed time is reported as p time steps, even though the sequential computation time required for n agents is $O(np)$ for the learning phase and $O(nf)$ for the communication phase (f is the number of approximator features). The memory required is also $O(nf)$. Reporting the results in this way allows us to assess the *potential* parallel speedup achievable if the agents were learning in parallel, and communication was essentially free. Obviously we cannot achieve these conditions on a real parallel system. In spite of this, these results are very useful for comparing different parallel RL algorithms, since it is easy to examine properties such as how efficiently the agents can combine sampled experience, or whether the convergence properties of the underlying single-agent algorithm are affected. We can examine these properties without requiring access to parallel hardware or making assumptions about the processing power or network bandwidth of the target system.

An Implementation on a Cluster of Workstations

While the above simulation is useful for determining some properties of parallel RL methods, there are some questions that remain difficult to answer using only simulated results. For example, what is the *actual* parallel speedup that can be achieved when communication has a given cost? Which parallel RL methods are practical for implementation on an *symmetric multiprocessor* computer or on a *cluster of workstations*? How can we compare two parallel RL algorithms, one of which is more computationally intensive but requires fewer communications? A richer simulation than the one described above (one which explicitly models communication costs) may be used to answer some of these questions. However, simulating large numbers of agents solving difficult RL problems will make experiments very time consuming. It is also difficult to accurately model all the forms of communication overhead observed on a real parallel system.

For these reasons, it will be useful to run some experiments on real parallel hardware. Results using real hardware will also be useful to assess how *practical* a parallel RL algorithm is for implementation on a particular kind of hardware. The performance achieved will be measured against *real time*, which unfortunately will tie the results to particular properties of the hardware and implementation used here. In spite of this, the results will be useful for predicting performance on

similar hardware with different specifications.

This thesis focuses on a *distributed memory* model of parallel computation using *message-passing*. The main advantage of concentrating on this model is that an effective message-passing algorithm can easily be replicated on a *shared memory* computer (message passing can be emulated by copying messages into data structures in shared memory). In this sense message-passing algorithms have a greater generality, since they can be used to achieve good performance on both distributed memory and shared memory computers. Parallel RL algorithms written *specifically* for shared memory machines may achieve even better performance, but this was not a topic investigated as part of this research.

The hardware used for the experiments reported in this thesis was a Beowulf cluster of 20 Linux workstations. Each workstation had a 1Ghz Pentium III processor, 768MB of memory and a local hard disk drive. The nodes were connected using a 100Mbs Ethernet network[1] based on a single Hewlett Packard switch. The implementation was written using C++ and version 1.2.5.2 of the MPICH parallel programming library. MPICH is an implementation of the *Message Passing Interface (MPI)* standard (Pacheco, 1997). Initially we implemented a manager agent to do the merging (as described in Section 4.2), but for efficiency reasons this was later replaced by the agents performing a distributed computation without a manager agent. Further details are given in Section 4.7.

4.3.1 Evaluation Domains

I will now describe the single-agent RL problems which will be used for the purpose of evaluating the parallel methods described in this thesis.

Mountain-Car Task

The first evaluation domain we will use is the *Mountain-Car Task*, as described in Sutton and Barto (1998). This is probably the most well known benchmark problem for RL algorithms. 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 4.3). 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.

The state of the Mountain-Car task is described by the position x_t and velocity v_t of the car. There are three actions, which set the car's acceleration a_t to either

[1]Note that 100Mbs Ethernet is a cheap but not particularly fast network for a cluster. High speed cluster interconnects have a lower latency than Ethernet and commonly achieve 2–10Gbps bandwidth.

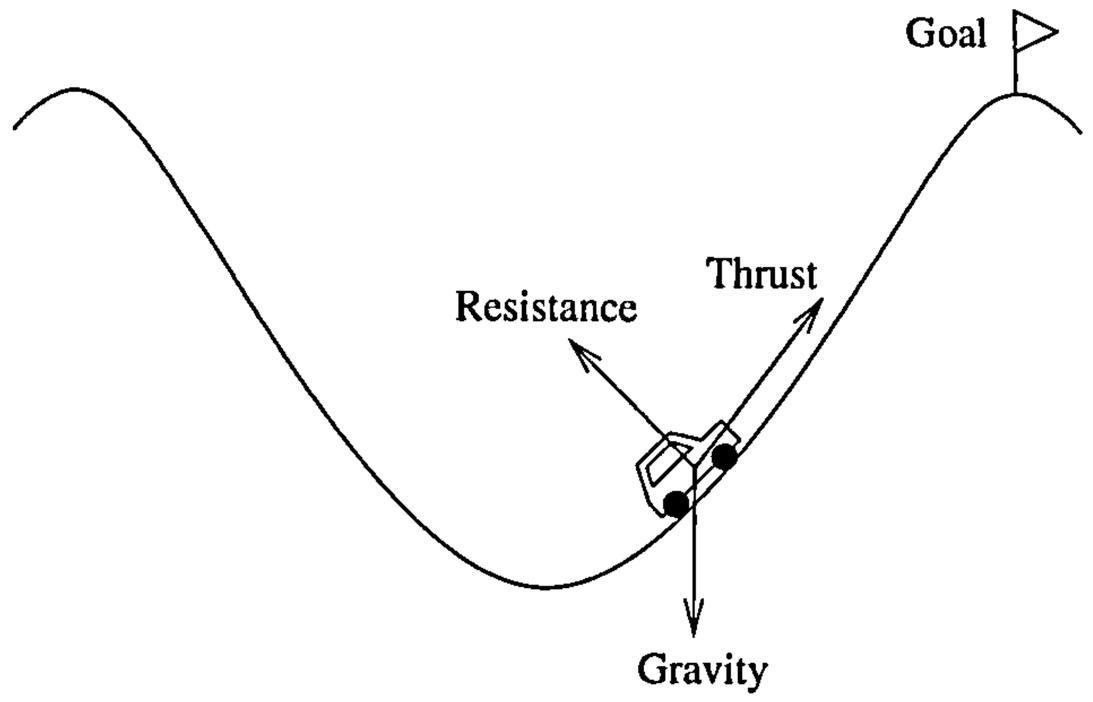


Figure 4.3: The Mountain-Car task.

-1, 0 or $+1$. At each time step, the state is updated according to a simplified physical model:

$$x_{t+1} = x_t + v_t$$

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

After x_{t+1} and v_{t+1} have been calculated, their values are *bounded* so that they remain in the ranges $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq v_{t+1} \leq 0.07$. The car begins each episode at rest at position $x = -0.5$. When the car reaches the goal at position $x = 0.5$ the episode is terminated. An optimal policy should *minimize* the number of steps required to reach the goal from the starting position. We have defined and used two different reward functions for this problem:

1. $r = -1$ on every step except when the goal is reached, when $r = 0$.

2. $r = 0$ on every step except when the goal is reached, when $r = 1$.

The approximator features were generated using 10 tilings of size 9x9 for each of the 3 actions. This results in a total of 2430 features. Random offsets are generated for each of the tilings at the start of a run of the experiment. Results graphed using the average of a set of runs therefore reflect the average over the distribution of tiling offsets.

The Mountain-Car task is a *deterministic* problem with a *continuous state space*. However, non-Markovian effects arising from the limited resolution of the coarse binary features mean that in the early stages of learning the environment can appear stochastic to a learning agent with an approximate value function.

Pole-Balancing Task

The next evaluation domain we will consider is the *Pole-Balancing Task*, which is based on the detailed description provided in Barto et al. (1983). Like the Mountain-Car task, this is a deterministic control problem with a continuous state space which is popular for benchmarking RL algorithms. However, the Pole-Balancing task is different in character to the Mountain-Car task because our goal in this problem is to learn to *maximize* the length of episodes.

In this task, a cart is situated on a track which constrains it to move in a single dimension. A pole is hinged to the top of the cart so that the top of the pole can swing freely (see Figure 4.4). The goal of this task is to keep the pole balanced near the vertical for as long as possible. There are only two actions available at each time step: the cart can accelerate at full power in either direction along the track. This kind of problem, where action in one of two opposite directions can not be less than full power, is known as a *bang-bang* control problem. In addition, there is only a small length of track available, so the cart must keep the pole balanced without hitting either end of the track.

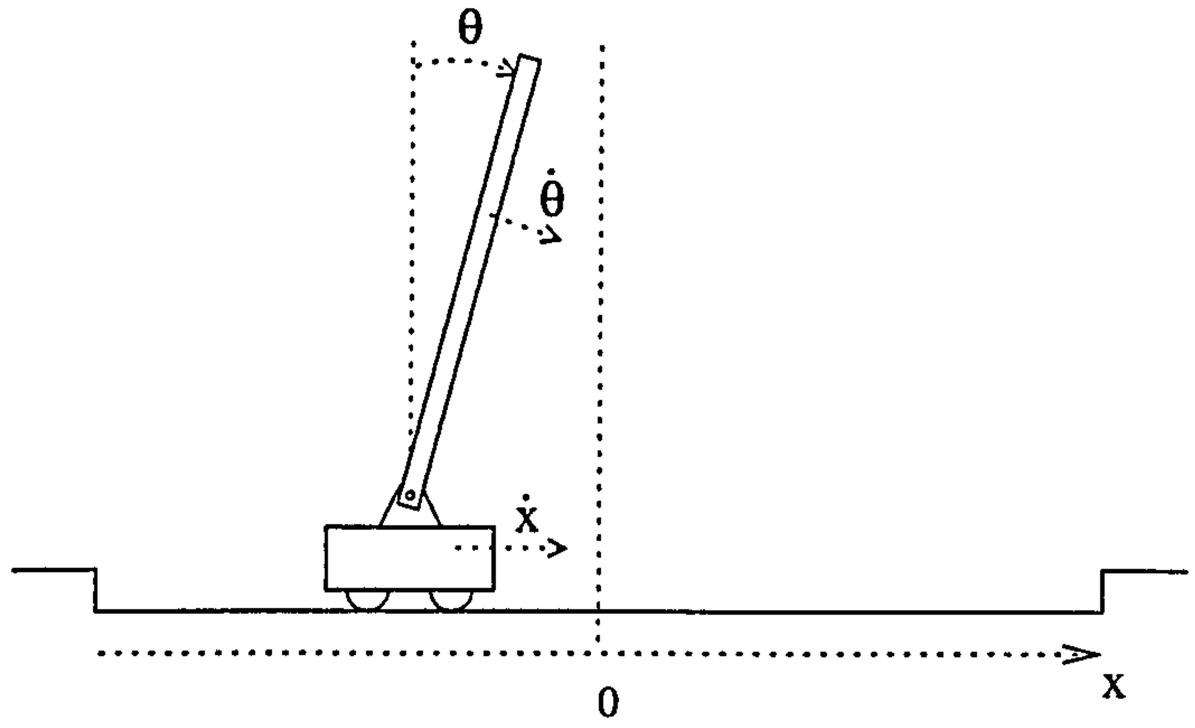


Figure 4.4: The Pole-Balancing task.

The state of this problem is described using four variables. The first two are the cart position x and its rate of change $\dot{x}$. The second two are the angle of the pole θ and its rate of change $\dot{\theta}$. At each time step we first calculate values for the acceleration of the cart and the angular acceleration of the pole:

$$\ddot{\theta}_t = \frac{g \sin\theta_t - \cos\theta_t \left[\dfrac{F_t + l m_p \dot{\theta}_t^2 \sin\theta_t}{m_c + m_p}\right]}{l\left[\dfrac{4}{3} - \dfrac{m_p \cos^2\theta_t}{m_c + m_p}\right]}$$

$$\ddot{x}_t = \frac{F_t + l m_p \left[\dot{\theta}_t^2 \sin\theta_t - \ddot{\theta}_t \cos\theta_t\right]}{m_c + m_p}$$

Constant	Description	Value
g	Gravitational acceleration	$9.8ms^{-2}$
f_{max}	Maximum force applied to cart	10N
m_c	Mass of the cart	$1kg$
m_p	Mass of the pole	$0.1kg$
l	Half-pole length	$0.5m$
τ	Simulation time step	$0.02s$

Table 4.1: Numerical constants used in the Pole-Balancing environment model.

Depending on the action chosen at time t, the force F_t applied to the cart is either $F_t = f_{max}$ or $F_t = -f_{max}$. The various constants used in the model are defined in Table 4.1. Note that in contrast to the model used in Barto et al. (1983) the effects of friction on the cart wheels and the pole hinge are ignored here. Now Euler's method can be used to update the state variables over some time step τ.

$$\begin{aligned} x_{t+1} &= x_t + \tau \dot{x}_t \\ \dot{x}_{t+1} &= \dot{x}_t + \tau \ddot{x}_t \\ \theta_{t+1} &= \theta_t + \tau \dot{\theta}_t \\ \dot{\theta}_{t+1} &= \dot{\theta}_t + \tau \ddot{\theta}_t \end{aligned}$$

Each episode begins with all four state variables set to 0.0. If the system state moves outside the bounds $-2.4 \leq x \leq 2.4$ and $-12° \leq \theta \leq 12°$ then the episode is deemed to have failed, and is terminated. Two different reward functions can be used to allow the learner to maximize the episode length:

1. $r = 1$ on every step except when the episode terminates, when $r = 0$.

2. $r = 0$ on every step except when the episode terminates, when $r = -1$.

The approximator features were generated using 4 tilings for each of the 2 actions. Each tiling partitions each of the x, $\dot{x}$ and $\dot{\theta}$ dimensions into 4 equally-sized regions. The θ dimension is partitioned into 8 equally-sized regions. There are therefore 512 features per tiling, and a total of 4096 features in the entire approximator. Random offsets were generated for the tilings at the start of each run.

Acrobot Task

The *Acrobot Task* described in Sutton and Barto (1998) is another well-known RL problem which is similar in character to the Mountain-Car task. Like the

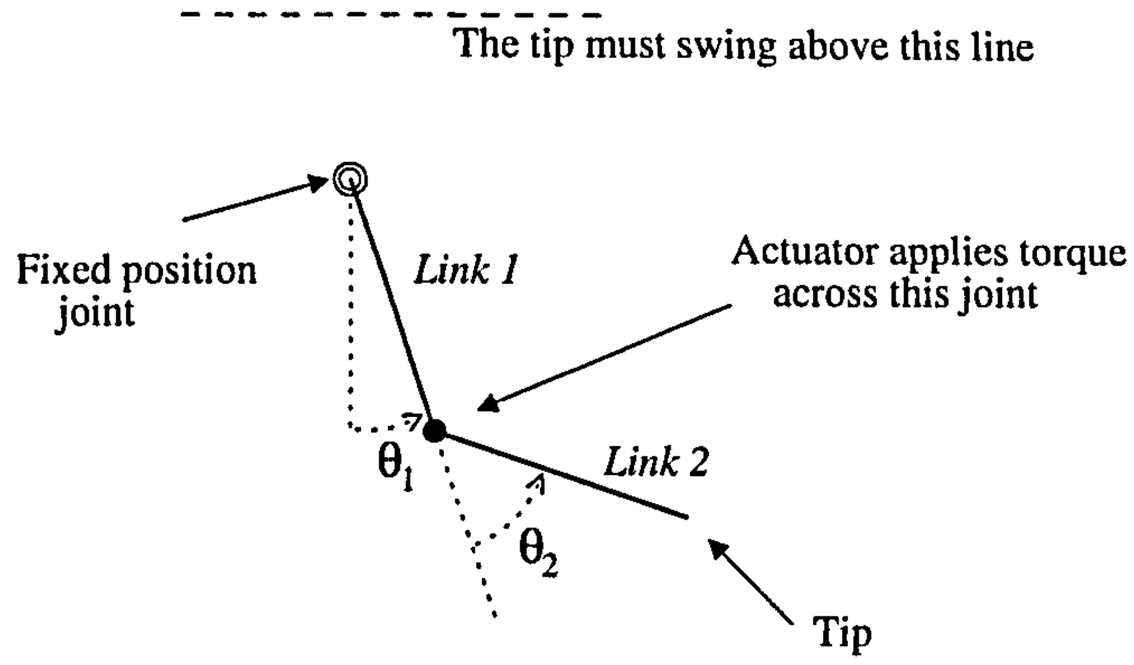


Figure 4.5: The Acrobot task.

Mountain-Car task, it is a deterministic continuous-state control problem where the objective is to minimize the number of steps required to reach a goal state. However, it is generally considered to be the more difficult control problem, since the state has 4 dimensions instead of 2 and the motion in the system is more complex.

The name of the task derives from the way the simple robot modelled by the task is similar to an acrobat swinging on a high bar. The robot consists of two links connected by a joint in a 2 dimensional space (see Figure 4.5). One end of the first link is attached to another joint which has a fixed position in space. The only actuator available to the robot can apply torque across the joint which joins the links. This is similar to the way an acrobat can build up momentum on the high bar by bending at the waist. The goal of the task is to swing the tip of the second link (the acrobat's feet) above a particular height.

Four continuous state variables are required to describe the environment: θ_1, $\dot{\theta}_1$, θ_2 and $\dot{\theta}_2$. The equations of motion which describe the rate of change of $\dot{\theta}_1$ and $\dot{\theta}_2$ are as follows.

$$
\begin{aligned}
\ddot{\theta}_1 &= -d_1^{-1}(d_2\ddot{\theta}_2 + \phi_1) \\
\ddot{\theta}_2 &= \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1} \left(\tau + \frac{d_2}{d_1}\phi_1 - \phi_2\right) \\
d_1 &= m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2} \cos\theta_2) + I_1 + I_2 \\
d_2 &= m_2(l_{c2}^2 + l_1 l_{c2} \cos\theta_2) + I_2 \\
\phi_1 &= -m_2 l_1 l_{c2} \dot{\theta}_2^2 \sin\theta_2 - 2 m_2 l_1 l_{c2} \dot{\theta}_2 \dot{\theta}_1 \sin\theta_2 \\
&\quad + (m_1 l_{c1} + m_2 l_1) g \cos(\theta_1 - \pi/2) + \phi_2 \\
\phi_2 &= m_2 l_{c2} g \cos(\theta_1 + \theta_2 - \pi/2)
\end{aligned}
$$

The values of the constants used in this physical model are given in Table 4.2. The torque τ (in N-m) applied to the actuator joint takes values -1, 0 or $+1$

Constant	Description	Value
g	Gravitational acceleration	$9.8ms^{-2}$
m_1	Mass of link 1	$1kg$
m_2	Mass of link 2	$1kg$
l_1	Length of link 1	$1m$
l_2	Length of link 2	$1m$
l_{c1}	Length to centre of mass of link 1	$0.5m$
l_{c2}	Length to centre of mass of link 2	$0.5m$
I_1	Moment of inertia of link 1	$1kgm^2$
I_2	Moment of inertia of link 2	$1kgm^2$

Table 4.2: Numerical constants used in the Acrobot environment model.

depending which of the three available actions is chosen. Given that $\ddot{\theta}_1$ and $\ddot{\theta}_2$ can be calculated using these equations, we can use Euler's method to update the state variables θ_1, $\dot{\theta}_1$, θ_2 and $\dot{\theta}_2$. While the RL agent chooses a new action every $0.2s$, within each of these time steps we use smaller *substeps* of $0.05s$ to calculate new values for the state variables using Euler's method.

Each episode begins with all state variables set to 0.0, i.e. with the acrobot hanging straight down at rest. The angular velocities are bounded to remain within the ranges $-4\pi \leq \dot{\theta}_1 \leq 4\pi$ and $-9\pi \leq \dot{\theta}_2 \leq 9\pi$. The goal is to raise the tip $1.45m$ above the fixed position joint, at which point the episode ends. Two different reward functions can be used in order to *minimize* the episode length:

1. $r = -1$ on every step except when the goal is reached, when $r = 0$.

2. $r = 0$ on every step except when the goal is reached, when $r = 1$.

The learning features for the RL agent are generated as follows. There are four continuous dimensions to the state space: θ_1, $\dot{\theta}_1$, θ_2 and $\dot{\theta}_2$. There are therefore 4 ways to select a group of 3 dimensions (leaving one out each time). For each possible group of 3, we create 3 randomly offset tilings which divide each dimension in the group into 8 regions. This results in a total of 12 tilings for each action, and therefore a total of 18,432 learning features.

Stochastic Grid World Task

In addition to the three well known RL problems described above, a domain was needed which would allow problems of increasing difficulty to be defined in order to investigate performance in large-scale problems. For this purpose, we used a

stochastic grid world domain which has some similarity to the *Puddle-world* domain (Sutton, 1996).

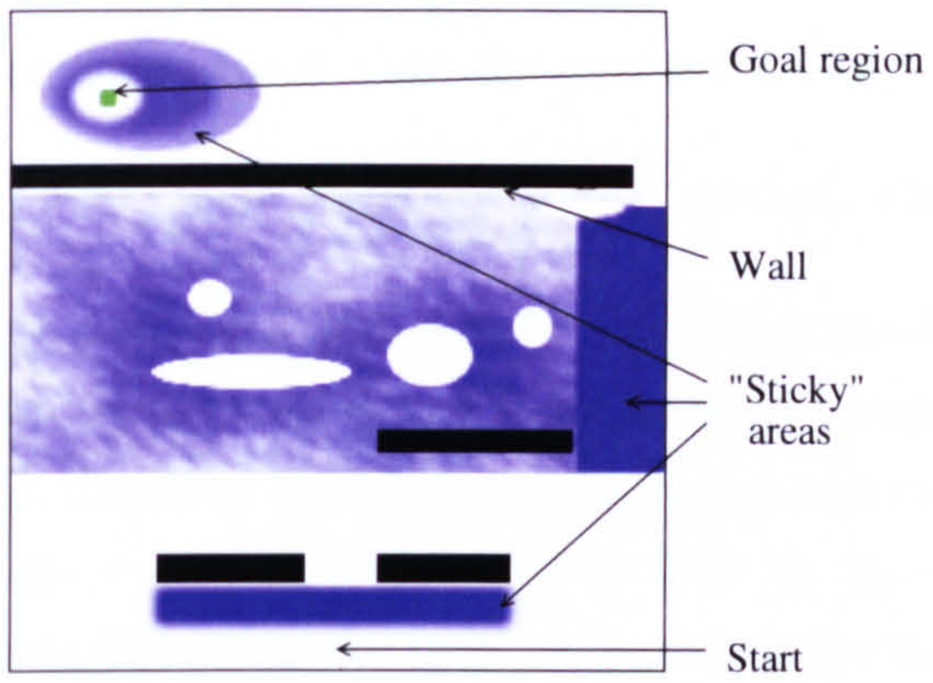


Figure 4.6: A bitmap image defining an instance of the stochastic grid world task.

An instance of the grid world is defined by a bitmap image file, such as the one shown in Figure 4.6. A red pixel indicates the starting point of an agent at the beginning of every episode. Four actions are available to the agent: *up*, *down*, *left* and *right*. Each action moves the agent a distance s in the specified direction. The units of s are measured in pixels, but s is not constrained to be 1, or even a whole number of pixels. The distance s can be any real number. This means that, in contrast to some grid world domains, the state space is not a discrete grid but a *continuous* 2D state space $\{(x, y)|x \in [0, x_{max}], y \in [0, y_{max}]\}$. This allows a domain instance defined by a single image to be made progressively more difficult by reducing the distance s. Alternatively, more difficult problems can be created by keeping the step size constant, and generating larger images.

Black pixels in the image denote *walls*, which are impassable. The edges of the image are also impassable. Any movement action which would take the agent into a wall or off the edge of the image will fail, leaving the agent at the same position. A group of green pixels indicates a goal region to which the agent must travel. The overall objective is to *minimize* the number of steps required to reach the goal region.

Blue pixels represent *sticky areas*. When the current state (x, y) of the agent is within a sticky area, any movement actions taken in that state will have stochastic outcomes. The more saturated the blue area is, the greater the probability that a movement will fail, leaving the agent in the same position. Suppose the pixel corresponding to (x, y) has a colour (r, g, b), where $r, g, b \in [0, 1]$. Sticky areas are defined as those where $b = 1$, $r < 1$ and $r = g$. The probability of an action failing in a sticky area is defined as:

$$p(fail) = 0.9(1 - r)$$

The two reward functions used for this problem are the same as those used in the Mountain-Car and Acrobot tasks:

1. $r = -1$ on every step except when the goal is reached, when $r = 0$.

2. $r = 0$ on every step except when the goal is reached, when $r = 1$.

For approximation we use a *single* two dimensional tiling, which has a similar effect to using a table-based representation and a simple discretization of the continuous state space (although the offset of the tiling remains random). The number of tiles and the move distance s vary according to how difficult we want the problem to be. Here are two particular instances that are based on the 256x256 image shown in Figure 4.6:

Low Difficulty A 30x30 tiling generates features for each of the four actions, resulting in a total of 3600 features. Movement distance $s = 2$.

High Difficulty A 64x64 tiling generates features for each of the four actions, resulting in a total of 16,384 features. Movement distance $s = 1$.

4.4 Comparing Merging Functions

In this Section, I will present several candidates for the merge function f, whose purpose was described in detail in Section 4.2. The motivation behind the choice of each of candidate function is given, in addition to an evaluation of each function using the Mountain-Car task and a simulation of parallel agents.

4.4.1 The Minimum Merge Function

A simple approach to exploration in RL is the use of *optimistic initial values*. By initialising an agent's value function so that all state-action pairs appear to lead to high rewards, the agent can follow a *greedy* policy (no actions are explicitly explorative) and still converge to the optimum in deterministic domains. This works because in a given state, each action in turn becomes the greedy choice as updates to the value function reduce the value of overestimated actions. Once the value of the best action is reduced to its true optimal value, the value remains unchanged, and so at this stage the greedy action is in effect the optimal action. This can be seen as a process of reducing the upper bound on each action value until the largest bound in each state is *tight*.

A parallel approach based on a similar idea is for each parallel agent to work on reducing the upper bounds. The agents' results can then be combined by taking

the *minimum* upper bound established by the group for each state-action pair. Thus we initially overestimate the value of each state-action pair, and thereafter at each merge preserve the minimum value estimate from the set of agents. The form of merge function f is therefore:

$$f(\theta_{1,i}, \theta_{2,i}, ..., \theta_{n,i}) = \min_j \theta_{j,i}$$

Note that a purely greedy policy can not be used for the parallel approach since the agents would have identical experiences, and parallel speedup by merging is only possible if there is some diversity in the set of agents. Therefore, each agent must take some number of explorative actions.

The minimum merge function was evaluated using the Mountain-Car task. Each of the agents used the SARSA(λ) algorithm with an ϵ-greedy exploration strategy as described in Section 4.2. All weights of the agents' VFAs were initialised to 0. Reward function #1 (see Section 4.3.1) was used, meaning that a reward of -1 was given on every time step except when the goal was reached. Initialising the weights to zero means that the return for every state-action is initially overestimated.

The parameters used were as follows: merge period $p = 250$, exploration parameter $\epsilon = 0.05$, learning rate $\alpha = 0.5$, discount factor $\gamma = 0.99$, and eligibility trace parameter $\lambda = 0.9$. Each episode was ended after 300 steps if the goal was not reached, and results were averaged over 200 runs. The results are shown in Figure 4.7.

Over the first 5 to 10 thousand time steps having a larger number of agents means that performance is improved at a faster rate, which indicates that agents are successfully combining information about which actions have a poor return. However, in the later stages of the experiment, the performance becomes worse at a rate that increases with the number of agents. This indicates that the use of minimum merging breaks the conditions for convergence of the underlying SARSA(λ) algorithm.

Performance becomes worse in the later stages because of the stochasticity introduced by the ϵ-greedy exploration strategy. The use of occasional explorative actions mean that the value of states can be *underestimated* when a random action choice results in a low return. A single-agent learner is probabilistically likely to eventually correct this underestimate. The parallel agents, however, are less likely to correct the underestimate, since merging preserves the minimum value of each weight. Unless all the agents revise a weight upwards in the same merge period p, an underestimated weight cannot be corrected. The underestimates will eventually propagate throughout the value function, making the performance worse and worse.

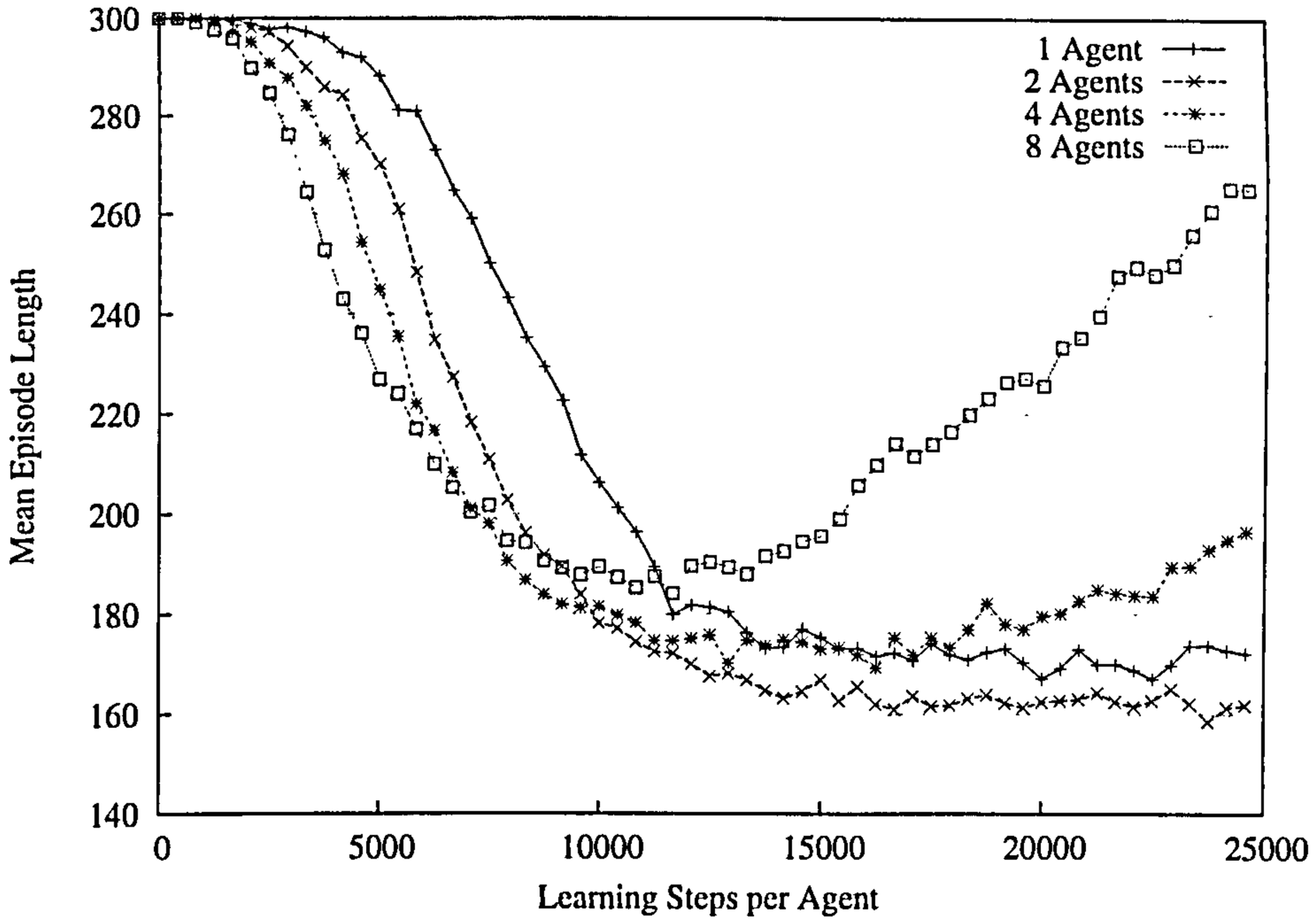


Figure 4.7: Results for the *minimum* merge function in the Mountain-Car task.

4.4.2 The Maximum Merge Function

The *maximum* merge function is the dual of the minimum merge function. The intuitive idea behind this approach is that the VFAs are initialised to *underestimate* the value of each state-action pair, and the agents gradually revise these values upwards as rewards are discovered in the environment. The values can be viewed as *lower bounds* on the return of an action. Agents can merge their experience by keeping the *maximum* lower bound established for each state-action pair. The maximum merge function f takes the form:

$$f(\theta_{1,i}, \theta_{2,i}, ..., \theta_{n,i}) = \max_j \theta_{j,i}$$

Reward function #2 (see Section 4.3.1) is used in this experiment, where the only reward is +1 on time steps when the goal is reached. Initialising all the weights to 0 would be a simple way to underestimate the value of all state-action pairs. However, since the reward will be 0 until the first time the goal is reached, updates to the value function would have no effect during this period—all the weights would remain at 0. The result would be that there is no way for the agent to track which state-action pairs have already been visited, forcing the agent to follow essentially a random walk behaviour until the goal is accidentally discovered. This means the time to find a good policy will be at least an order of magnitude longer than the earlier experiment.

To avoid this, each weight is initialised to some small value $\theta_{init} > 0$. This value is chosen to be small enough so that each state-action pair is still underestimated. Since θ_{init} is non-zero, updates to the value function do have an effect, and state-action pairs which *have* been explored will have lower values than those which have not until the goal is found. The ϵ-greedy strategy can now find the goal in a reasonable time.

The Mountain-Car task evaluation for the maximum merge function used the settings given in Section 4.4.1, except that reward function #2 was used and $\theta_{init} = 0.0001$ instead of 0. The results are shown in Figure 4.8.

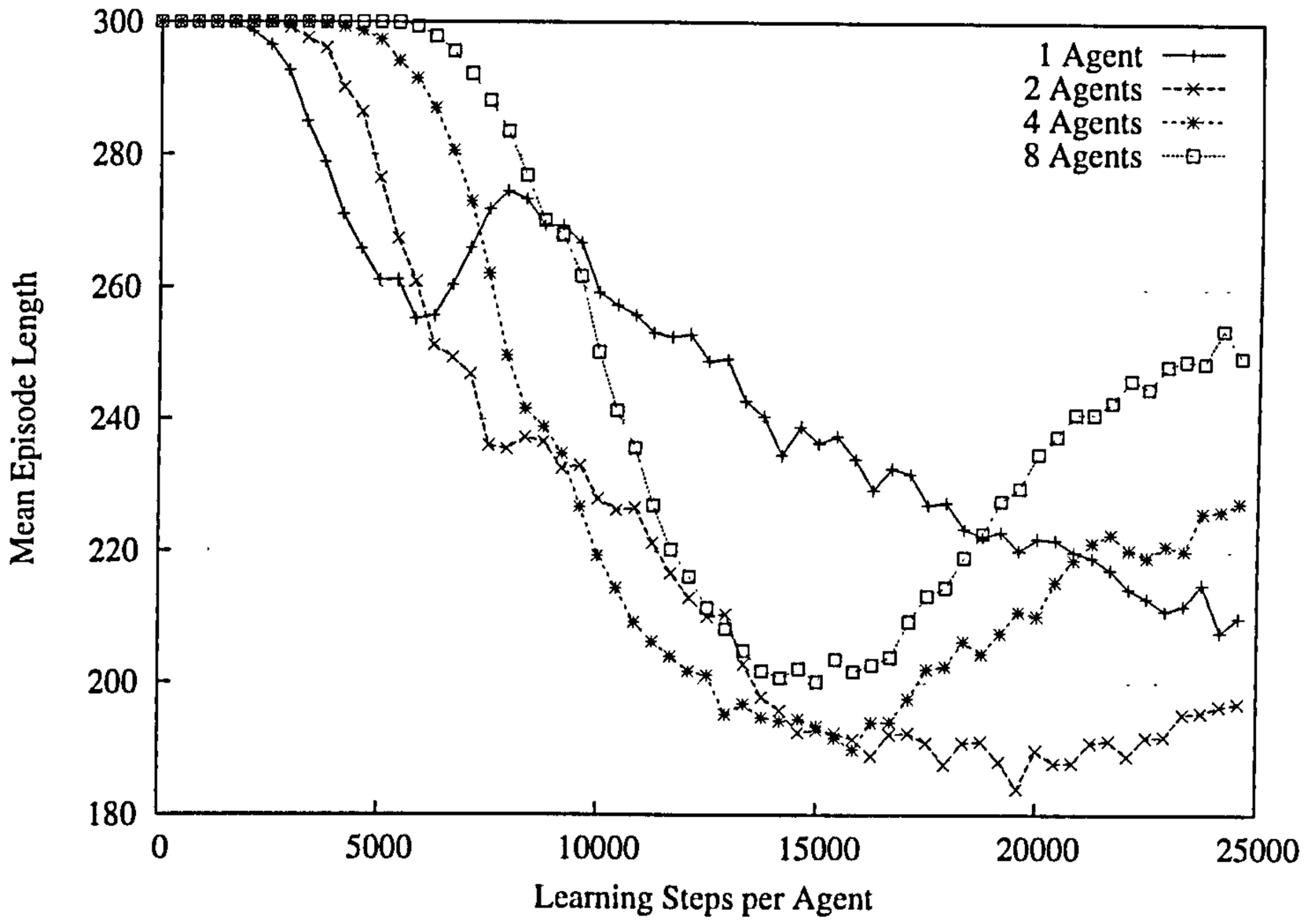


Figure 4.8: Results for the *maximum* merge function in the Mountain-Car task.

Observe that in these results there is a delay at the start of the experiment until the agents begin to achieve a performance better that 300 steps per episode. The delay appears to be longer the more agents there are. However, once the delay is over, multiple agents improve performance more quickly than the single agent. In the later stages of the experiment performance starts to become worse for the larger sets of parallel agents.

The initial delay corresponds to the time before the goal has been reached by any of the agents. During this phase of learning, the agents gradually reduce the initial weights as the environment is explored, effectively marking out which actions are better explored. The maximum merge *interferes* with this phase, causing the agents as a group to *forget* some parts of the exploration as the maximum value of each weight is taken. The more agents there are, the greater the interference with

this initial phase of exploration.

A mechanism to eliminate this increased delay can be introduced as follows. Define some value θ_{limit} which separates the weight values into two distinct regions. Weights $\leq \theta_{limit}$ only arise from the initial exploration phase, before any rewards are found in the environment. Any weight $> \theta_{limit}$ must have been updated on a path to a goal, and ought to work well with the maximum merge function. Once the maximum merged weights $\{\theta_i^m\}$ have been calculated, each agent only copies into its value function those merged weights greater than θ_{limit}:

$$\begin{aligned} \text{if } \theta_i^m > \theta_{limit} \quad &\text{then} \quad \theta_i^{t+1} \leftarrow \theta_i^m \\ &\text{else} \quad \theta_i^{t+1} \leftarrow \theta_i^t \end{aligned}$$

The results for an experiment using this mechanism are shown in Figure 4.9. The experimental settings are identical to those used for the results in Figure 4.8 except that the above mechanism is used with $\theta_{limit} = 0.0002$. The results show that this mechanism eliminates the extra delay exhibited in the first experiment. However, since merging is essentially inhibited for the first 2 to 3 thousand steps, there is no way that this phase of the experiment can be speeded up by adding more agents.

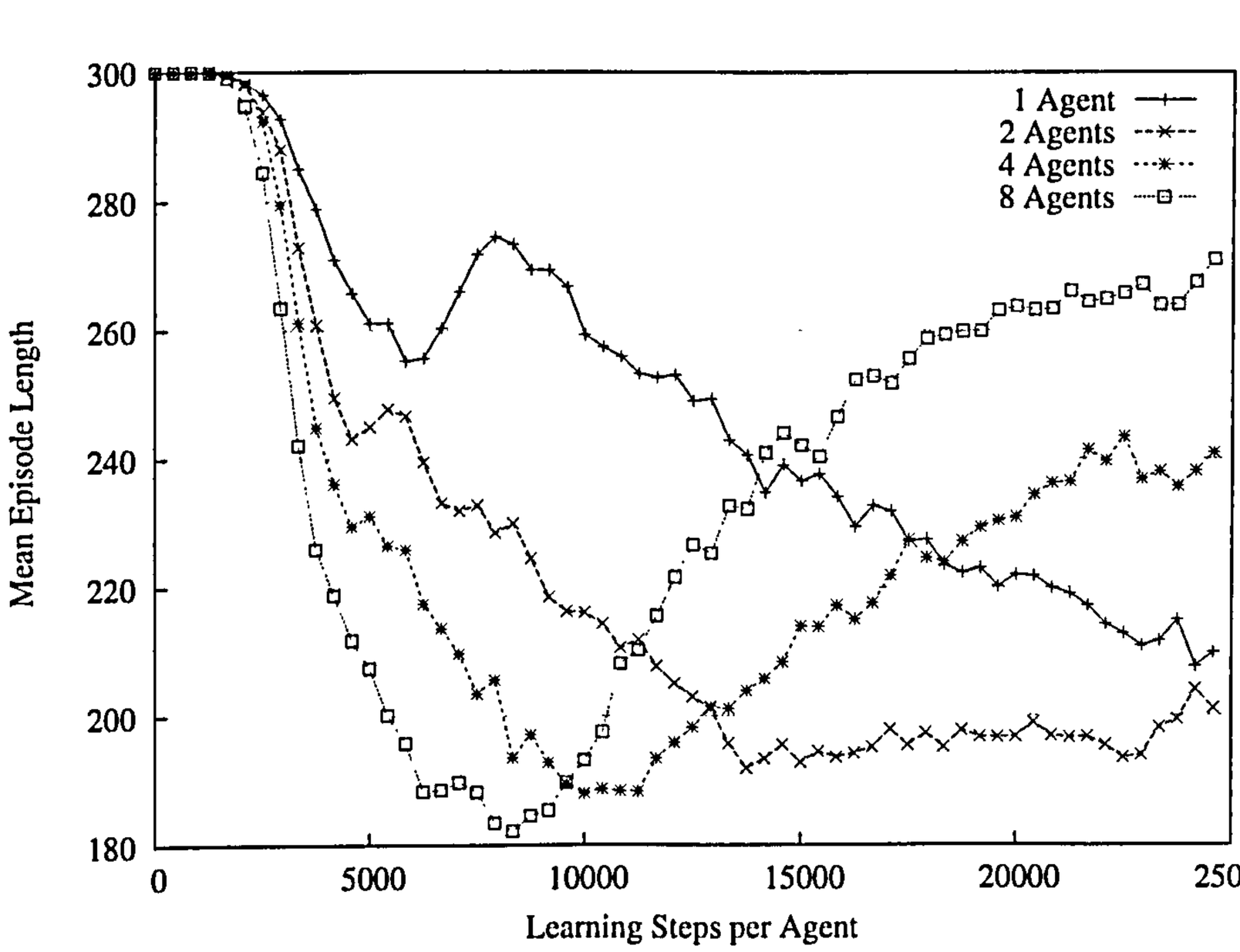


Figure 4.9: Results for the *maximum* merge function with $\theta_{limit} = 0.0002$ in the Mountain-Car task.

The results in Figure 4.9 show more clearly the rapid improvement in performance achieved by the maximum merging function in the early stages of the

experiment. The improvement in learning speed is much greater than that seen with the minimum merge (see Figure 4.7). This could be because learning with reward function #2 is inherently more parallelizable. Alternatively, it could be because the maximizing behaviour of the merging function is closer in character to a Bellman update than the minimum merging function. In any case, in the later stages of the experiment performance again becomes worse, suggesting that the stochasticity introduced by the ϵ-greedy strategy causes state-action values to be repeatedly overestimated, interfering with convergence.

4.4.3 The Mean Merge Function

The *mean* merge function simply calculates each merged weight using the mean of all the agents' estimates for the weight:

$$f(\theta_{1,i}, \theta_{2,i}, ..., \theta_{n,i}) = \frac{1}{n} \sum_{j=1}^{n} \theta_{j,i}$$

The mean merge function is a natural way to combine weight estimates from a number of agents, where equal significance is given to each agent's estimate. While there is no mechanism to prioritize information about large rewards discovered by only one or two of the agents, this mechanism will improve estimates of immediate stochastic rewards, and provide an approximate summary of the "group knowledge" of the set of agents.

Since the mean merge requires no assumptions about optimistic or pessimistic initial values, we can use either of the two reward functions defined in Section 4.3.1. In either case each weight is initially set to $\theta_{init} = 0.0001$. The rest of the settings for these experiments are identical to those given in Section 4.4.1.

The results for the mean merge function using reward function #1 and #2 are shown in Figures 4.10 and 4.11 respectively. In these experiments, we do not observe worsening performance towards the end of the experiment as we did with the earlier merge functions. This suggests that the mean merge function does not break the convergent properties of the underlying SARSA(λ) algorithm. However, we do not see the rapid improvements in performance that were observed using the minimum and maximum merges (compare Figures 4.7 and 4.10, and also Figures 4.9 and 4.11.) While the mean merge function provides better estimates of stochastic returns, it performs poorly because large weight changes discovered by a single agent will be drowned out in the average by the rest of the agents making small or no changes.

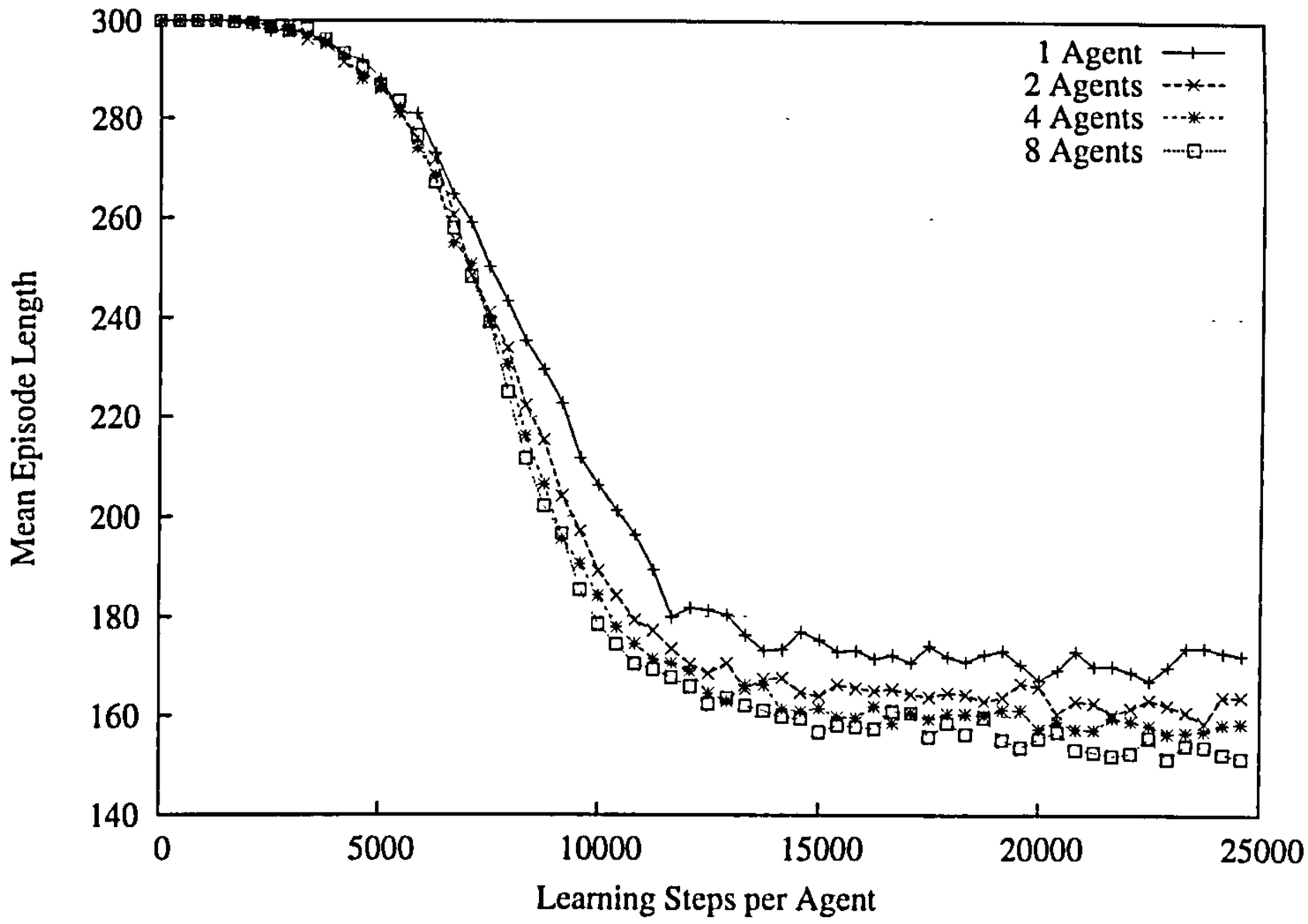


Figure 4.10: Results for the *mean* merge function using reward function #1 in the Mountain-Car task.

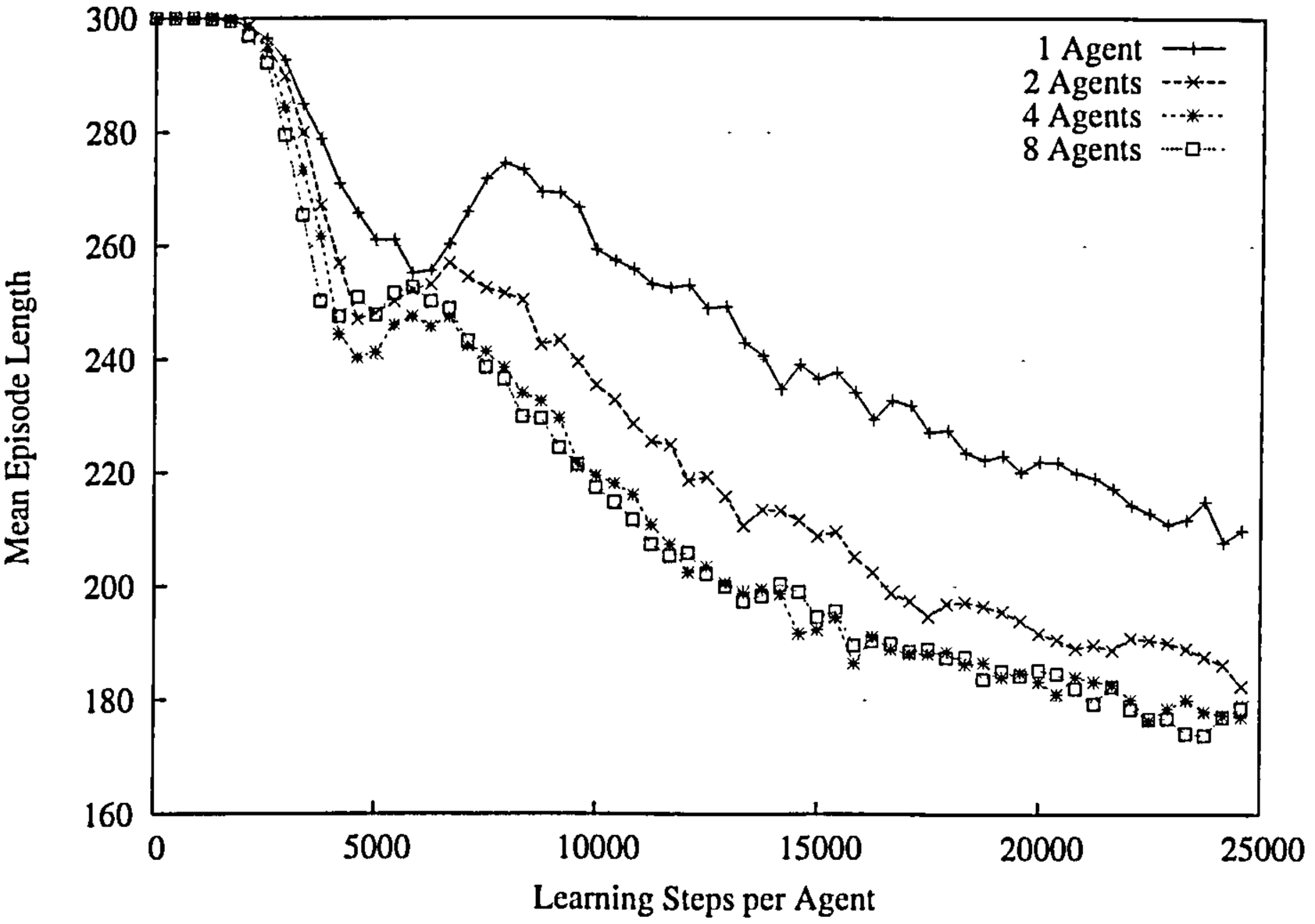


Figure 4.11: Results for the *mean* merge function using reward function #2 in the Mountain-Car task.

4.4.4 The Visit-Count Merge Function

One of the reasons that the mean merge is poor at combining the agents' knowledge is that there is no mechanism for measuring the *relative experience* of the agents in different areas of the state space. Suppose we want to combine the agents' estimates for one of the weights θ_i. Suppose also that only one of the agents in the set has actually visited an area of state space where feature ϕ_i was active. This agent will have updated the value of θ_i several times, but the other agents will still have the initial value for θ_i. In such a situation it is clear that one of the agents has a much better estimate of θ_i, but the mean merge function weights all the agents' estimates equally, and so some of the valuable information discovered by this one agent may be lost in the process of merging.

The weights $\{\theta_i\}$ provide only the best current estimate of each weight. It is not possible to extract a measure of experience directly from the set of weights. The minimum and maximum merge functions got around this problem by considering the agent which had most extended the lower/upper bound on θ_i to be the most experienced. To improve on the performance of the mean merge function it is necessary to store *additional data* to measure the experience of each agent.

For each feature ϕ_i, an agent will now store a *visit-count* c_i in addition to a weight θ_i. The visit-count c_i measures the number of times feature ϕ_i has been *active* during the current merge period (of length p). At the beginning of a merge period, all the $\{c_i\}$ values are set to zero. Every time a state is visited where ϕ_i is active, the value of c_i is incremented. The *visit-count merge function* can now be used to calculate the merged value function:

$$f(\theta_{1,i}, ..., \theta_{n,i}, c_{1,i}, ..., c_{n,i}) = \frac{\sum_{j=1}^{n} c_{j,i}\theta_{j,i}}{\sum_{j=1}^{n} c_{j,i}}$$

Here $c_{j,i}$ is the i^{th} visit-count of agent j. Note that this function takes the agents' visit counts as arguments as well as the agents' weights. It is essentially a weighted average, with greater emphasis given to those agents with larger visit-counts for a particular feature. Note also that if one of the features has been continually inactive for *all* the agents since the last merge (i.e. $\forall j.(c_{j,i} = 0)$) then the value of f is *undefined.* When these situations are detected, θ_i^m is assigned the value of θ_i before the merge took place (all the agents will have retained an identical value for θ_i in such cases).

The results for reward functions #1 and #2 using the visit-count merge function are shown in Figures 4.12 and 4.13 respectively. The settings for the experiment are identical to those used in Section 4.4.3 (other than the use of a different merge function).

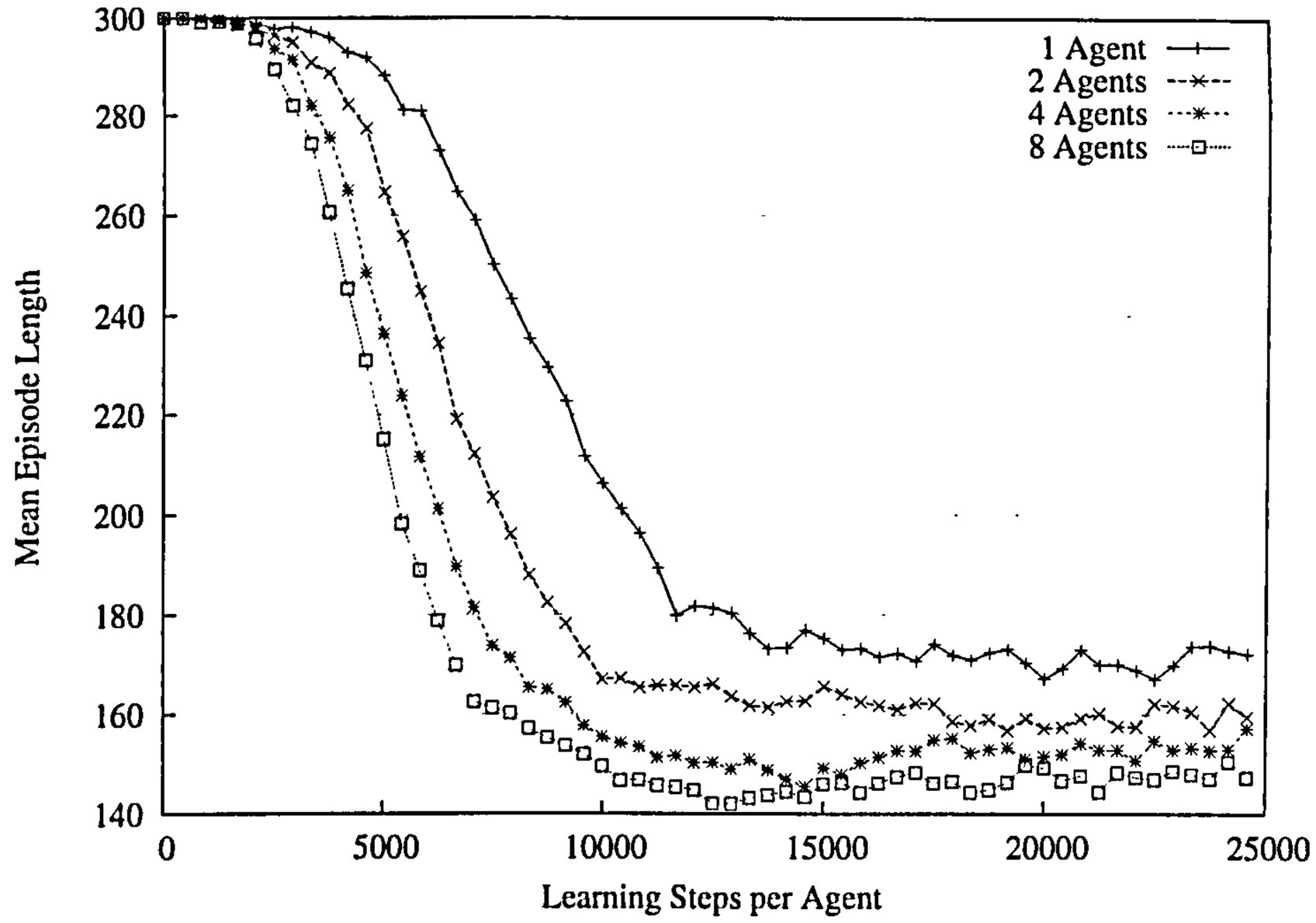


Figure 4.12: Results for the *visit-count* merge function using reward function #1 in the Mountain-Car task.

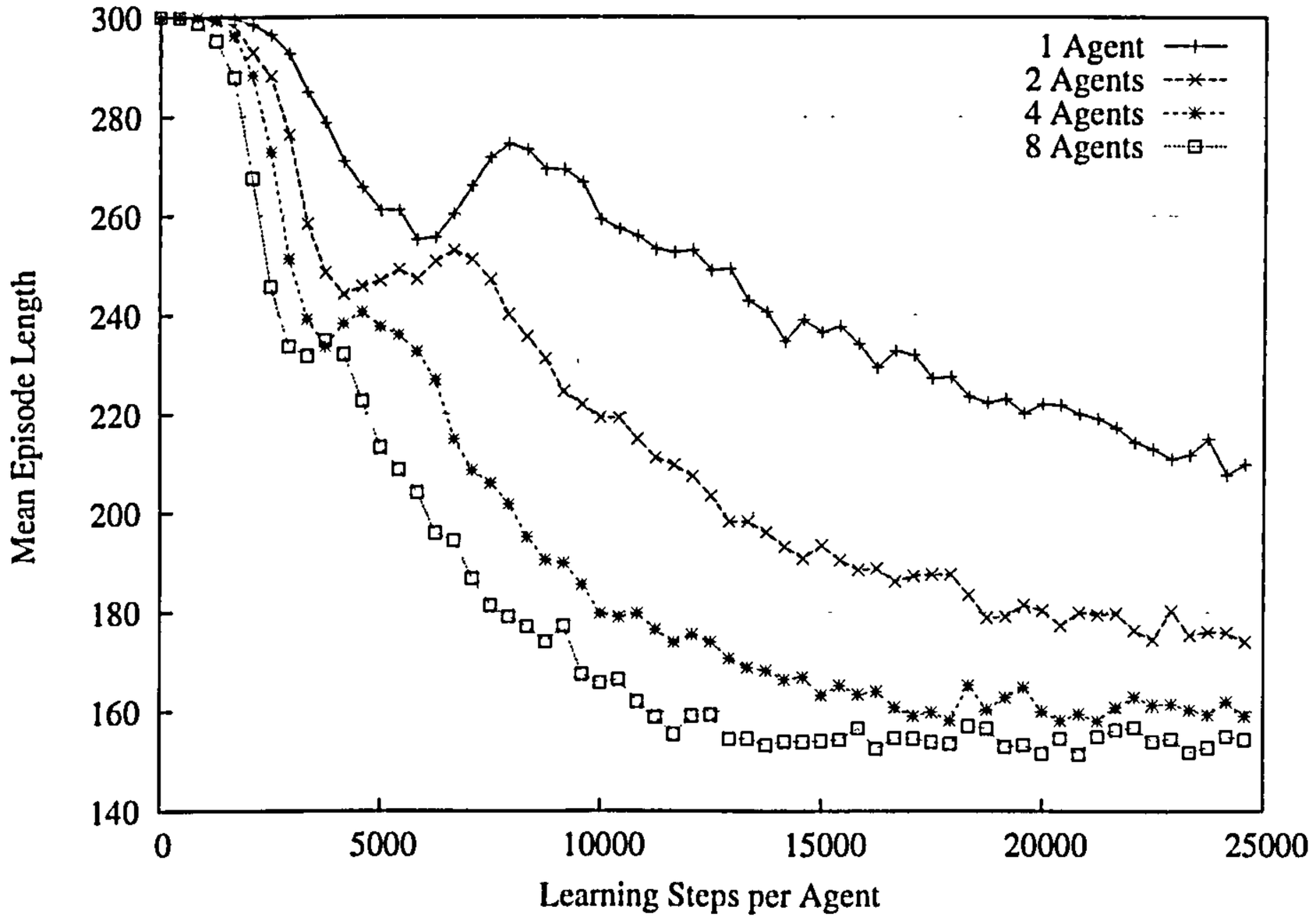


Figure 4.13: Results for the *visit-count* merge function using reward function #2 in the Mountain-Car task.

The visit-count merge function improves performance in the Mountain-Car task more than any of the other merge functions. Like the mean merge function, convergence of the underlying SARSA(λ) algorithms is not affected—the performance does not get worse towards the end of the experiment. Additionally in the early stages of the experiments we observe rapid improvements in performance, similar to those observed using the minimum and maximum merge functions, but without the eventual divergent behaviour. It is fairly clear that identifying the relative experience of the agents in each area of the state space is an important step in exploiting value function information from several agents.

The visit-count merge function achieved the best improvement in performance for all the domains we used with the simulation of parallel agents. Some results for the visit-count merge function in the Pole-Balancing and Acrobot domains are shown in Figures 4.14 and 4.15 respectively.

The results for the Pole-Balancing experiment shown in Figure 4.14 were generated using a reward function where a reward of +1 was received on every time-step except on terminal steps. At the beginning of each run weights were initialized to 0. Episodes were allowed to continue for a maximum of 20,000 steps. A merge period p of 100 was used. The other parameters were $\alpha = 0.2$, $\epsilon = 0.1$, $\gamma = 0.99$ and $\lambda = 0.5$. Results were averaged over 200 runs.

The Acrobot experiment (see Figure 4.15) used a reward function where a reward of -1 was received on every non-terminal time step. Weights were initialized to 0 at the beginning of each run. Episodes were allowed to continue for a maximum of 600 steps. A merge period p of 100 was used. The other parameters were $\alpha = 0.1$, $\epsilon = 0.05$, $\gamma = 1.0$ and $\lambda = 0.9$. Results were averaged over 100 runs.

Similar results were obtained for both the instances of the Stochastic Grid World task, with the visit-count merge function producing the best performance out of the four merge functions evaluated. Graphs for these results are not included in this section, but later in the chapter a graph (Figure 4.22) is included which illustrates the performance (in the simulation of parallel agents) of the visit-count merge function in the low-difficulty Stochastic Grid World.

4.4.5 Comparison Summary

The *minimum* and *maximum* merge functions initially allow the group of agents to converge more quickly towards the optimal policy. The maximum merge function in particular produces rapid policy improvement with larger numbers of agents. However, eventually both of these merge functions cause the agents to *diverge* from the optimum, with the divergence occurring more quickly the greater the number of agents there are in the group.

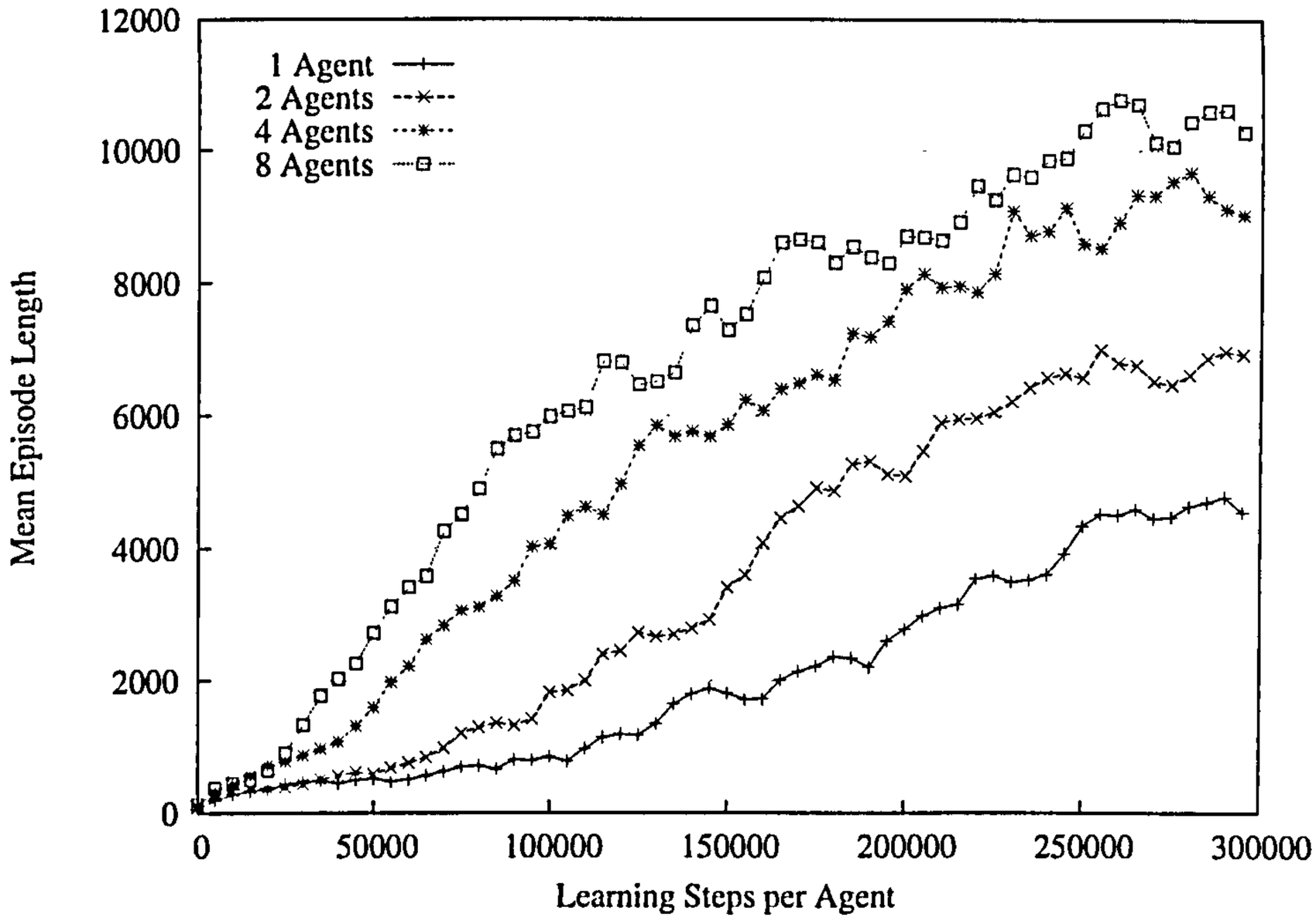


Figure 4.14: Results for the *visit-count* merge function in the Pole-Balancing task.

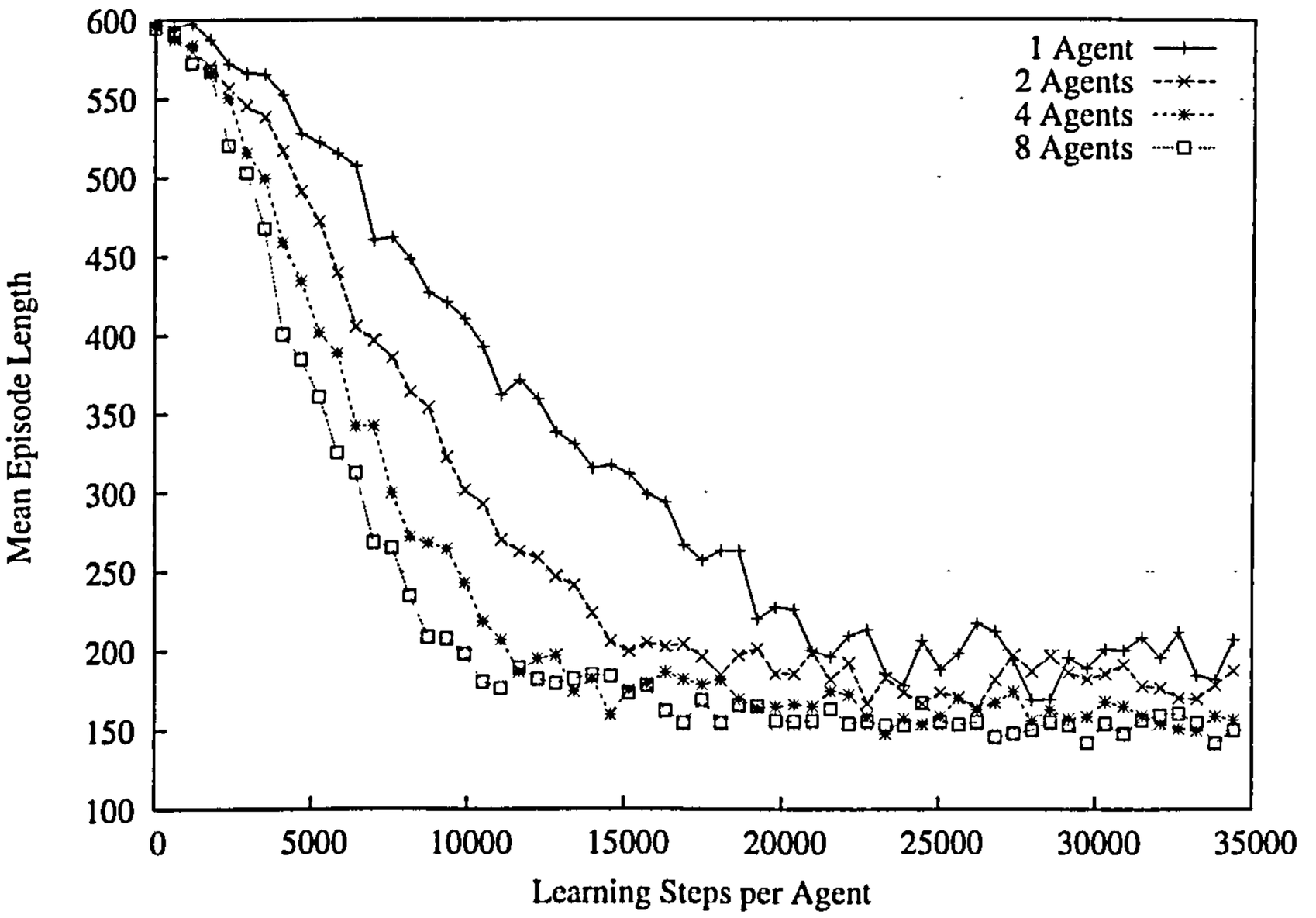


Figure 4.15: Results for the *visit-count* merge function in the Acrobot task.

The *mean* merge function, on the other hand, will reliably converge to the optimal policy. However, the rate of convergence improves relatively slowly as more agents are added to the group. The rapid policy improvement exhibited by the maximum merge function in the initial stages cannot be reproduced.

The *visit-count* merge function requires more information to be exchanged between the agents, but produces the best performance out of the four merge functions evaluated. Policy improvement in the initial stages is almost as rapid as that of the maximum merge function, but without the risk of divergence in the later stages. The visit-count merge produced the best performance in all of the evaluation domains tested: the Mountain Car task, the Pole-Balancing task, the Acrobot task and the two instances of the Stochastic Grid World task.

4.5 Decaying Parameters and Binary Search

An interesting property of the results shown in Figures 4.12, 4.13 and 4.15 is that the final quality of policy learned is different depending on the number of parallel agents used. This remains true even if the experiments are run over arbitrarily long periods of time. The reason this effect arises is down to two key properties of the experiment:

- The learning rate α and exploration parameter ϵ have *fixed* non-zero values.
- The visit-count merge function is based on (weighted) *averaging.*

The use of fixed values for α and ϵ means that no matter how long the experiment is, each agent will continue to take the same proportion of random actions and update its VFA according to the outcome. One consequence of this is that at all times during the experiment there will be a small chance that a series of exploratory actions will result in poor rewards, with subsequent updates significantly changing the VFA. This means that there will always remain a small chance of making a large step *away* from the optimal policy. A second consequence is that a true optimal policy may never be found if the return from a given state is significantly different depending on whether a greedy or an ϵ-greedy strategy is followed.

The *averaging* behaviour of the visit-count merge function seems to have a *noise-reducing* effect, which reduces the chance of making a large step away from the optimum. While each of the individual agents in the group will have the same chance as the single agent of making a change which moves away from the optimum, it is likely that others in the group will not make such a change at exactly the same time. Once the VFAs are averaged across the agent group, the step away from the optimum is much smaller. Over time, this means that a group of agents

can maintain a policy much closer to the optimum than a single agent, provided that α and ϵ remain fixed and that merge phases occur every p steps throughout the agents' lifetimes.

In practice, we are only likely to have parallel hardware available for a limited time, after which a policy must be extracted from one of the group of agents. Additionally, the parallel hardware may only support a small number of parallel agents. In order to learn a policy arbitrarily close to the optimum without requiring large numbers of agents, it is necessary that parameters α and ϵ *decay* over the course of the experiment. By running a given experiment for a longer time with a slower rate of decay, we can get closer to the highest quality policy representable using the VFA.

Singh et al. (2000) proved that tabular SARSA(0) with ϵ-greedy exploration converges to the *optimal* value function Q^* if α decays in a way that satisfies the Robbins-Monro criteria (Robbins and Monro, 1951) and ϵ tends towards zero over time at a rate which ensures that each state-action pair is explored infinitely often in the limit. However, while this proof of asymptotic optimality suggests that decaying α and ϵ will also be necessary for approximate SARSA(λ) to approach the optimum, it provides no indication of how fast to decay their values if only a *finite learning time* is available. The goal in this context is to approach as close to the optimal approximation as is possible in the limited time available.

To my knowledge there has not been an analytical study of appropriate mechanisms to decay these parameters over a finite time. However, previous empirical work (Rummery and Niranjan, 1994; Loch and Singh, 1998; Claus and Boutilier, 1998) has shown that good results can be achieved in practice by decaying these parameters to zero at a steady rate over the time available. Appropriate *initial* parameter values (before decay begins) must be determined for each learning domain. In this thesis α and ϵ are decayed *linearly* using *identical rates*, since this approach seemed to produce the best policies in the empirical evaluation. This means that over the course of a run each agent's control policy becomes increasingly greedy. In addition, the decreasing learning rate allows the *expectation* of the value of each stochastic action to be more closely approximated.

For the purposes of parameter decay, time is measured as the proportion completed of a single run of an experiment (recall from Section 4.3 that the time limit for a run may be measured in *episodes, time steps* or *real time*). Define time $t = 0$ as the start of the run and $t = 1$ as the end of the run. The values of the initial parameters at time $t = 0$ are defined as α_0 and ϵ_0. A time $t_{lim} \in [0, 1]$ is chosen to be the time at which the parameters must have decayed to zero. The α and ϵ

parameters now become functions over time:

$$\alpha(t) = \max\left(0, \frac{\alpha_0(t_{lim} - t)}{t_{lim}}\right)$$
$$\epsilon(t) = \max\left(0, \frac{\epsilon_0(t_{lim} - t)}{t_{lim}}\right)$$

In practice it may be inefficient to recalculate the values of α and ϵ on every time step. It is sufficient to choose some small number of time steps q (e.g. $q = 25$ in the following experiments) to define an interval or *quantum*. After every q time steps new values for the parameters are calculated using the above equations. This works well as long as q is much smaller than the total number of time steps experienced by an agent in a single run.

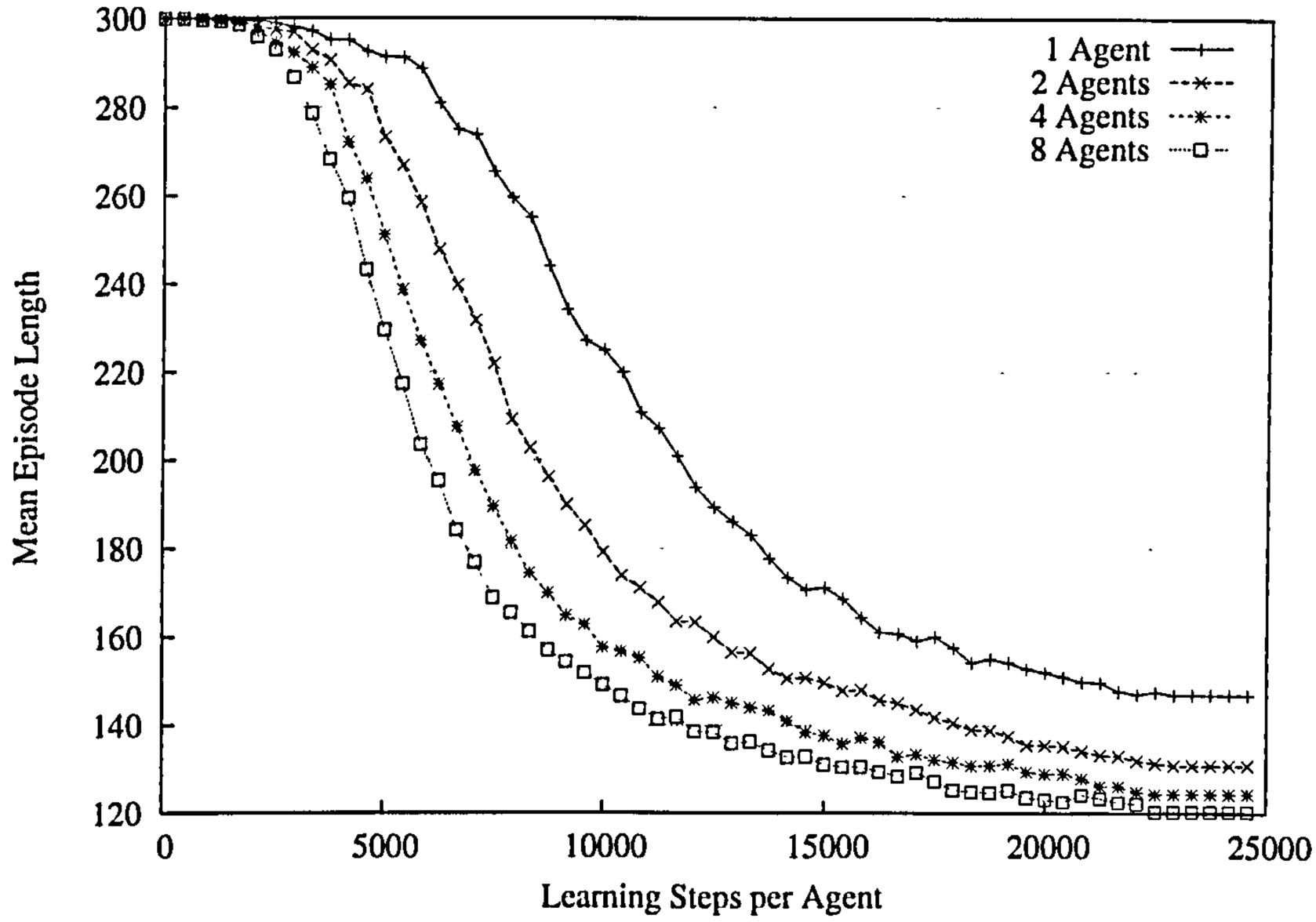


Figure 4.16: Results in the Mountain-Car task, decaying the values of α and ϵ linearly towards zero over 90% of the learning time.

Figure 4.16 shows the results of an experiment using the Mountain-Car task and the visit-count merge function. Reward function #1 is used, in addition to parameters $\gamma = 0.99$, $\lambda = 0.9$, $\theta_{init} = 0.0001$ and $p = 250$. Results are averaged over 200 runs. This makes the experimental settings the same as in Figure 4.12 except that α and ϵ now undergo linear decay, using parameters $\alpha_0 = 0.5$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. This means that in the last 10% of the experiment time the VFA is not modified, and so we can assess the quality of the learned policy without the effects of noise from exploration or learning updates. It can be observed in Figure 4.16 that the mean episode length at the end of the experiment is much better (for

all numbers of agents) than that achieved in Figure 4.12. Furthermore, the larger numbers of agents achieve a better quality of policy in the given time.

Eliminating the noise from learning at the end of the experiment now allows the policy quality to be properly assessed. In addition, by running experiments over longer periods and continuing to decay over 90% of the experiment time, the rate of linear decay of α and ϵ becomes slower. In a manner similar to that of simulated annealing, a slower decay of these parameters means that there is a greater chance over the decay time of settling into a policy which is very close to the optimum. Ideally, if we ran the experiment over a long enough time we would expect all groups of agents to have a high probability of finding the optimal VFA parameters. Unfortunately, the visit-count merge does not seem to behave in exactly this way.

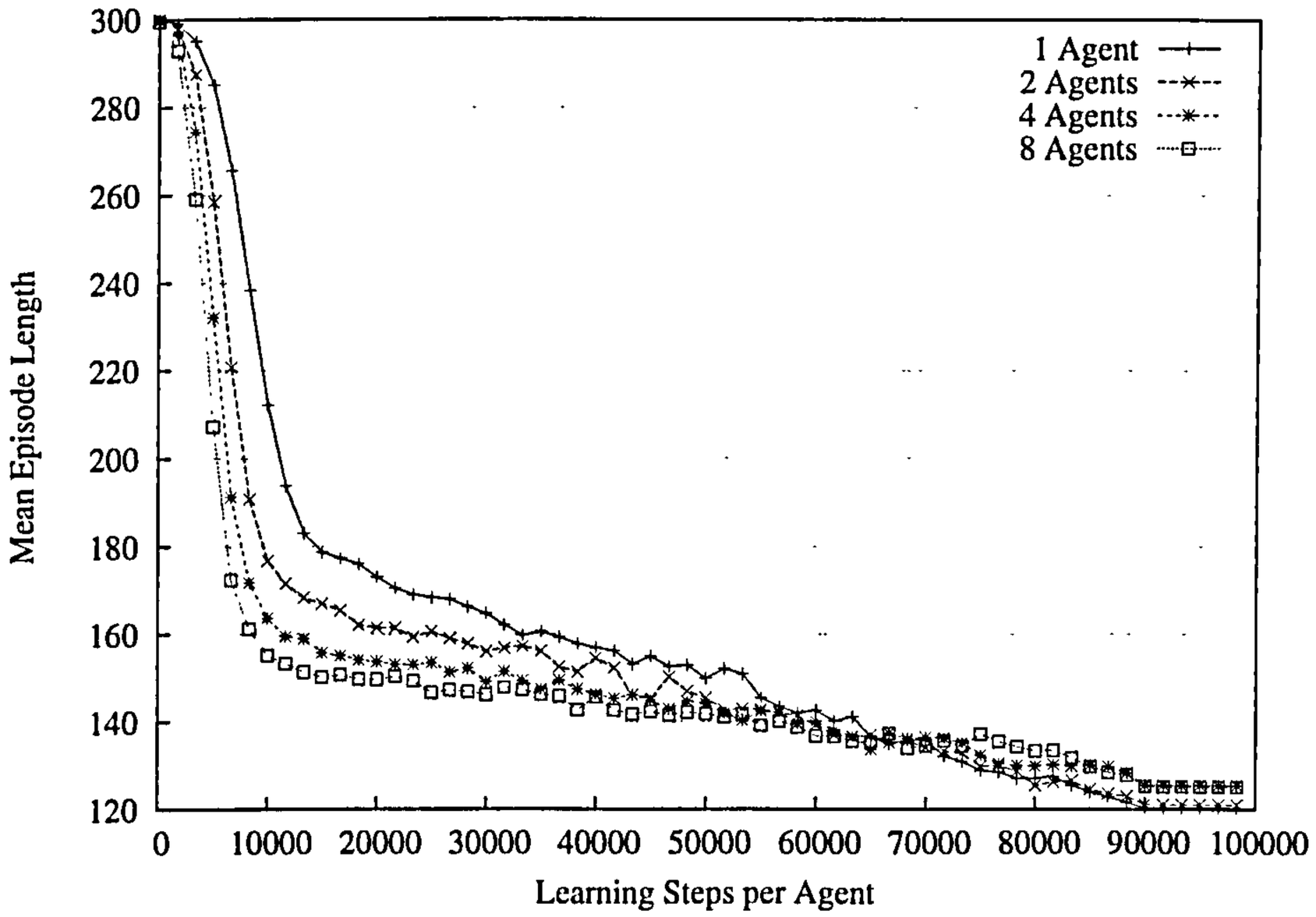


Figure 4.17: Results in the Mountain-Car task, where α and ϵ still decay linearly, but the experiment takes place over a longer period of time.

Figure 4.17 shows the results of an experiment identical to that shown in Figure 4.16 except that the length of the experiment is 100,000 steps rather than 25,000 steps. While the single agent learner and 2-agent group achieve a average final quality close to 120, both the 4 and 8 agent groups settle on a policy quality which is closer to 125. This is a trend which continues to occur even in much longer experiments. It seems that the *noise-reducing* effect of weighted-averaging discussed above has an adverse effect on the latter stages of learning. Learning in the mountain car problem can be divided into two distinct stages. The first stage

involves exploring the state space randomly until a reasonable path to the goal is found. This corresponds to the first 15,000 steps in Figure 4.17. The second stage consists of a gradual *refinement* of the best known path to the goal, by means of a decaying number of exploratory actions. The noise-reducing effect appears to reduce the probability of large changes to the VFA during the refinement stage. This means that in some cases the optimal VFA may not be found, since the learner may get stuck on an approximation which accurately estimates state value when ϵ has a small positive value but which produces sub-optimal performance once ϵ decays to zero.

Ignoring for the moment the fact that the single agent can get closer to the optimum in the long term, it will be useful at various points in this thesis to compare the time required for a given number of agents (using some algorithm) to achieve a particular quality of policy (for now restricting our attention to "good" policies rather than optimal ones). However, this task is complicated by the addition of decaying parameter values. Observe that in Figure 4.17 the group of 8 agents learn an average policy quality of 125 in 100,000 steps, but in Figure 4.16 the group can achieve an average quality closer to 120 in only 25,000 steps. This is because the shape of the graph in the "refinement" stage is predominately determined by the rate of decay of α and ϵ. The rate of decay must be slow enough to achieve the desired quality, but any slower and the learning time is essentially wasted. To properly compare the learning times for different numbers of agents, the rate of decay must be different for each group.

To achieve different rates of parameter decay, we will keep the proportion t_{lim} of the experiment over which the linear decay occurs *fixed* at 0.9. We will vary the experiment time independently for each group of agents. Since there is no analytical method for determining the minimum experiment time for each group, we will use *binary search* to find the minimum for each group within a specified tolerance. Before beginning the binary search, we determine the number of steps required for a single agent to achieve the specified policy quality. This is the initial *upper bound* on the experiment time. The initial *lower bound* for the search is 0. At each stage in the search, the mid-point between the upper and lower bounds is calculated, and an experiment is run using this as the experiment time. Each experiment started as part of the binary search uses a fixed number of agents n and reports the average quality achieved over r runs, where each run lasts for the specified experiment time. If the specified quality bound is achieved, the mid-point becomes the new upper bound. If it is not achieved the mid-point becomes the new lower bound.

Writing u and l for the upper and lower bounds, the binary search ends when $u - l < \delta l$, where δ is some level of tolerance which defines how accurately the minimum must be determined. Throughout this thesis a tolerance of 5% (i.e. $\delta = 0.05$) is used.

The results of the binary search approach for the Mountain-Car task using the visit-count merge function are shown in Figure 4.18. The settings for this experiment are identical to those used in Figure 4.16, except that the initial upper bound on the experiment length was set to 50,000 steps. The bound on average policy quality that the agents had to achieve was 130 steps per episode. These results show that the improvements achieved using merging fall some way short of linear speed-up, even in the absence of any communication costs. We can move closer to linear speed-up by reducing the merge period p to a smaller value than 250, but a very small merge period will not be feasible once communication has a realistic cost. The choice of merge period p is considered in more depth in Section 4.8.

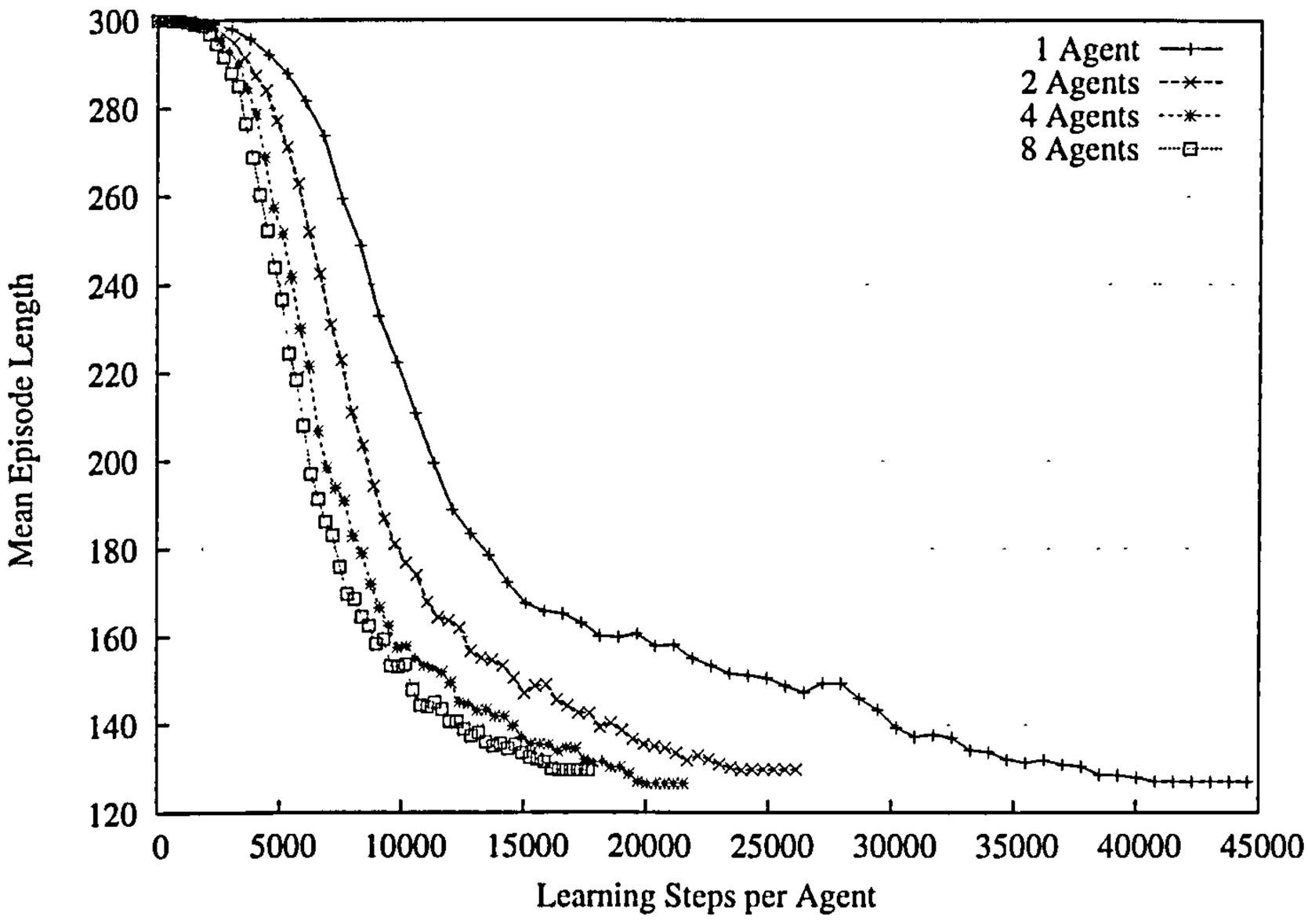


Figure 4.18: Results in the Mountain-Car task, where binary search is used to determine for each number of agents the shortest learning time that can still achieve a mean episode length of 130.

4.6 Examining Parallelism Without Merging

At this point it is worth asking the following question: to what extent is VFA merging a necessary (or even useful) component of this parallel RL approach? It may be the case that the best benefits of parallelism arise because, when there are several agents searching for a good policy, it is more likely that one of them will stumble across such a policy even without VFA merging. Alternatively, the use of VFA merging may be a vital component, allowing each agent to build upon intermediate results discovered by other agents.

To assess the benefits of communication and merging, it is useful to define a baseline algorithm which uses neither. The approach in the baseline algorithm is for the n agents to learn in parallel, but also in *isolation*. There is no communication between the agents. During a single run of the experiment, each of the agents learns in essentially the same manner as a single-agent learner. At the end of the run, each agent reports the mean episode length that was achieved in the final 10% of the experiment. At this point α and ϵ will have decayed to zero, and so the quality of the policy can be assessed without exploratory actions. The agent which achieves the highest quality is deemed to be the best, and we store the learning curve for this agent, discarding results from the others. Over a series of runs, at each time step we report the average of the performance achieved by the best agent in each of the runs. This baseline algorithm will be referred to as the BESTOF method. A major advantage of the BESTOF method is that its performance will be similar even if communication costs are extremely large.

Experiments with the BESTOF method using the domains defined in Section 4.3.1 have shown that its performance depends primarily on the character of the learning environment. A particularly important characteristic is the distribution of the learning curves achieved on different runs of a *single-agent* learner. Figure 4.19 is a boxplot graph illustrating the distribution for a single agent in the Mountain-Car task. The experimental settings were identical to those given for the experiment shown in Figure 4.16. The distribution was measured over 200 separate runs of the single agent. The distribution in Figure 4.19 shows that while most of the agents cluster around the mean performance (the quantity plotted in previous graphs), there are a number of outliers of both good and poor quality. Of the outliers of good quality, some achieve a near-optimal policy in as few as 15,000 time steps.

Agents learning using the BESTOF method behave essentially as single-agent learners, and so each will have a learning curve drawn from the distribution in Figure 4.19. However, the fact that we can choose the *best* agent at the end of

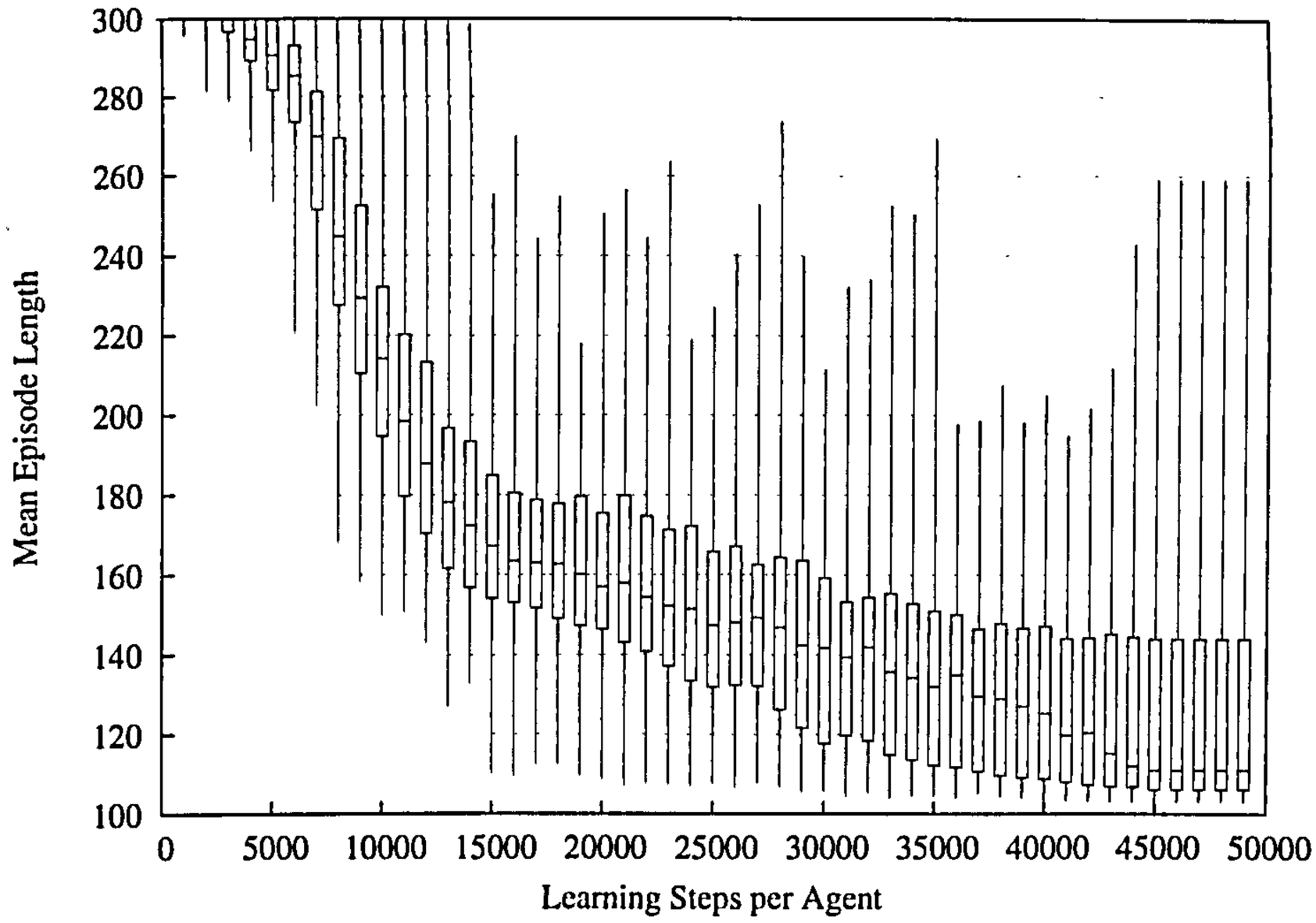


Figure 4.19: A graph showing the distribution of learning curves for a *single-agent learner* over a series of runs in the Mountain-Car task. The limits of the box represent the 25^{th} and 75^{th} quartiles, the line in the box represents the median, and the whiskers represent the maximum and minimum values.

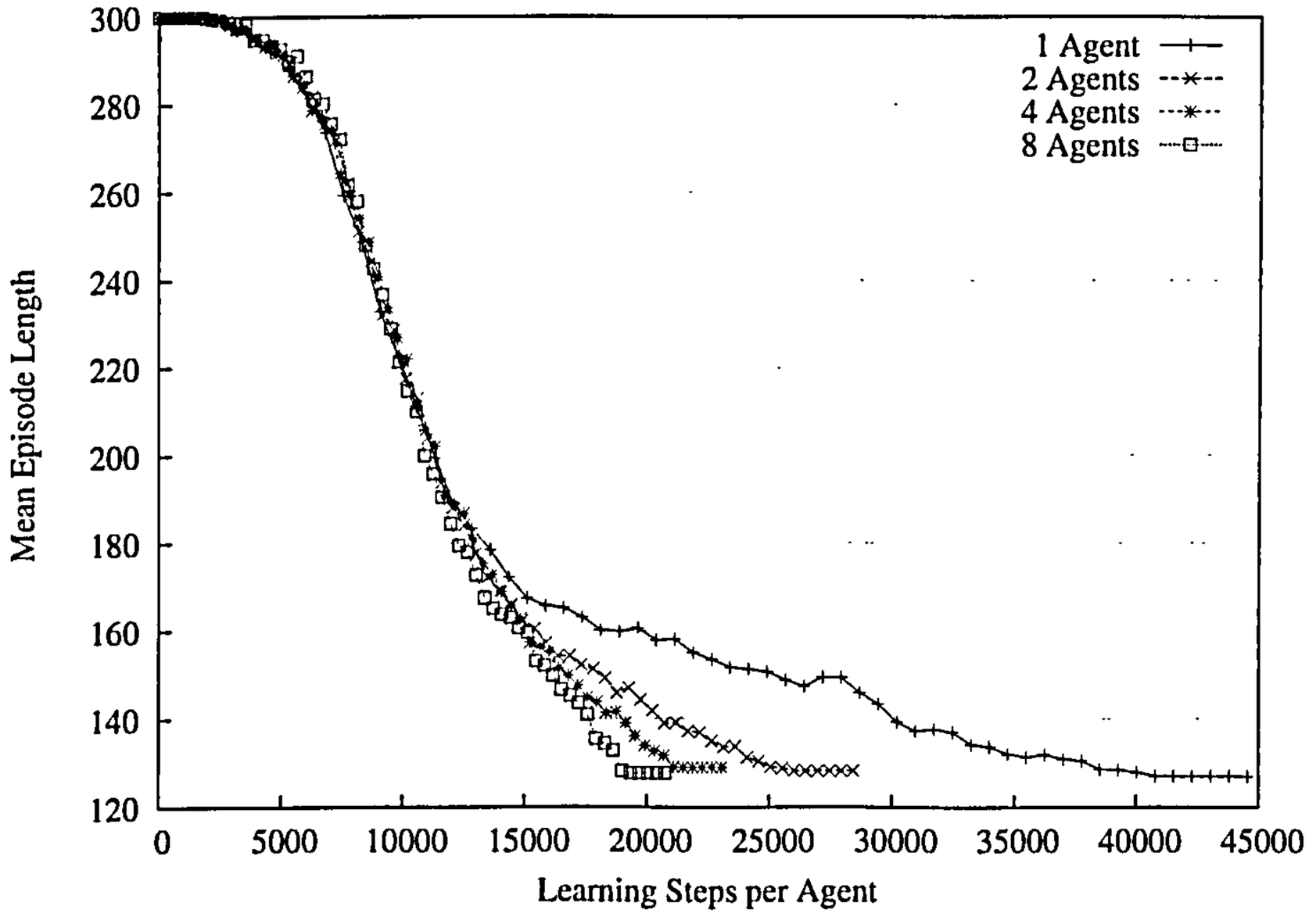


Figure 4.20: Performance of agents using the BESTOF method in the Mountain-Car task.

each run means that as the number of agents n is increased, the mean performance of the best agents approaches that of the outliers (see Figure 4.20). As the number of agents is increased, the group of agents will on average find a policy of a given quality in less time. These results were generated using a binary search to establish the minimum time for each group of agents to achieve on average a quality of 130 steps per episode. By comparing these results with those shown in Figure 4.18, we can observe that the resulting speedup is not much worse than those obtained using the visit-count merge, with the extra advantage that the BESTOF method remains practical even when communication costs are very high. However, it should also be clear from the distribution shown in Figure 4.19 that no matter how large we make the number of agents n, it is very unlikely we will find a good policy in fewer than 15,000 steps.

In some domains, however, the BESTOF method can perform particularly poorly. Figure 4.21 shows the results of the BESTOF method in the Stochastic Grid World domain. For purposes of comparison, Figure 4.22 shows results in the same domain using the visit-count merge function. The settings for these two experiments were as follows. The low-difficulty grid size was used. The decay of α and ϵ was defined by parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The other parameters were $\gamma = 0.99$ and $\lambda = 0.9$. Initial weight values[2] were set to $\theta_{init} = 1 \times 10^{-8}$. Reward function #2 was used, where the only non-zero reward is given when the goal is reached. Episodes were terminated after 10,000 time steps if the goal had not been reached. The visit-count merging experiment used a merge period of $p = 10000$ steps. Results were averaged over 10 runs.

Comparing the two graphs, it is clear that in this domain the BESTOF method can only achieve very small speedups. In contrast, the visit-count merging agents can achieve large speedups in this domain, since they are not limited by the performance of outliers in the distribution of single-agent learning-curves. This demonstrates that communication between the agents will be *vital* for achieving good parallel performance in many domains.

Small control problems such as the Mountain-Car and Pole-Balancing tasks have the property that once a reasonable policy is found, it can be very rapidly

[2]A small positive value for θ_{init} prevents the exploration behaviour from degenerating into a random walk (see Section 4.4.2). The value of θ_{init} in this experiment is a heuristic choice to facilitate rapid convergence to a short goal path. It is informed by the fact that the goal can be reached in under 120 steps from the initial state. When the goal is reached for the first time, the TD(λ) update in the initial state will be approximately $\alpha(\gamma\lambda)^{120}$, which is about 2×10^{-7}. Setting $\theta_{init} = 1 \times 10^{-8}$ ensures that this update will make actions which were used to reach the goal *immediately* appear more valuable. Therefore as soon as the goal is found for the first time, exploratory effort will be focused closely around this first (sub-optimal) path to the goal.

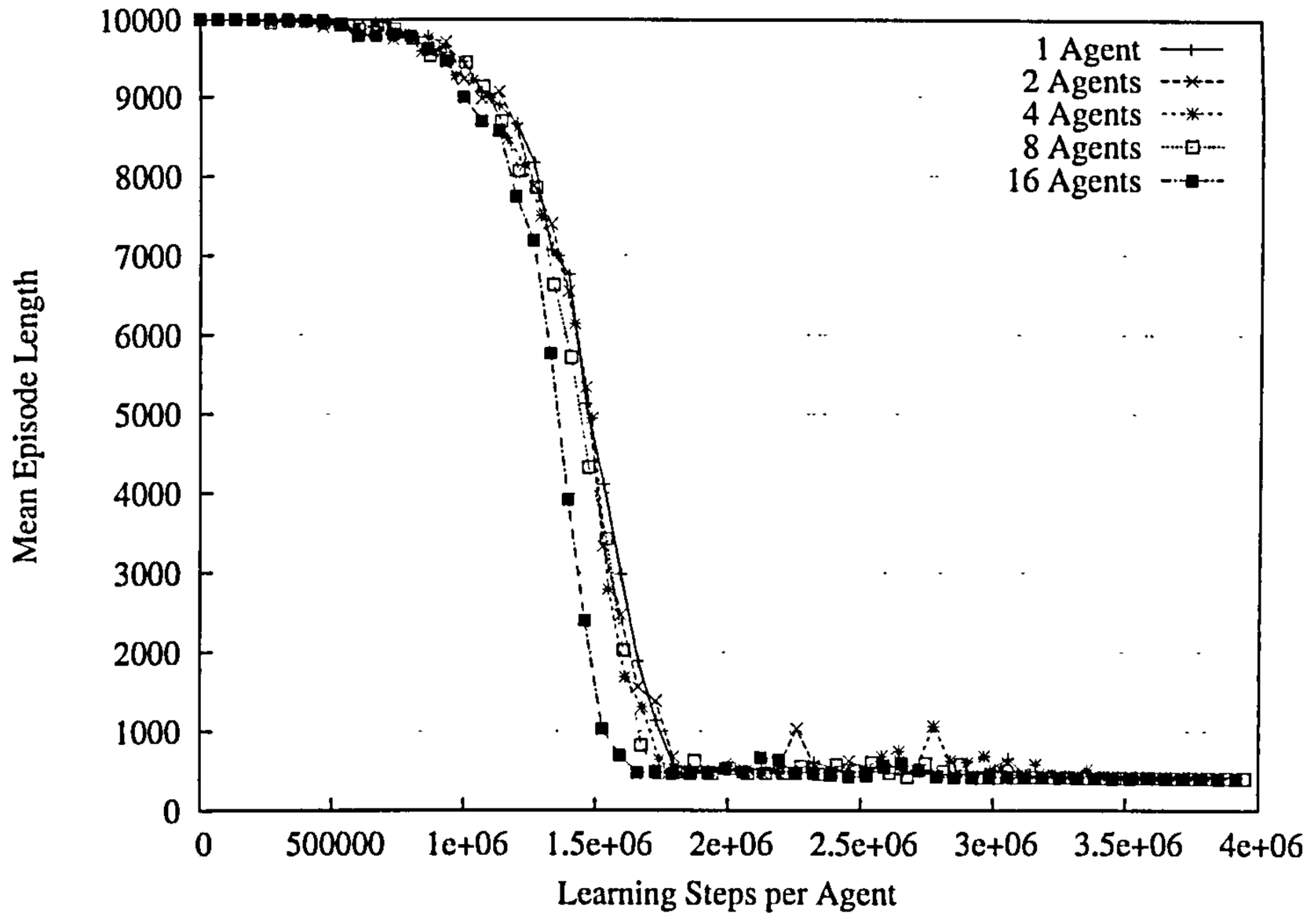


Figure 4.21: Performance of agents using the BESTOF method in the Stochastic Grid World task (low difficulty version).

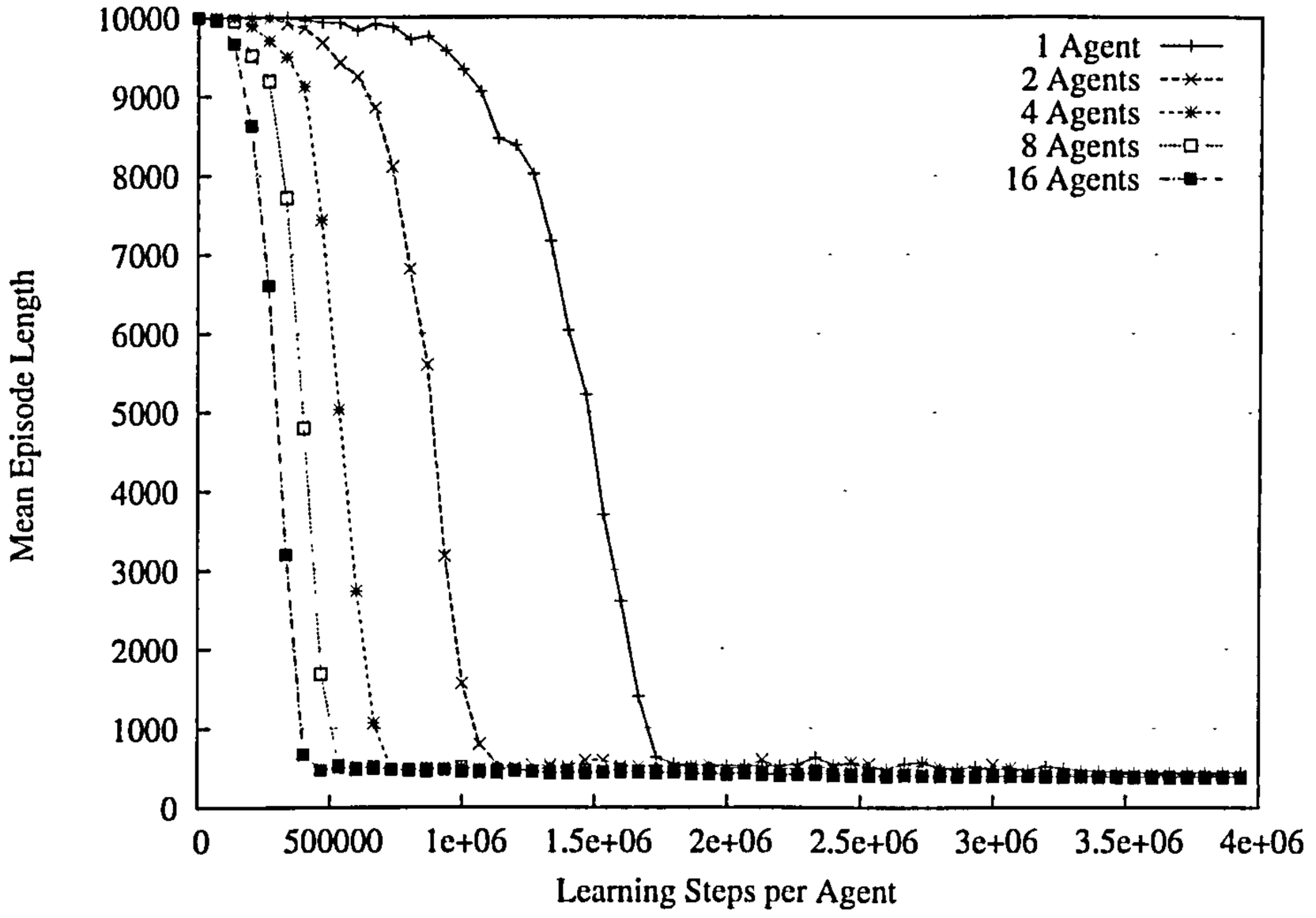


Figure 4.22: Performance of agents using the visit-count merge function in the Stochastic Grid World task (low difficulty version).

refined to approach the optimal policy. Learning time in these tasks is dominated by the period of unguided random exploration that continues until a reasonable policy is found by accident. Because of this property, the BESTOF method can achieve good speed-ups for small groups of agents. For larger numbers of agents, however, the improvements that can be obtained diminish very quickly as agents are added.

In the Stochastic Grid World Task it is extremely unlikely that a single agent will find a path to the goal completely by accident. Instead, the agent (through random exploration) gradually discovers which areas of the large state space contain poor rewards. Updates to the VFA means that these states are marked as having a low-value expectation, which leads the ϵ-greedy exploration strategy to focus on relatively unexplored areas of the state space. This means the agent can gradually eliminate possible locations of the goal until it is found. By merging the agents' approximations, the labour required for this process of elimination can be divided among the agents, greatly reducing the time required to find the goal and converge towards the optimal policy. This division of labour is not possible with the BESTOF method, since no communication can take place between the agents.

From these results, a general conclusion may be drawn that *communication* between the agents must play an important role if a parallel RL method is to be useful in a wide variety of domains. Otherwise the method will always be limited in performance by the properties of the distribution of single-agent learning-curves.

4.7 A True Parallel Implementation

In this section I will describe an evaluation of the VFA merging approach using an implementation on a real parallel system. The system used as the basis for this implementation was the cluster of workstations described in Section 4.3. The C++ implementation used the MPICH v1.2.5.2 library for the required communication operations. MPICH is an implementation of the *Message Passing Interface (MPI)* standard (Pacheco, 1997).

4.7.1 An Initial Implementation.

The initial implementation closely followed the architecture illustrated in Figure 4.2. In this original design, a central *manager agent* entirely separate from the learning agents took responsibility for the merging process. The manager would receive VFA weights from all the agents, calculate the merged VFA using the specified merge function f, and distribute the result to each of the learning agents before learning recommenced. However, assigning a dedicated parallel process to

the manager agent is somewhat wasteful of processing resources, since most of the time the manager is idle or waiting for messages. To avoid this, in the initial parallel implementation one of the learning agents takes responsibility for the manager's duties. Each of the n agents has a rank, a whole number between 0 and $n-1$ which uniquely identifies the agent. The agent with rank 0 always performs the functions of the manager at merge time.

We restrict our attention in this section to merging with the *visit-count* merge function, since this was shown in Section 4.4 to reliably produce the best performance out of the considered merge functions in *all* the evaluation domains. For agent 0 to calculate the visit-count merge function, all agents with rank$\neq 0$ must send a vector of approximator weights and a vector of visit counts to agent 0. After the merged VFA has been calculated, agent 0 must send a vector of merged approximator weights to all the other agents. A Gantt chart illustrating this process is shown in Figure 4.23.

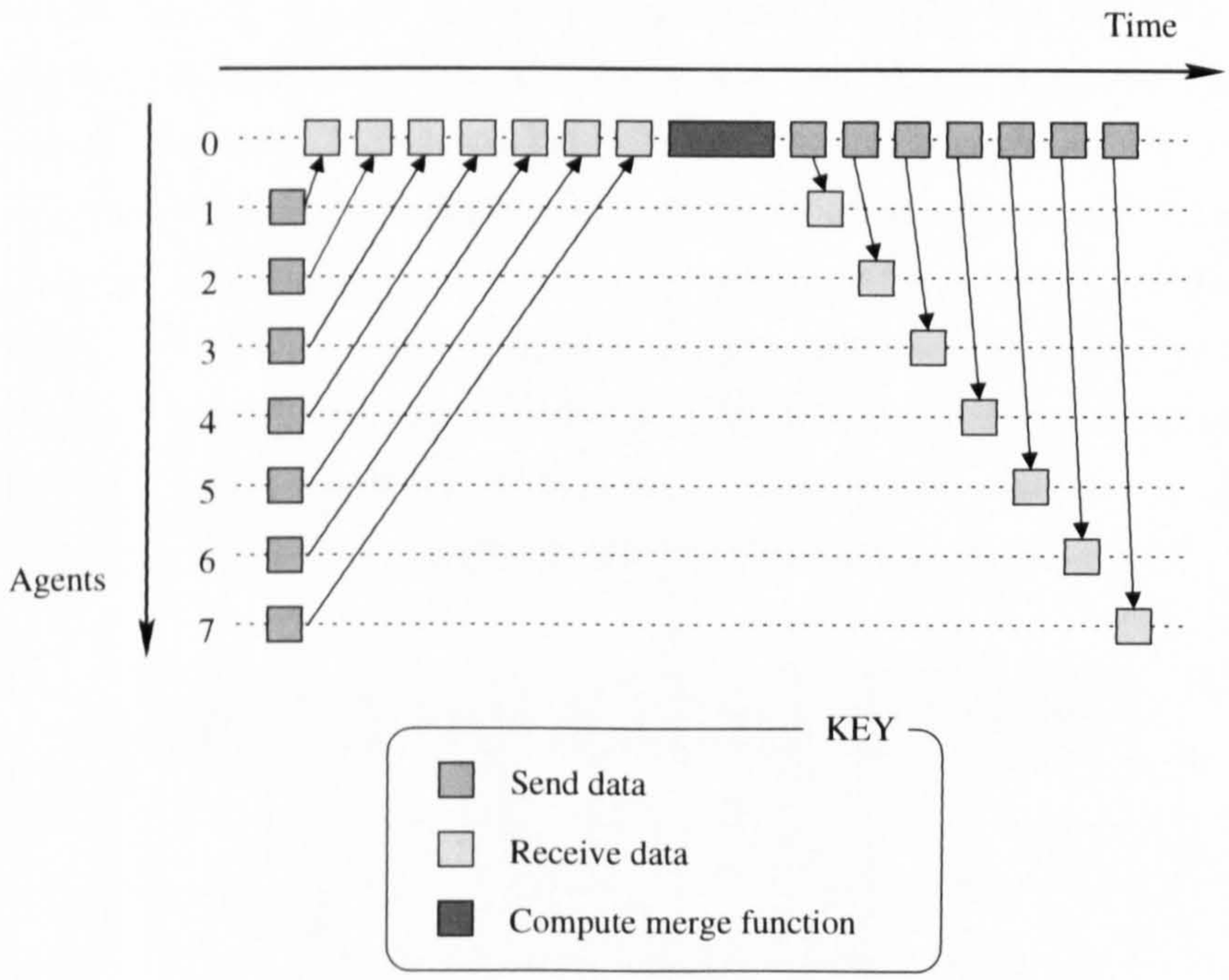


Figure 4.23: Gantt chart illustrating the exchange of messages required to complete each merging operation in the initial parallel implementation.

Messages are exchanged between agents using the `MPI_Send` and `MPI_Recv` library functions provided by MPICH. These functions represent the most basic point-to-point communication mechanism provided by the MPICH library. These functions can be used to send arbitrary length vectors of data values from one agent to a *single* destination agent. The type of the data values may be any of the C++ primitive types. In our implementation, the approximator weights are rep-

resented as 32-bit floating point numbers, and the visit counts are represented as 32-bit integers. The agents use this representation for both internal representation (used during learning) and the content of messages exchanged. The `MPI_Send` and `MPI_Recv` functions are *blocking* calls. `MPI_Send` will only return once a matching call to `MPI_Recv` has been started at the destination agent, and only when the message data has been safely copied out of user memory. `MPI_Recv` will only return once the full message from the source agent has been successfully received. MPICH also provides *non-blocking* functions for point-to-point communication, which will be used later in Chapters 5 and 6.

Preliminary experiments with this first implementation revealed that the length of time required for a merge operation was growing too quickly as the number of parallel agents was increased. It was not possible to achieve significant speed-ups in *any* of the evaluation domains. To investigate this effect, we used logging functionality in the MPICH library to collect precise timings for the series of merge periods performed in a single run of the algorithm. The timings reported in Table 4.3 were generated in the low-difficulty Stochastic Grid World task. A merge period of $p = 10000$ was used. In contrast to the feature set described in Section 4.3.1, this particular set of experiments used 4 tilings of dimension 32x32. There are 4 actions, so this makes the total number of approximator features 16,384. Given that 4 bytes are required to store either a single weight or a single visit count, each message sent *to* agent 0 is approximately 128KB in size, and each message sent *from* agent 0 is approximately 64KB in size. Note that the errors reported here are the *range* of the timings observed, not the standard deviation.

Number of Agents	Time for Merge Operation (ms)
2	33±2
4	82±2
6	131±2
8	181±3
10	233±5
12	280±2
14	332±3
16	381±3

Table 4.3: Timings for the visit-count merge operation in the Stochastic Grid World task using the initial parallel implementation.

Each of the timings is measured from the time at which the first agent tries to

send its message to agent 0 to the time at which the last of the agents receives the merged weights from agent 0. From the measurements given in Table 4.3 we can observe that the growth in the merging time is close to $O(n)$, where n is the number of agents. The timings are dominated by the growth of communication costs. Each period of learning in the simulation (between the merges) lasts consistently around 30ms. The time for agent 0 to calculate the merged weights varies approximately linearly, but only takes 15ms even when there are 16 agents. It is clear that the parallel RL method will not be practical on real parallel hardware if the cost of communicating grows at such a rate.

4.7.2 Distributed Computation of the Merge Function

To reduce the effect of growing communication costs on our algorithm, it is necessary to abandon the notion of a *manager agent* which performs the computation required to calculate the merged VFA. While this remains a useful conceptualization of the method, a practical parallel implementation must spread the workload more equally among the available agents, instead of assigning agent 0 to do all the work.

Recall from Section 4.4.4 the form of the visit-count merge function f:

$$f(\theta_{1,i}, ..., \theta_{n,i}, c_{1,i}, ..., c_{n,i}) = \frac{\sum_{j=1}^{n} c_{j,i}\theta_{j,i}}{\sum_{j=1}^{n} c_{j,i}}$$

Each of the two summations in this function can be calculated using a *distributed computation.* To understand this computation, it will be useful to consider a couple of simple examples. Suppose we have n agents, where each agent i has stored in local memory a vector $\vec{v_i}$. All the vectors are of the same dimension d. Suppose also that we want to calculate the vector sum $\vec{s} = \sum_i \vec{v_i}$ and store the result in the memory of agent 0. If d is large, both communicating the vectors and calculating the sum will be expensive operations. The naive approach of sending all the vectors to agent 0 and then adding them together produces a major bottleneck at agent 0. It is more efficient to arrange the summation using a tree structure, as shown in Figure 4.24.

The example in Figure 4.24 results in the same number of messages (seven) being sent over the network as if all the agents had sent their vectors directly to agent 0. However, because the messages sent during each stage have different destinations, they can travel *simultaneously* over the switched Ethernet network. This removes the communication bottleneck at agent 0. In addition, the computation of the vector sum is shared more evenly among the agents, and parts of the computation in the same stage can be performed *in parallel.* Each of the larger

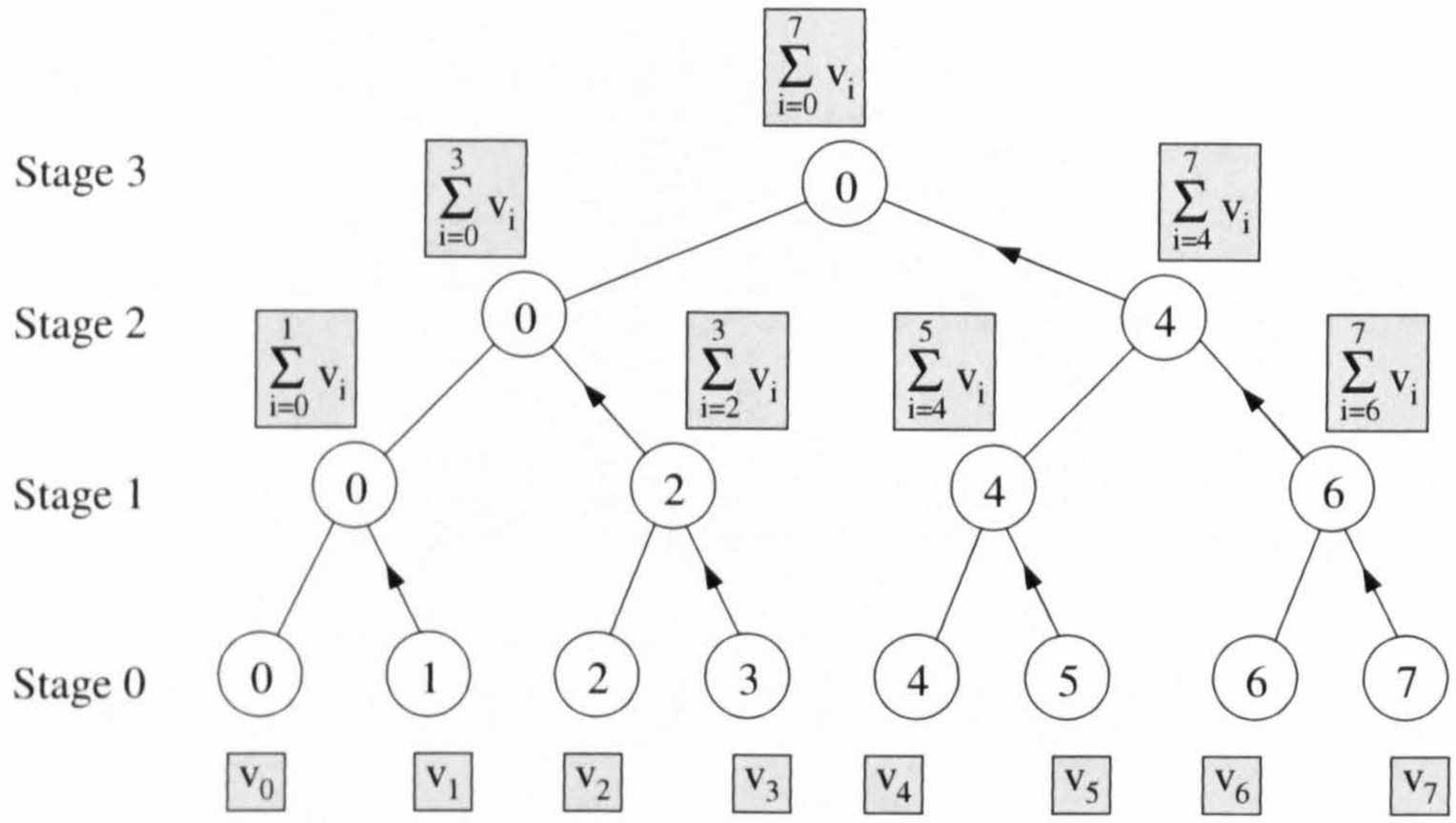


Figure 4.24: Agents (represented as circles) performing a distributed computation of a sum of vectors. The arrows indicate data communicated between two agents, and the grey boxes show which parts of the computation are carried out by each agent.

grey boxes in Figure 4.24 represents one of the agents adding the vector received over the network to its own local vector. If the number n of vectors to be added is a power of two, then n agents can perform the addition in $\log_2 n$ stages. The time required for each stage is $t_1 + t_2$, where t_1 is the time required to send a vector between two agents over the network, and t_2 is the time required for a single agent to add two vectors together.

Suppose now that all agents need to know the result of the summation once it is completed (as is the case for our parallel RL method.) One way to achieve this would be to compute the sum as shown above, then have agent 0 broadcast the result to all the other agents. The broadcast can also be implemented using a tree structure in a similar way, by reversing the direction of the communication arrows in Figure 4.24. However, a more efficient way of achieving the same effect is to use the communication structure shown in Figure 4.25. This type of communication structure is sometimes known as a *butterfly*. It can be interpreted as a set of trees, one rooted at each agent, with the common subtrees combined together to produce a *directed acyclic graph (DAG)*. Since the communications within each of the stages can be performed simultaneously, this operation can be completed in the same number of stages ($\log_2 n$) as the operation shown in Figure 4.24.

This discussion has thus far relied on the assumption that n is a power of two. If n is not a power of two then additional communications are needed to ensure that all the agents end up with the correct sum. In such cases the required

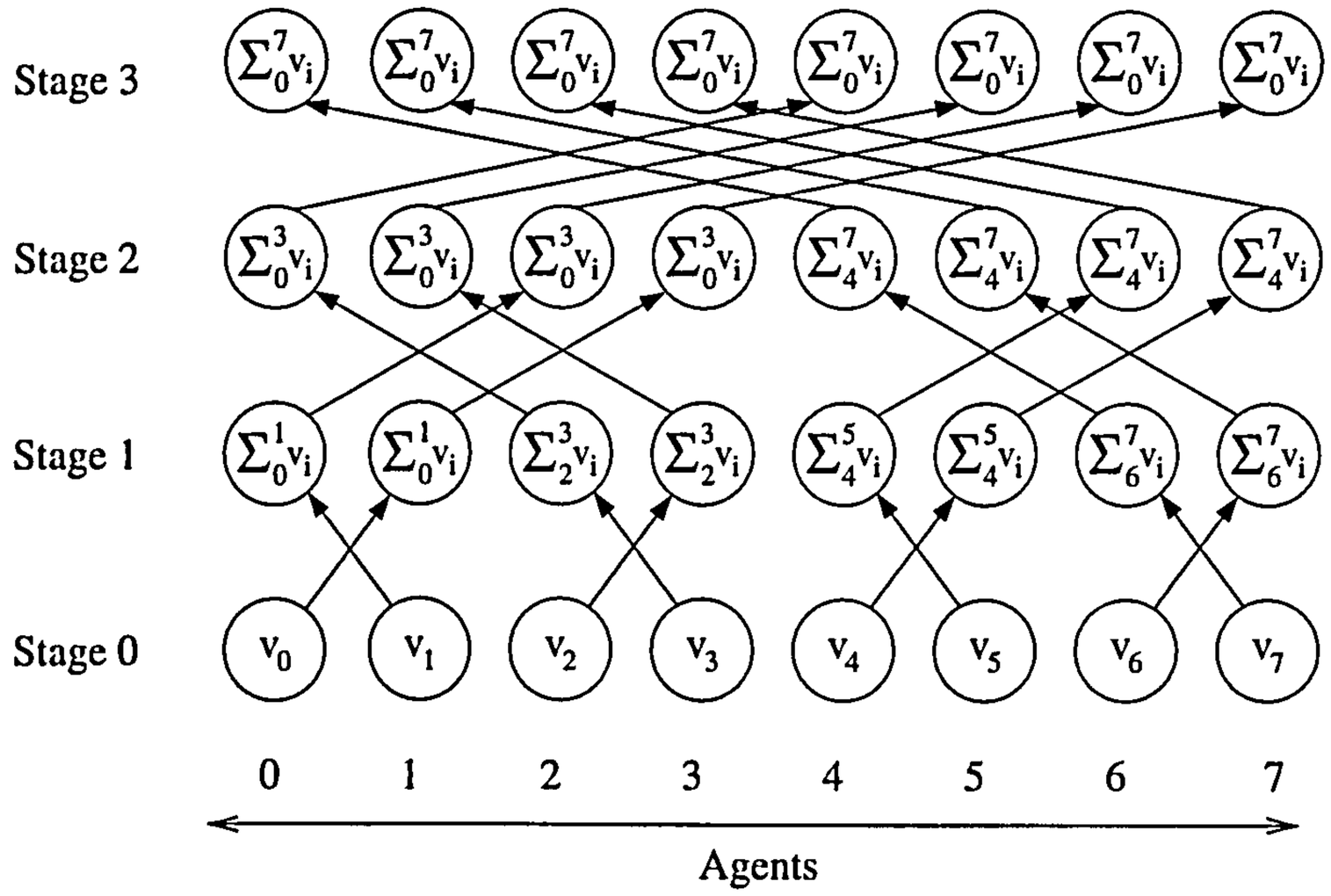


Figure 4.25: A distributed sum of vectors where the result is required by *all* the participating agents. Intermediate results at each stage are shown in a circle for every agent.

number of stages has an upper bound of $2\lfloor \log_2 n \rfloor$. This approach is known as a *recursive doubling* algorithm. For further details on the use of the recursive doubling algorithm in MPICH, the reader should refer to Benson et al. (2003).

The MPICH library provides implementations of the distributed summations in Figures 4.24 and 4.25 with the functions `MPI_Reduce` and `MPI_Allreduce` respectively. The two summations in the visit-count merge functions will now be implemented using the `MPI_Allreduce` function. The pseudocode for the algorithm followed by each of the agents is shown in Algorithm 3.

A new series of timings for the merge operation was collected for this new parallel implementation. They are shown in Table 4.4, where the errors again reflect the *range* of the timings rather than the standard deviation. The settings used were identical to those described at the beginning of this section, except that this new implementation was used. The results show with this new implementation, as the number of agents n is increased the growth of communication costs is $O(\log n)$ rather than $O(n)$. Note that when n is not a power of two additional stages of communication are required, which is why the groups of 10, 12 and 14 agents take longer to merge than the 16 agent group.

Still using this new implementation, two further sets of timings were collected. The purpose of these further timings were to investigate the effect of the number of approximator features on the time required for merging. The experiments used for

Algorithm 3 Agent pseudocode for the improved parallel implementation (based on the visit-count merge function.)

{Initialization}
for all i **do**
  $\theta_i \leftarrow \theta_{init}$
  $c_i \leftarrow 0$
end for

{Main loop}
while time elapsed $< t_{end}$ **do**
  for *step* $= 1$ to p **do**
    Execute a simulation step.
    Update weight vector $\vec{\theta}$
    For each active feature ϕ_i increment the visit-count c_i.
  end for

$\vec{n} \leftarrow$ `MPI_Allreduce`$\left(\vec{c}.\vec{\theta}, \texttt{MPI\_SUM}\right)$ {Parallel summation}
$\vec{d} \leftarrow$ `MPI_Allreduce`$\left(\vec{c}, \texttt{MPI\_SUM}\right)$ {Parallel summation}

  for all i **do**
    if $d_i \neq 0$ **then**
      $\theta_i \leftarrow n_i/d_i$
      $c_i \leftarrow 0$
    end if
  end for
end while

Number of Agents	Time for Merge Operation (ms)
2	25±2
4	51±5
6	92±5
8	80±15
10	119±3
12	105±5
14	120±10
16	100±10

Table 4.4: Timings for the visit-count merge operation in the Stochastic Grid World task using the improved parallel implementation.

Number of Agents	Time for Merge Operation (ms)
2	11±1
4	21±3
8	30±2
16	43±5

Table 4.5: Timings for the improved implementation in the Stochastic Grid World task, using half the previous number of features.

Number of Agents	Time for Merge Operation (ms)
2	53±3
4	102±5
8	149±2
16	198±7

Table 4.6: Timings for the improved implementation in the Stochastic Grid World task, using twice the previous number of features.

the timings in Tables 4.3 and 4.4 both used 4 tilings of size 32x32 for each of the 4 actions. The timings shown in Table 4.5 were collected using 2 tilings of the same size (halving the total number of features). The timings shown in Table 4.6 were collected using 8 tilings of the same size (doubling the total number of features). These two sets of data provide some evidence that as the number of features f is increased, the growth of communication costs is $O(f)$. This makes sense because doubling the number of features means that the number of weights and visit counts is doubled, so every single message that needs to be communicated between the agents becomes twice as large.

4.7.3 Experiments using the Improved Parallel Implementation

While the above improvements to the implementation achieved a slow asymptotic growth of communication costs, there remains a large constant factor not reflected by the use of "big O" notation. The domains used for evaluation in this work use between 2000 and 20,000 weights in each VFA. Each agent also maintains and communicates a similar number of visit-count values. Communicating large vectors of weights (and visit-counts) between the agents results in a large delay while the data travels over the intercommunication network.

In the experiments carried out using the cluster of workstations, it was found that the visit-count merging approach could not achieve a parallel speed-up in any of the simple control problems considered (i.e. the Mountain-Car task, the Pole-Balancing task and the Acrobot task). Figure 4.26 shows one set of results collected for 2 agents learning in the Mountain-Car task. A range of different values were tried for the merge period p. Reward function #2 was used, and α and ϵ were decayed linearly using parameters $\alpha_0 = 0.5$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The other parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 0.0001$. Results were averaged over 200 runs.

Since each agent only has time to simulate about 60,000 steps within the 0.35 second time limit for this experiment, the performance of the two agents using $p = 150,000$ is essentially the same as that of a single-agent learner (i.e. no merging operations occur within the duration of the experiment for these agents.) The results in Figure 4.26 show that, regardless of the value we pick for the merge period p, two agents cannot significantly speed-up learning in the Mountain-Car task using the improved parallel implementation. In fact, when $p < 1000$ performance in this domain is significantly *worse*. This is because any improvement in convergence speed achieved by the agents sharing intermediate results is cancelled out by the extra time spent sending, waiting for and receiving messages. Similar results were obtained for the Pole-Balancing and Acrobot tasks, and for greater numbers of

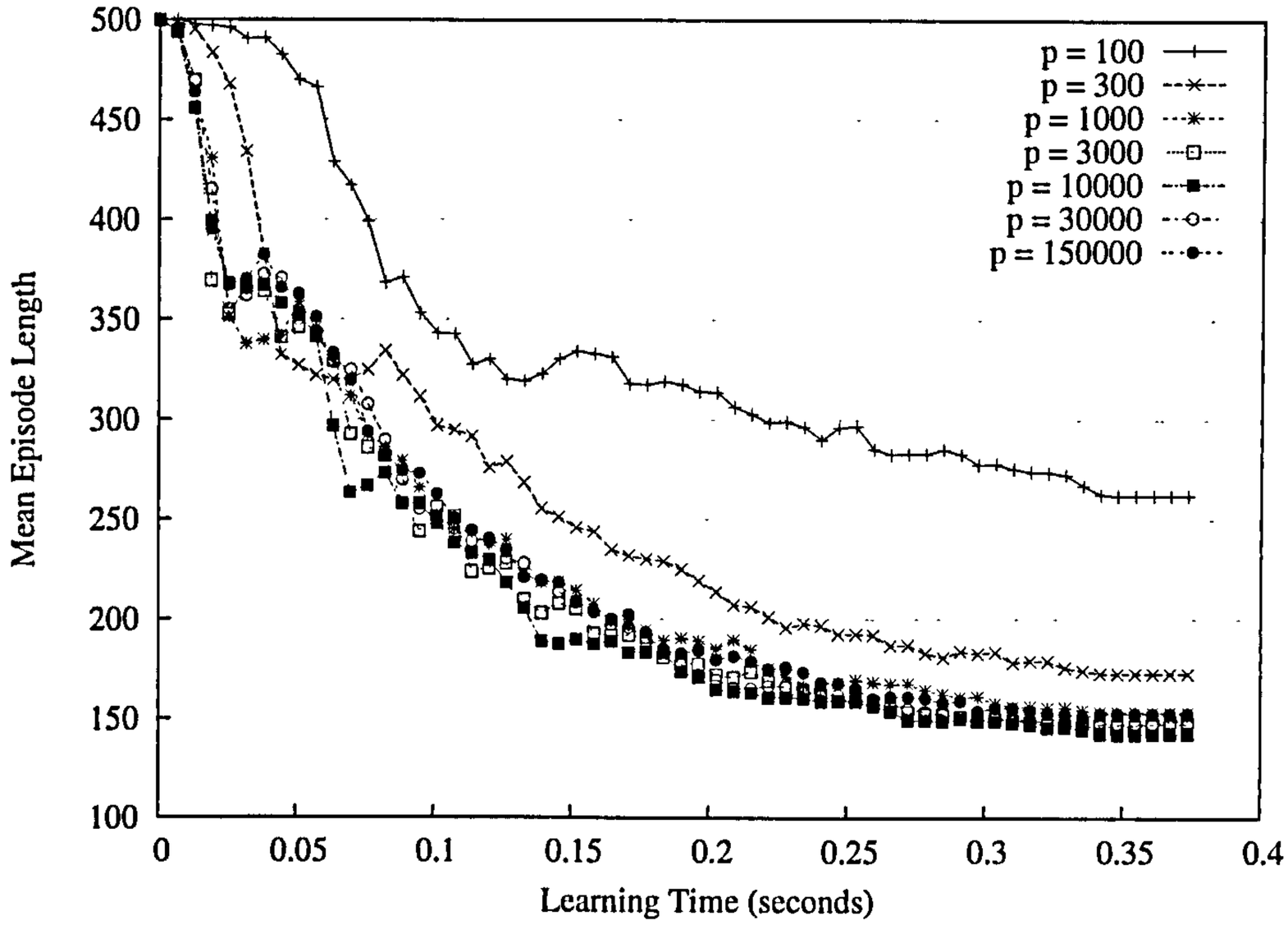


Figure 4.26: Learning curves generated with the cluster of workstations for 2 agents in the Mountain-Car task. Each curve uses a different value for the merge period p.

agents in these domains.

In the Stochastic Grid World task, however, it was possible to achieve learning speed-ups using the cluster of workstations. Results for the low-difficulty instance of the grid world are shown in Figure 4.27. Reward function #2 was used, in addition to parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$, $t_{lim} = 0.9$, $\gamma = 0.99$, $\lambda = 0.9$, $p = 10,000$ and $\theta_{init} = 1 \times 10^{-8}$. Results were averaged over 10 runs (the variation in performance on different runs is small compared to the Mountain-Car task).

The results in Figure 4.27 show that a reasonable parallel speedup can be achieved with different numbers of agents. However, the speedup falls some way short of the ideal case of *linear* speedup. Comparing this graph with the one in Figure 4.22 (an experiment with similar settings using the *simulation* of parallel agents) allows us to assess the degree to which realistic communication costs degrade the performance measured in simulation.

The relatively large speedups achieved by the groups of 2 and 4 agents remain significant on the cluster of workstations, outpacing the logarithmic growth in communication costs. However, in the simulation we observed diminishing returns as the number of agents was scaled up to 8 and 16. As the speedup effect of merging is diminishing, the growth of communication costs continues at a similar rate. This results in a very small parallel speedup moving from 4 to 8 agents, and a

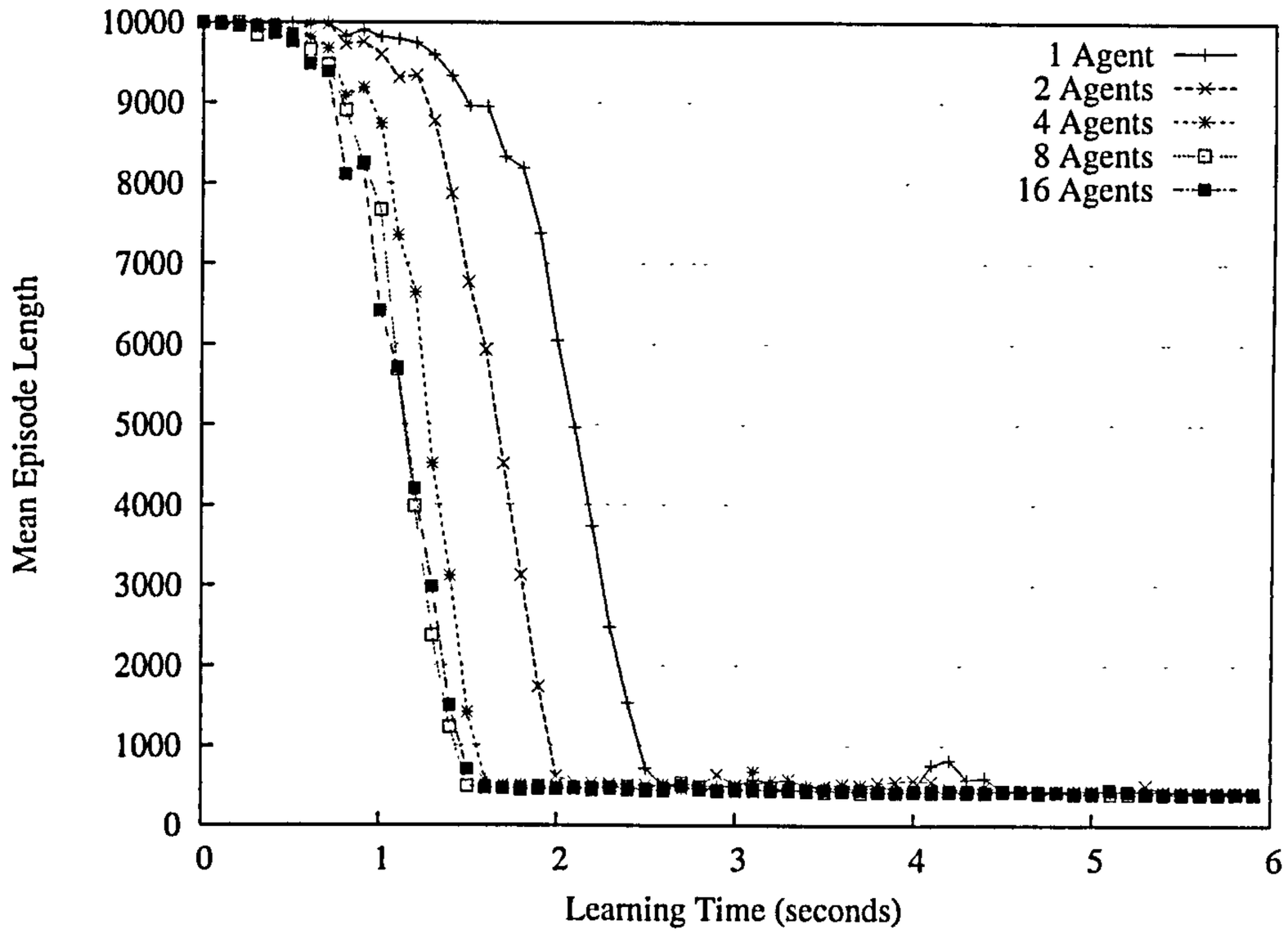


Figure 4.27: Performance of the visit-count merge method on the cluster using the (low-difficulty) Stochastic Grid World task.

very similar performance as we move from 8 to 16 agents. With the implementation in its current state, it is only useful to scale-up to a maximum of 8 agents.

Parallel speedups were also achieved using the more difficult instance of the Stochastic Grid World task. This (high-difficulty) instance of the task requires on average about twice as many simulation steps to reach the goal from the initial state, uses a greater number of features in the VFA and requires an order of magnitude more real time to learn a near-optimal policy. Results for the high-difficulty instance are shown in Figure 4.28. Reward function #1 was used, in addition to parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$, $t_{lim} = 0.9$, $\gamma = 1.0$, $\lambda = 0.95$, $p = 50,000$ and $\theta_{init} = 0$. Results were again averaged over 10 runs.

A similar pattern of speedups is achieved in this experiment, although overall the results are slightly worse. 2 agents converge to a near-optimal policy in about 20% less time than a single agent, which is obviously some way short of the 50% that would be necessary for linear speedup. 4 agents improve over the performance of 2 agents, but not by much. Groups of 8 and 16 agents seem to learn slightly faster in the early stages of the experiment, but can only converge completely to a near-optimal policy in the time required by 4 agents.

The results presented in Figures 4.27 and 4.28 show that agents implemented using distributed-memory parallel hardware *can* learn good solutions to RL problems more quickly than a single-agent learner. However, using our particular

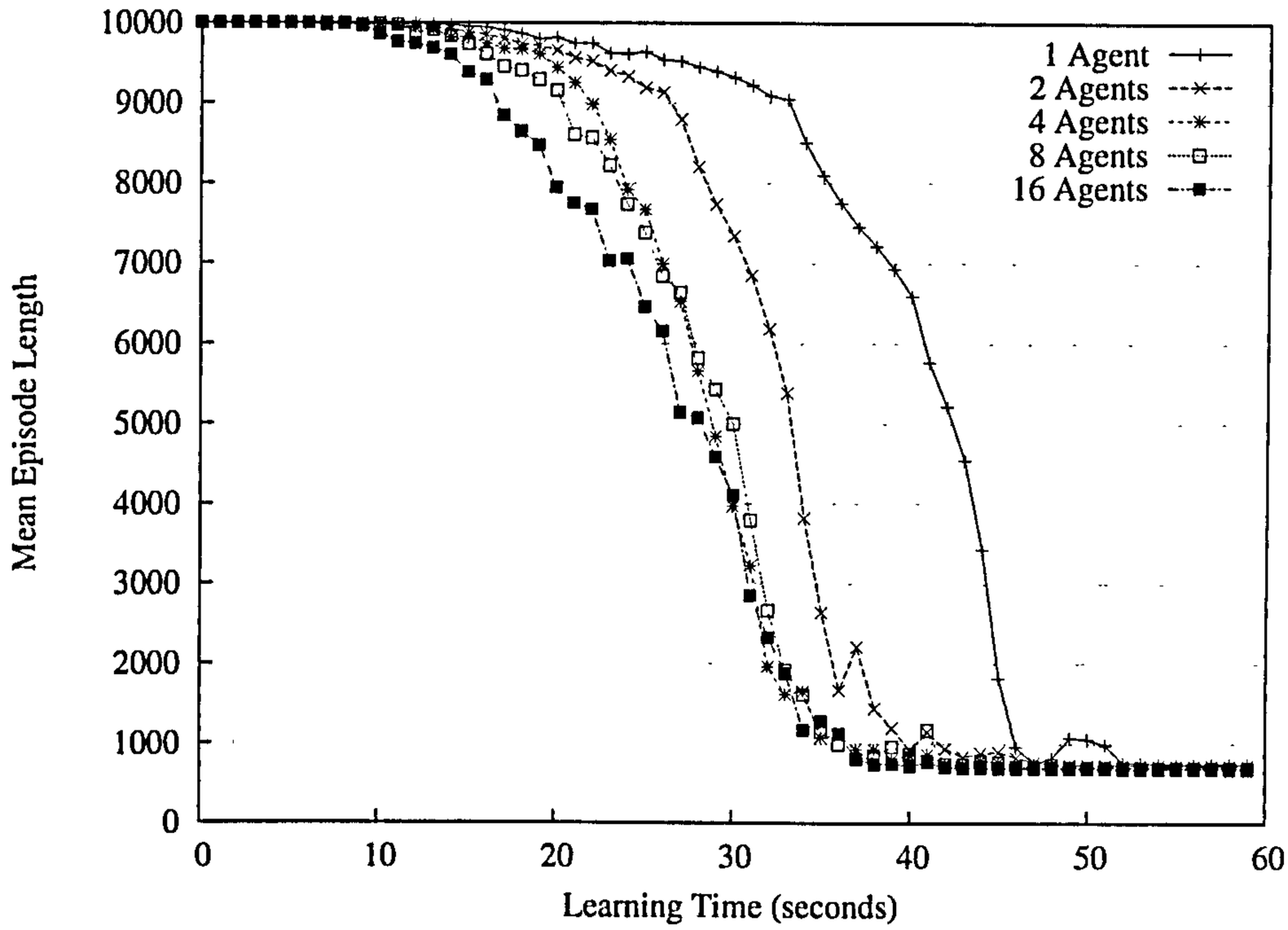


Figure 4.28: Performance of the visit-count merge method on the cluster using the (high-difficulty) Stochastic Grid World task.

implementation with the hardware resources available resulted in a high cost of communication relative to the total learning time. This meant that where parallel speedups were achieved, they were only significant for groups of 2 or 4 agents, and for many domains not even this could be achieved (see Figure 4.26). These results suggest that to develop a parallel RL method which is practical to use with a cluster a workstations, the efficient use of network bandwidth must be emphasised to a much greater degree.

4.8 The Influence of the Merge Period

In the experiments discussed so far in this chapter, the value used for the merge period p in each domain has been specified, but no justification has been provided for the particular values chosen. In this section I will examine more closely the influence of the merge period on the overall performance of the visit-count merge method, and address the problem of selecting a suitable value for p.

The merge period controls how often the agents are able to share value function information. A smaller merge period means that information exchanges are more frequent. Intuitively, the more frequent the exchange of information, the greater the probability that an agent can exploit information discovered by one of its peers *before* the agent discovers the same information for itself. In other words, frequent

information exchange reduces the level of duplicated effort.

For example, consider the low-difficulty version of the Stochastic Grid World task. The graph in Figure 4.29 plots learning curves for 2 *simulated* parallel agents using a variety of values for the merge period p. Since the curves were generated using the simulation of parallel agents, the communication time required to merge the VFAs is not included in these results. The curve for $p = 2,500,000$ is essentially identical to that of a single-agent learner, since the results are only plotted for the first 2,000,000 simulation steps. The full settings for this experiment were as follows. Reward function #2 was used, and results were averaged over 50 runs. The experiment lasted for 4,000,000 simulation steps, with α and ϵ decaying linearly to zero over 90% of this time. The other parameters used were $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$, $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 1 \times 10^{-8}$.

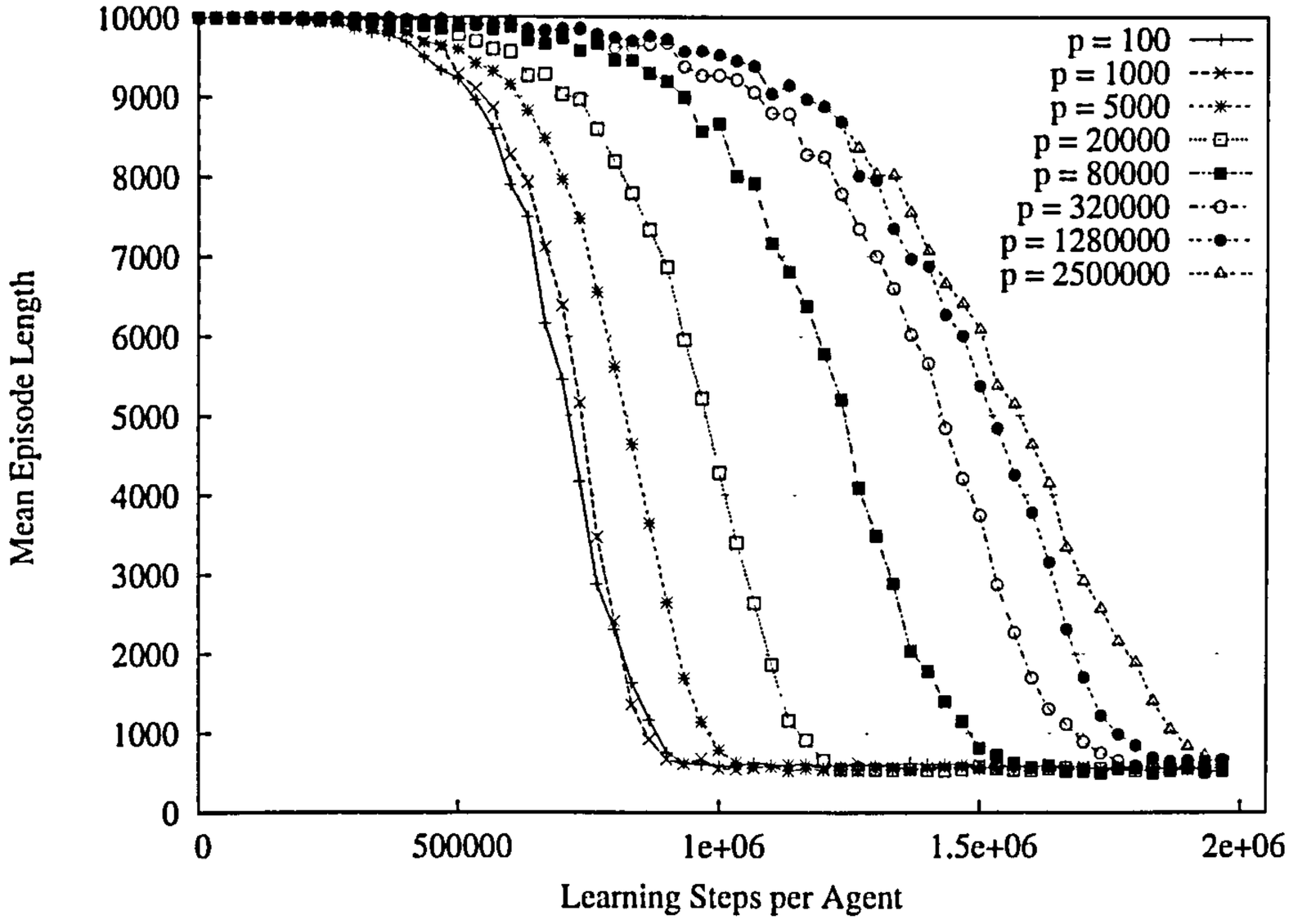


Figure 4.29: Varying the merge period p for *2 agents* in the low-difficulty Stochastic Grid World, using the *simulation* of parallel agents.

As progressively smaller values of p are used, the number of simulation steps required for the agents to converge is gradually reduced, improving significantly over the performance of a single agent. The fastest convergence which can be obtained seems to be limited to about half the steps required by the single-agent, which would correspond to a *linear speedup* for 2 agents. Note that in Figure 4.29 the curves for $p = 100$ and $p = 1000$ both converge very close to this limit. There is therefore little incentive to use a merge period smaller than 1000 in this domain, since it will not improve the performance any further.

We will now consider an experiment using almost exactly the same settings on the cluster of workstations. The main difference between the two experiments is that on the cluster, each run finishes after 6.0 seconds of real time instead of after 4×10^6 simulation steps. This means that the communication penalty incurred by the merge operation *does* now have an effect on the results, shown in Figure 4.30. The full extent of this penalty for a given merge period is shown in Table 4.7, which reports the percentage of the total experiment time consumed by the distributed calculation of merged value functions.

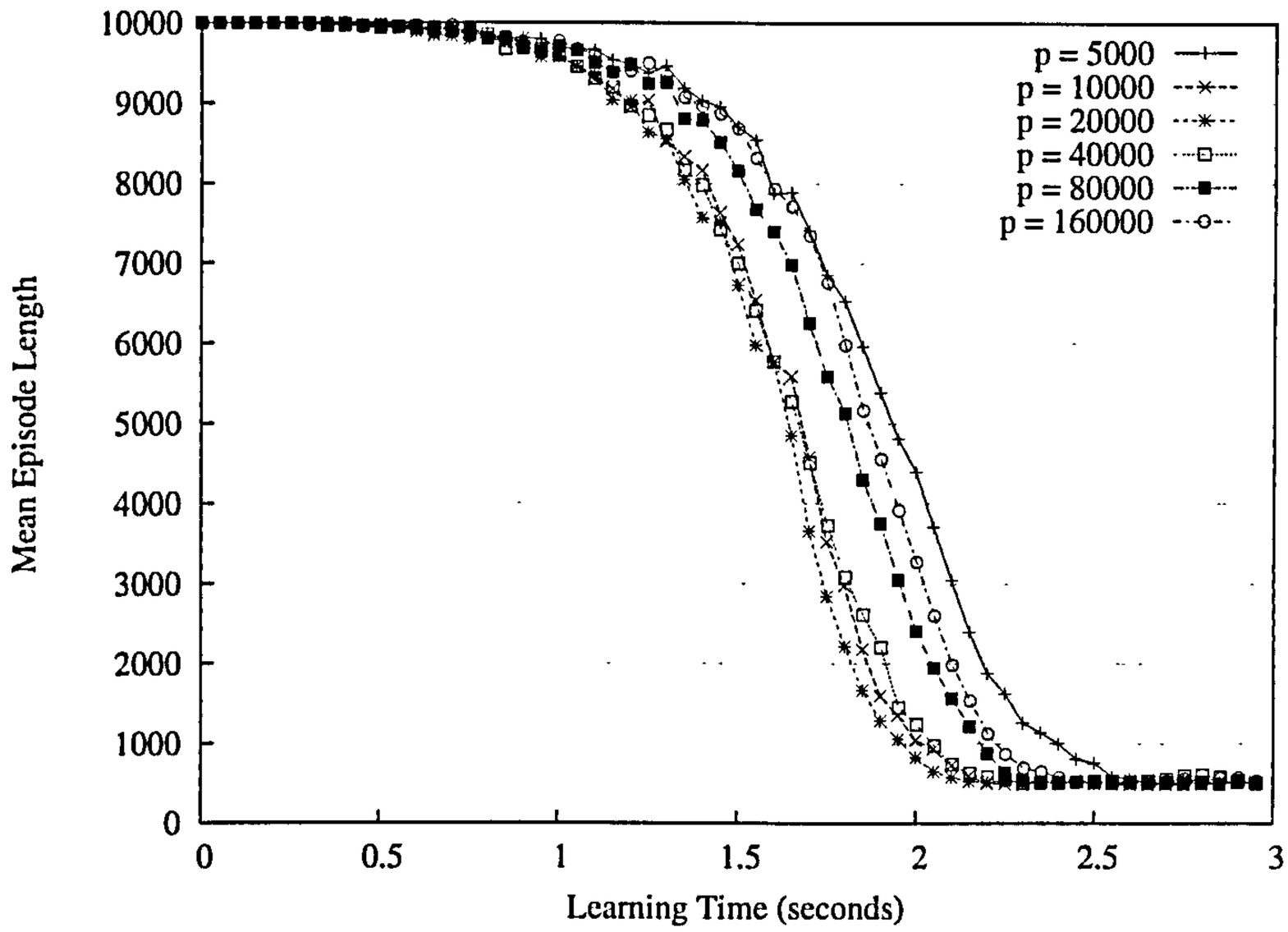


Figure 4.30: Varying the merge period p for *2 agents* in the low-difficulty Stochastic Grid World, using the *cluster* of workstations.

Merge Period	Proportion of time communicating
5000	36.2%
10,000	22.3%
20,000	13.2%
40,000	7.5%
80,000	4.3%
160,000	2.6%

Table 4.7: Proportion of experiment time expended on communication by the 2 agents for different merge period values.

The distribution of the learning curves on the cluster is quite different from those obtained in simulation. As we observed in Figure 4.29, there comes a point when reducing the merge-period p no longer reduces the number of simulation steps required for convergence. On the cluster, however, reducing the merge-period p will *always* increase the overall proportion of the experiment time that the agents must dedicate to communication. This reduces the time available for simulation and VFA updates, to some extent cancelling out the increased sample efficiency obtained by more frequent merging.

The overall effect on performance is therefore as follows. When we start with a very large value for p and gradually reduce it, performance gradually improves over that of a single agent, but not by as much as was achieved in simulation. Eventually there comes a point where the increase in communication costs outweighs the benefits of more frequent merging, and reducing the value of p causes performance to become progressively worse. There is therefore some optimal value for p in this experiment at the point where these two effects are perfectly balanced. The closest merge period in Figure 4.30 to the optimum is $p = 20,000$, where the overall time dedicated to communication is 13.2% of the total experiment time.

Figure 4.31 shows a similar experiment on the cluster of workstations using 16 agents rather than 2. While the overall pattern is quite similar to that in Figure 4.30, an interesting outcome of this experiment is that the optimal merge period for the experiment with 16 agents is *different* from the optimum for 2 agents. The closest merge period to the optimum in Figure 4.31 is $p = 10,000$, where the overall time dedicated to communication is 49.7% of the total experiment time (see Table 4.8.) Note how such a large amount of time must be dedicated to communication in order to achieve as much as possible of the 16x speedup achievable in simulation. Note also that although there are different optimal values of p for 2 and 16 agents, choosing a single value for p between 10,000 and 20,000 will produce results which are *close* to optimal for both numbers of agents in this domain.

At this point it is reasonable to ask how we can choose a good (or even optimal) value for p given a particular domain and some number n of available agents. Performing a series of experiments (as shown above) to approach the optimal value for p is impractical given that the stated goal of this thesis is to *speed up* RL using parallel hardware. By the time we have run enough experiments to choose the optimal value for p, it is likely that a single agent could have already solved the problem.

On the other hand, it is unlikely that we can derive an analytic method for determining the optimal value of p. This is because the *sample complexity* of the group of agents using the visit-count merge operation is difficult to model

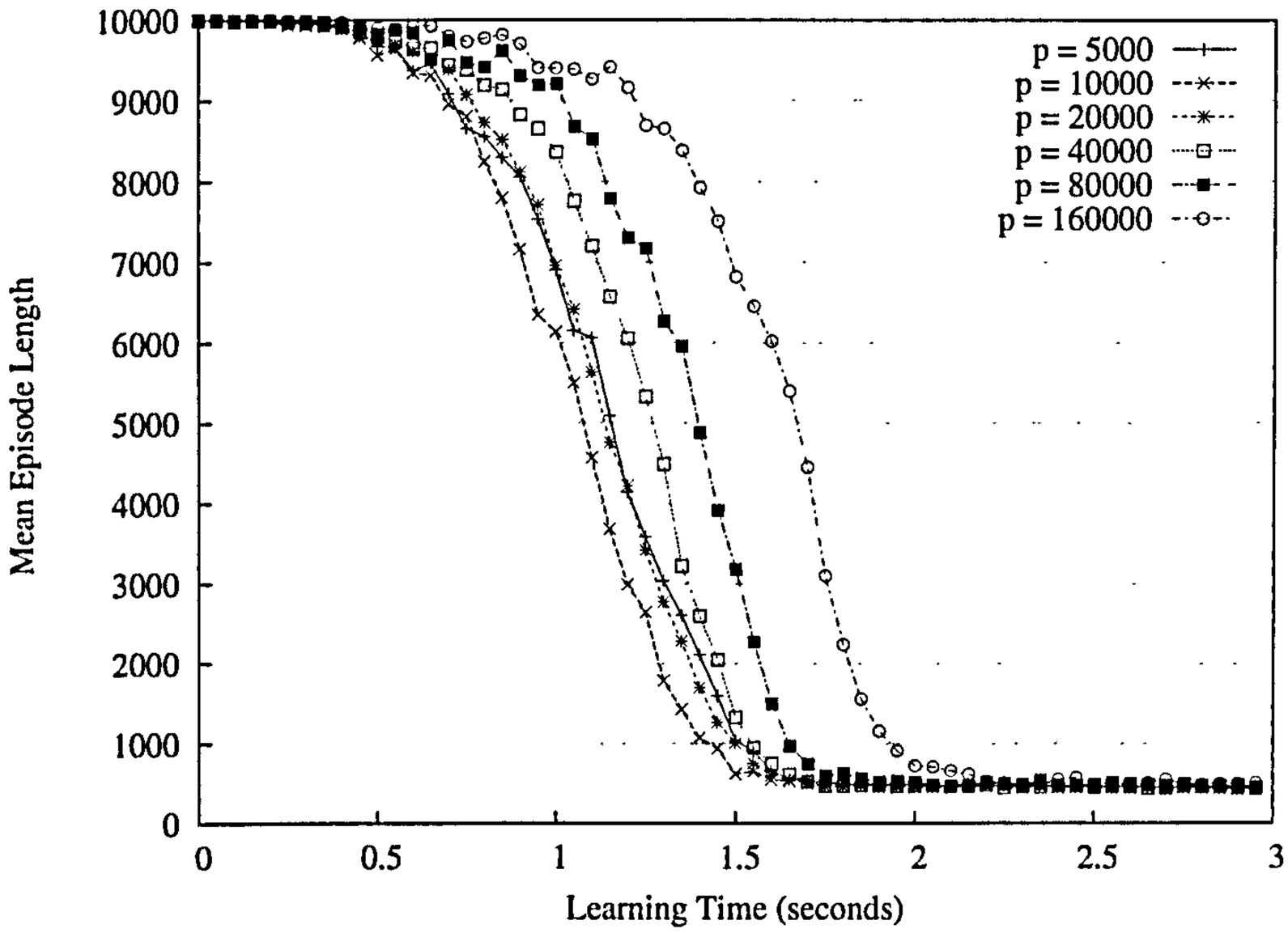


Figure 4.31: Varying the merge period p for *16 agents* in the low-difficulty Stochastic Grid World, using the *cluster* of workstations.

Merge Period	Proportion of time communicating
5000	64.8%
10,000	49.7%
20,000	35.2%
40,000	22.7%
80,000	14.7%
160,000	10.3%

Table 4.8: Proportion of experiment time expended on communication by the 16 agents for different merge period values.

analytically. Empirical results have shown that the performance of the agents depends on (at least) the merge period p, the number of agents n, the number of features f, and the overall difficulty of the target learning domain. Deriving an analytical model of convergence is also complicated by the fact that the *rate of convergence* of RL algorithms has seen little theoretical study to date.

Therefore for practical applications of the visit-count merge method in new domains, the merge period p must be chosen using a mixture of trial and error and reference to previous applications of the method in domains of similar scale and/or difficulty[3]. This approach is helped by the fact the performance of the method is relatively *insensitive* to small changes in the value of p, as shown in Figures 4.30 and 4.31. This means that as long as the value of p is only a few times smaller or larger than the optimum value, the overall performance will only be degraded by a small amount. It may be possible in the future to develop a heuristic method for selecting p based on the processing power of the parallel nodes, the bandwidth of the network, and some measure of the overall difficulty of a given learning domain.

4.9 Summary and Conclusions

The following material has been presented in this chapter:

- Motivation for the use of parallel hardware to find near-optimal solutions to RL problems more quickly than is possible with sequential computation.
- A description of the necessary assumptions to use parallel hardware for RL.
- A general approach to parallel RL based on *merging* value function approximations.
- Several instantiations of the general approach using a series of *merge functions.*
- An evaluation of each merge function using a *simulation* of parallel agents.
- A description of how *decaying* α and ϵ parameters in combination with a *binary search* can be used to compare parallel RL methods by final solution quality.
- A comparison of the merging method to a parallel RL method which does not merge.

[3]There are already several RL parameters (such as α, ϵ, γ and λ) which require this kind of selection.

- An evaluation of the *visit-count* merge function using real parallel hardware; a *cluster of workstations.*

- An analysis of how the choice of merge period p affects performance, in both the simulation and on the cluster.

From this material we can draw the following broad conclusions:

- In the simulation of parallel agents, a group of agents which *merge* their value functions require significantly fewer *simulation steps per agent* to converge to a near-optimal policy than a single-agent learner.

- While the *maximum* and *minimum* merge functions exhibited divergent behaviour, the body of empirical evidence collected suggests that both the *mean* and *visit-count* merge functions allow (and accelerate) convergence to a near-optimal policy.

- The *visit-count* merge function achieves better performance in simulation than all the other merge functions in all the domains tested. This is mainly because it uses a measure of experience to weight the agents' value estimates.

- As the merge period p is reduced, the simulated agents approach the limit of *linear speedup* over the single-agent learner.

- The cluster implementation showed that the communication overhead of the merging method is so large that in many domains a parallel speedup could not be achieved with the available hardware.

- In spite of the large overhead, significant parallel speedups were obtained in several instances of the Stochastic Grid World task, demonstrating the practical potential of parallel RL.

- An appropriate choice of the merge period p is a prerequisite for achieving good performance on parallel hardware. The overall performance of the method is not very sensitive to small changes in the value of p.

- While the efficiency of the merge function could probably be improved, it is the reduction of network bandwidth requirements which is most likely to allow a related parallel RL method to achieve better results in practice.

In Chapter 5 I will go on to examine how parallel speedups can still be achieved while drastically reducing the communication costs of the merging method described in this chapter.

Chapter 5

Selective Merging

In the previous chapter, a parallel RL method based on *merging* value function approximations (VFAs) was presented. This method was shown to require fewer simulation steps per parallel agent as the number of agents n was increased. However, the method was also expensive in terms of communication overhead, which meant that speedups on real parallel hardware could not be achieved in all of our evaluation domains.

In this chapter, I will present a new parallel RL method based on a similar notion of merging, but with a much lower communication overhead. This method is based on agents exchanging their *recent changes* to the VFA weights. Communication overhead is greatly reduced by each agent broadcasting only the *largest* of its recent changes. Several candidate mechanisms are proposed for combining changes received from several agents, and are evaluated using an implementation on the cluster of workstations. I will also examine the effect of varying both the *period* between consecutive communications and the *number of changes* sent in each communication.

5.1 Motivation

The parallel RL method described in Chapter 4 was based on a periodic merge operation. During this operation, for every feature ϕ_i a merged weight θ_i^m was calculated from all the agents' most recent estimates of θ_i. This meant that *every* weight of *every* agent had to be communicated over the network *at least once* during the merge operation. The advantage of this approach is that the merged VFA is an estimated summary of the total group knowledge over the whole state space. The key disadvantage is that a large quantity of data must be exchanged between the agents. Even using an efficient distributed computation to calculate the merged VFA results in a significant delay while the operation is completed. This consumes

time which could have been spent learning in the simulated environment.

If every agent must communicate every weight in its VFA, there will be a great deal of redundant information exchanged between the agents. For example, suppose that for some RL problem there is a subset of the total feature set $\vec{\phi}$ containing features which are only active when the agent visits a particular region of state space. Suppose that none of the agents have visited this region by the time one of the merge operations takes place. During the merge operation, weights (and possibly visit-counts) will be communicated for all the features in this subset, despite the fact that the agents derive no useful information at all from this communication.

To remove this redundancy, it could be suggested that only weights that have been updated since the last merge (i.e. those with a non-zero visit count) should be communicated to the other agents. However, consider the following alternative example. Feature ϕ_i has been active many times for all the agents since the last merge, which means that each agent has made many updates to weight θ_i. After all the updates have been made, the agents' updated values for θ_i show little change from the result of the previous merge. This is a situation which occurs when state-action values in some region of the state space are already predicted well by the VFA. Transmitting all the agents' weights to make such a small adjustment to θ_i is not redundant as such, but is clearly less informative to the group than the large weight adjustments which occur when one agent finds a previously undiscovered high reward region of state space.

In this chapter a new parallel RL method is defined in which each agent *prioritizes* its most informative weight adjustments. This means that our focus will shift away from an agent's absolute value of θ_i and onto the recent *change* $\Delta\theta_i$ in the agent's value of θ_i. Messages sent to other agents will no longer contain values of θ_i, but values of $\Delta\theta_i$ instead. It is reasonably simple to make this adjustment, since all the agents use the same value θ_{init} for the starting value of θ_i, and the use of reliable message transport means that every agent is guaranteed to receive all the transmitted values of $\Delta\theta_i$. Therefore, an agent can calculate a new value for θ_i as each change arrives, allowing the agents as a group to derive identical[1] values for θ_i.

The method defined in this section is based on the following principle for choosing which information is worth communicating to the other agents:

[1]Assuming that the agents are running on a homogeneous cluster or on an SMP machine. If the agents are running on a heterogeneous cluster, there is the possibility that differences in the floating point implementation will cause the agents' calculated values of θ_i to differ to some extent.

> The most informative elements of the weight change vector $\Delta\vec{\theta}$ are the elements which have the *largest absolute values*, and these are the changes which should be sent to the other agents in the group.

While there may be some situations in which this principle could be misleading (e.g., if two actions in a state have a similar mean reward but very different variances), in the majority of cases it will allow the prioritization of the most significant changes to the VFA as learning progresses.

Note that none of the selective methods described in this chapter employ the *visit-counts* which were the basis of the best method described in Chapter 4. This is because having information about the *change* undergone by a weight makes the information about the number of updates to the weight much less valuable[2]. Methods which transmit visit-counts in addition to weight changes seem to increase communication costs without making much difference to the convergence rate, and therefore perform poorly by comparison.

5.2 Method Definition and Implementation

Like the merging method of Chapter 4, the *selective merging* method studied in this chapter is based on a *periodic merge operation* which occurs after every agent has performed a set of p simulation steps. I will continue to refer to p as the *merge period* parameter.

The selective merging method makes extensive use of the weight change vector $\Delta\vec{\theta}$. Maintaining a copy of this vector during learning would require a change to the underlying SARSA(λ) implementation so that $\Delta\vec{\theta}$ was updated each time $\vec{\theta}$ changed. To avoid having to modify the implementation and make it less efficient, each agent instead stores a vector $\vec{\theta}^{ref}$, which holds the last known "group" value for each weight. The SARSA(λ) implementation continues to update $\vec{\theta}$ only, and it is easy to calculate $\Delta\vec{\theta} = \vec{\theta} - \vec{\theta}^{ref}$ when required. Each element $\Delta\theta_i$ of the weight change vector signifies the change made to weight θ_i since the "group" value was last determined. At the beginning of each run, the values of θ_i and θ_i^{ref} are set to the initial value of θ_{init} for all i. This makes the initial value of each $\Delta\theta_i$ zero.

At the start of a merge operation, each agent calculates the weight change vector $\Delta\vec{\theta}$. The agent is now able to rank the set of weight indices $I = \{0, 1, ..., (f-1)\}$ using the absolute weight change $|\Delta\theta_i|$ for each index i. An additional parameter f_{com} defines how many of the weight changes will be communicated by each

[2]Though not valueless. Informing an agent that a weight has a high visit count but a small change may increase the agent's confidence in the value of the weight. However, if rewards are sparse then this confidence could be misleading in the early stages of learning.

agent during a single merge operation. The choice of parameter f_{com} involves another trade-off between growing communication costs and sample efficiency during learning. The effect of parameters p and f_{com} on the performance of the selective merging method will be examined in more detail in Section 5.5. Even if f_{com} is much smaller than the total number of weights f, it may still possible to speed up convergence significantly.

In the original merge method described in Chapter 4, the message format consisted of a vector containing the values of all f weights. Each of these transmitted values was associated with one of the features ϕ_i by the fact that the value appeared in the i^{th} position of the vector. In the selective merging method, only a *subset* of elements of the weight change vector $\Delta\vec{\theta}$ are sent during each merge operation. This means that we cannot identify the associated feature using the position of each $\Delta\theta_i$ value in the message. Instead, each message contains a set of $(i, \Delta\theta_i)$ tuples. The first member of the tuple identifies which of the f approximator weights is being referred to, and the second member is the recent change in weight θ_i observed by the sending agent. Each agent constructs a message in this way and *broadcasts* it to all agents in the group *including itself*[3].

Towards the end of the merge operation, an agent will have received a message from each agent in the group (including itself). The final stage of the merge operation is to incorporate all of the changes received in the messages into the agent's local data structures. There are three cases to consider for each feature θ_i, as illustrated in Figure 5.1. The first case is if *none* of the messages contain a change $\Delta\theta_i$ associated with θ_i. In this case we make no update to θ_i or θ_i^{ref}. Note that this allows small changes to accumulate over several merge periods, eventually resulting in a large change which will be submitted.

The second case is if *only one* of the messages contains a change $\Delta\theta_i$ associated with θ_i. In this case both θ_i and θ_i^{ref} are set to the value of $\theta_i^{ref} + \Delta\theta_i$. This ensures that after this particular merge operation all the agents will start measuring changes from the same "group" estimate for the weight value. Note that a side-effect of this update is that any small change discovered by an agent which was *not* transmitted will be lost after the update.

The third case is if *more than one* message contains a change $\Delta\theta_i$ associated with θ_i. If the set C contains all the change values associated with weight θ_i, then a (partial) function $g : \mathcal{P}(\mathbb{R}) \rightarrow \mathbb{R}$ is required to combine the information received from all the agents who discovered a significant change in θ_i. Making a suitable

[3]This allows the algorithm for updating the VFA to be defined as an operation performed on a *set of messages*, without distinguishing the local agent's message in any way. The implementation on the cluster of workstations simply keeps the local message in memory until it is required.

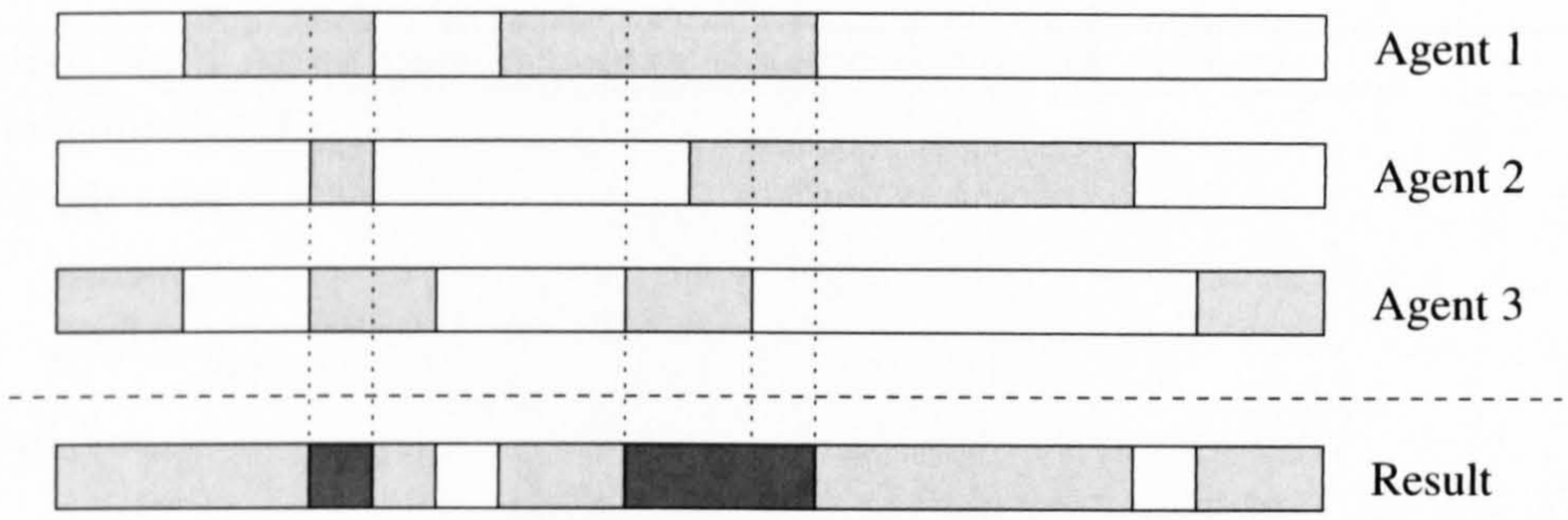


Figure 5.1: The *selective merge* operation. With the approximator weights represented as a one dimensional vector, the light grey regions indicate weight changes transmitted by each of the agents. The resulting merged weights consist of unchanged weights (white), weights changed by a single agent (light grey), and weights where changes from several agents must be *combined* together (dark grey).

choice for function g is non-trivial, and a number of different candidates for such a function will be considered in Section 5.3.

For a given choice of the combination function g, the procedure followed by an agent using the selective merging algorithm is shown in Algorithm 4.

Note that at the high level (or equivalently at the *superstep* level) selective merging is a *synchronous* algorithm. While each agent can execute its p simulation steps without requiring any synchronization, the merge operation requires that each agent waits for a message from every member of the group before it can update its VFA and proceed with the next p simulation steps.

While the merge operation is synchronous at the high level, the message send and receive operations used in Algorithm 4 were actually implemented using non-blocking asynchronous *point-to-point* operations provided by the MPICH library. This contrasts with the synchronous *collective* operations that were used in the improved implementation of the original merging method (see Section 4.7).

In our implementation, a message (consisting of a vector of $(i, \Delta\theta_i)$ tuples as described above) is constructed in a buffer of user-allocated memory. The `MPI_Isend` function is used to initiate an asynchronous send to each of the agents in the group. An agent waits for messages from the rest of the group using `MPI_Probe`. When `MPI_Probe` returns, indicating that a message has been sent by one of the other agents, the agent uses `MPI_Recv` to receive the message into a temporary user buffer. Each incoming buffer is processed immediately, allowing part of the calculation towards a vector of $g(C)$ values to be completed.

The use of asynchronous message passing at the low level outperformed any of the MPICH collective operations or synchronous point-to-point operations. There are two main reasons for this. The *order* in which messages are received and

Algorithm 4 Agent pseudocode for the selective merge method.

{Initialization}

for all i **do**

$\theta_i \leftarrow \theta_{init}$

$\theta_i^{ref} \leftarrow \theta_{init}$

end for

{Main Loop}

while time elapsed $< t_{end}$ **do**

{Learning Phase}

for $step = 1$ to p **do**

Execute a simulation step and update weight vector $\vec{\theta}$.

end for

{Construct Message and Send}

Calculate $\Delta\vec{\theta} = \vec{\theta} - \vec{\theta}^{ref}$.

Rank each index i according to the value of $|\Delta\theta_i|$.

$best \leftarrow$ { the f_{com} highest ranked indices }

$m \leftarrow \{(i, \Delta\theta_i) \mid i \in best\}$

Send message m to all agents (including self).

{Receive Message and Update Weights}

$mset \leftarrow$ {Messages received from self and others}

for all i **do**

$cset \leftarrow \{\Delta\theta_i \mid m \in mset, (i, \Delta\theta_i) \in m\}$

if $|cset| = 0$ {Case 1} **then**

{No update}

else if $|cset| = 1$ {Case 2} **then**

$\theta_i^{ref} \leftarrow \theta_i^{ref} + c_0$ {where c_0 is the only element of $cset$}

$\theta_i \leftarrow \theta_i^{ref}$

else if $|cset| > 1$ {Case 3} **then**

$\theta_i^{ref} \leftarrow \theta_i^{ref} + g(cset)$

$\theta_i \leftarrow \theta_i^{ref}$

end if

end for

end while

processed *is not significant* in the selective merge method, so it is more efficient to use `MPI_Probe` to retrieve the first message received than imposing a fixed order to receive the messages. In addition, as we shall see in Section 5.3, it is possible to perform incremental computation of the required $g(C)$ values *as and when messages are received.* This allows communication and computation to be *overlapped* to some degree, which is more efficient than waiting for a collective communication to complete before beginning to calculate the $g(C)$ values.

5.3 Combining Changes from Several Agents

Section 5.1 described the core procedure followed by the selective merging method, while temporarily leaving undefined the $g(C)$ function for combining changes received from several different agents. In this section, we will more closely examine the purpose of this function, and motivate a number of candidates for the algorithm which will be evaluated in Section 5.4.

5.3.1 Criteria for Combining Changes Together

It is first necessary to consider what criteria are important for selecting the $g(C)$ function. From a high level perspective, our method must be effective at combining information from several agents, some parts of which will be complementary, and other parts of which will be conflicting. At the lower level, the changes made by each of the agents to θ_i must be combined into a single representative change. Each of these change values represents an agent's accumulation of all the recent value function updates where feature ϕ_i was active. An update to θ_i occurs for one (or more) of the following three reasons:

1. Stochasticity arising from either the transition and reward functions or the exploration strategy means that once a weight is close to the actual expected value, small updates in both the positive and negative directions will occur in response to *noise in the sampled value.*

2. The VFA must *generalize* over the state space using only a few features. Once the VFA accurately approximates the true value function, states encountered during learning will have both positive and negative *generalization error*, resulting in a series of small updates to the weights in both directions.

3. The weights are initialized arbitrarily at the start of learning, which means that early estimates of feature value are highly *biased.* As sampled experience is collected, more accurate estimates of long term reward are *propagated*

backwards through the state space, reducing the initial bias. Weight updates which reduce bias are unidirectional.

Weight updates corresponding to reasons 1 and 2 are those which arise from variance in the samples once a weight approaches its true expected value. However, weight updates corresponding to reason 3 are fundamentally different in character, producing a series of large updates in a single direction until the large initial bias in the weight values is reduced.

Now consider a set of changes C which were made by several agents to feature θ_i over the last merge period. How can these changes be combined in a way which improves upon the performance achieved by a single agent? It is fairly clear that taking the *mean* of the values contained in C will improve the effect of updates corresponding to reasons 1 and 2. Using the mean of the changes in this way results in the estimate of θ_i staying much closer to the expectation as updates in response to sampled experience. The agents essentially combine their individual estimates to obtain an improved group estimate of the expected value.

However, consider the effect of using the *mean* of the changes in the following situation. Through random exploration, a single agent discovers a previously unknown high reward region, and updates θ_i to reflect this. None of the other agents manage to find this region, and only make very small changes to the value of θ_i. This means that C will contain one very large change and a series of small changes (both positive and negative). Assuming that the small changes are relatively insignificant, the effect of taking the *mean* of these changes is to reduce the magnitude of the large change by a factor of n (where n is the number of agents.) This would actually result in a slower rate of convergence than that of a single agent learner[4].

This example illustrates the conflict which has to be addressed when choosing a function $g(C)$. When a large reward is discovered for the first time, the change in value must be propagated quickly through *both* the value function *and* the population of agents. However, it is also desirable that function $g(C)$ improves the estimation of the expected return once the initial bias has been eliminated.

This conflict is an example of a more general property of machine learning algorithms, namely the *bias/variance dilemma* (Geman et al., 1992). The average mean squared error (MSE) in the approximation of the optimal value function can be decomposed into two parts: bias and variance. Denoting the optimal value

[4]In practice the selective merge method actually performs much better than this as long as f_{com} is much smaller than the total number of features f. If this is the case, then the process of ranking the weight changes in order of magnitude makes it fairly unlikely that C will contain a large number of very small changes.

function by Q^*, using $\tilde{Q}$ for the approximation learned after real time T, and denoting the expectation after time T as E_T, bias and variance in this context of this chapter can be expressed as follows:

$$bias(s,a) = (Q^*(s,a) - E_T\{\tilde{Q}(s,a)\})^2$$

$$variance(s,a) = E_T\{(\tilde{Q}(s,a) - E_T\{\tilde{Q}(s,a)\})^2\}$$

The overall bias and variance of the approximator at time T can be obtained by integrating over the state-action space. Intuitively, the *bias* is the squared difference between the expected approximation value and the true optimal value, and the *variance* is the expected deviation of the approximation value from the expected approximation.

The bias/variance dilemma is related to the problem of *overfitting* an approximation to a set of training data. If the approximation fits the data too closely, the bias tends to be small but the variance will be large. To get a small generalization error it is important to achieve a good trade-off between the size of the bias and the size of the variance.

Thus the issues involved in choosing the combination function $g(C)$ can be interpreted in terms of the effects on bias and variance. For example, a simple mean function for $g(C)$ will result in a reduction of variance in the value estimates, but bias is likely to be reduced more slowly over time.

5.3.2 The Problem of Overshooting

Consider how a single SARSA(λ) learning agent updates its VFA. The evolution of each weight over time can be described by a *summation* of small changes caused as experience is collected. In the limit (assuming that α and ϵ are decayed appropriately) this summation will tend towards the expected value of the associated feature. But what if n agents were simultaneously contributing changes to the same summation? Would the value of each weight approach the expectation more quickly? To answer this question, the combination function $g(C)$ can be defined as:

$$g(C) = \sum_{c \in C} c$$

This simplistic approach exhibits some serious problems. Results for the Pole-Balancing task (in Figure 5.2) show that 2 agents using this combination function learn a policy of higher quality than the single-agent. Unfortunately, if 4 or more agents are used the agents do not converge to any useful policy. Results for the

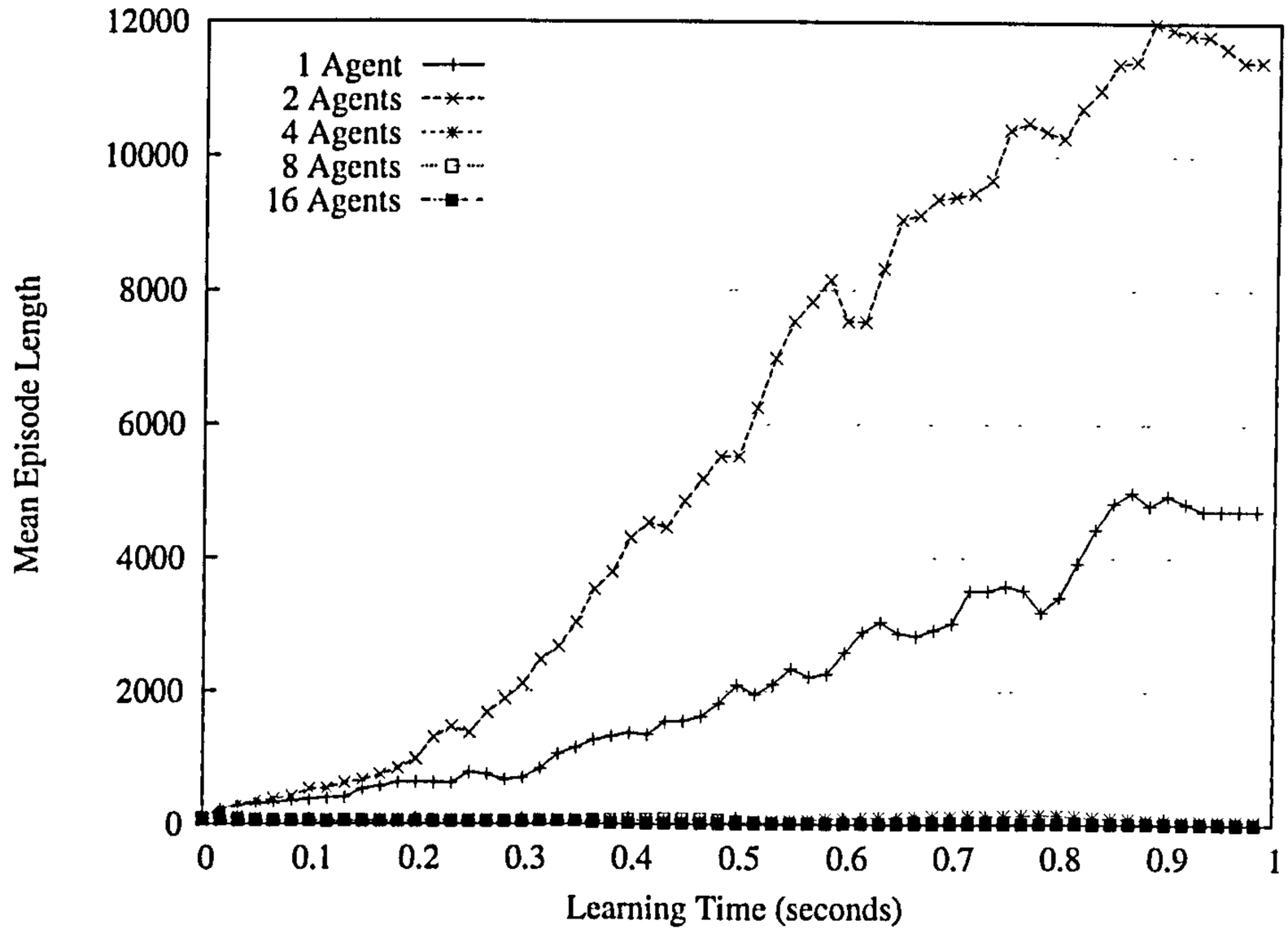


Figure 5.2: Using a simple summation for $g(C)$. Results for the selective merge method in the Pole-Balancing task, collected using the cluster of workstations.

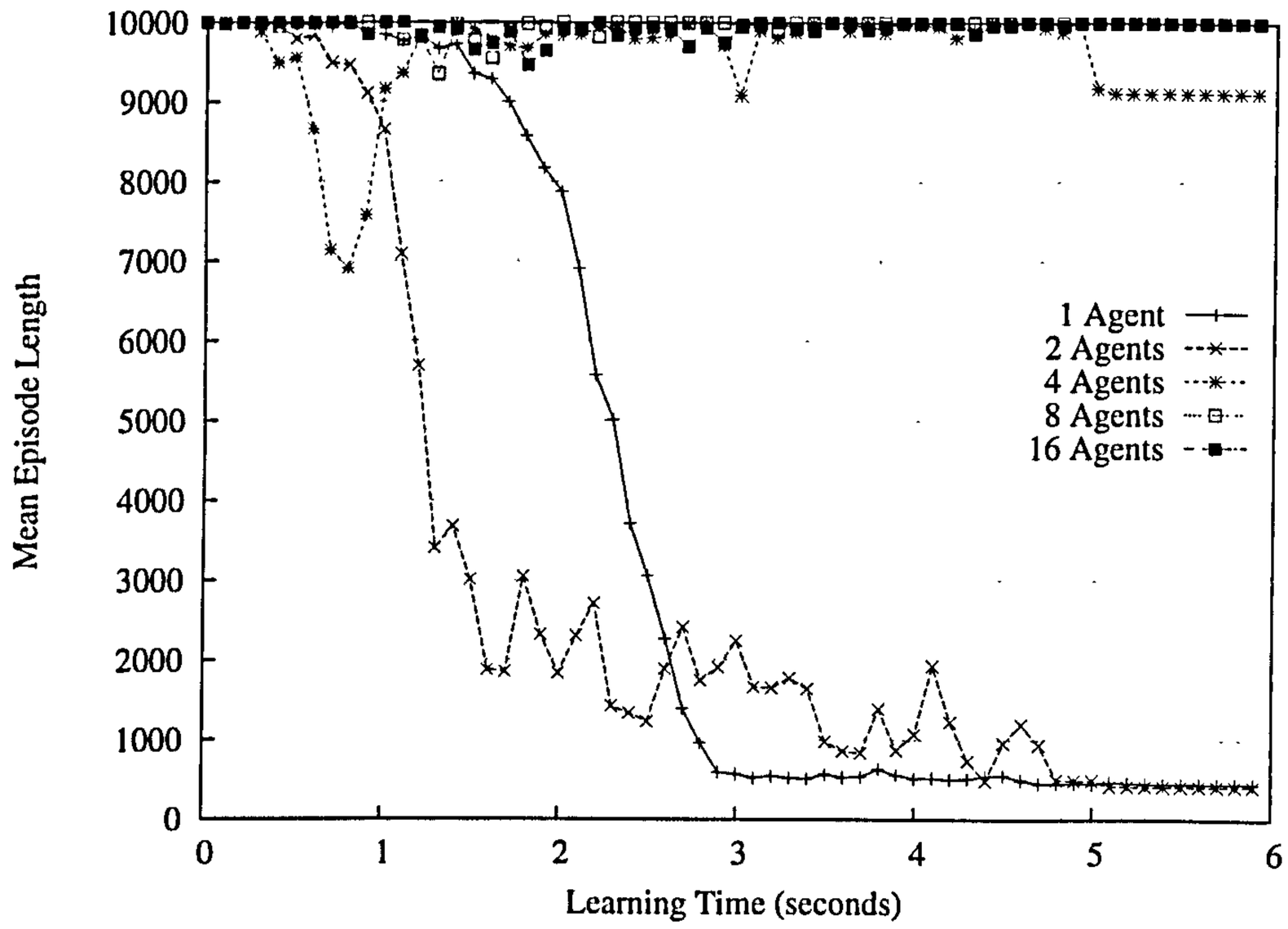


Figure 5.3: Using a simple summation for $g(C)$. Results for the selective merge method in the (low-difficulty) Stochastic Grid World task, collected using the cluster of workstations.

Stochastic Grid World (in Figure 5.3) exhibit a similar pattern. The 2 agent group does learn more quickly in the initial stages of each run, but produces noisy results and takes a lot longer to settle on a near optimal policy. If 4 or more agents are used, selective merging does not converge. Full details of the parameter settings used for these experiments are given in Section 5.4.

The problem with the summation is that there is no mechanism for dealing with agents which make *identical changes*. For example, suppose there are n agents just beginning a run of selective merging. Suppose also that there is a reward close to the initial state that can be found with very little exploration. Each of the agents receives the reward a number of times during the first merge period, so that the state in which the reward is available has an estimated value similar to the reward value. Now when the selective merge occurs, and these changes are summed together, the new estimated value of the state could be as much as n times the true expected value of the state. *Overshooting* the expected return of a state in this way is a major problem. It is likely that overshooting will set up an oscillation about the true value of the state, which may even cause it to diverge to infinity.

5.3.3 Candidates for the Combination Function

I will now define four candidates for the combination function $g(C)$. Each of these candidates has been selected to avoid the overshooting problem observed in Section 5.3.2. In addition, I will use the criteria discussed in Section 5.3.1 to assess the potential of each of the candidates for successfully combining changes to the VFA weights. The four candidates are evaluated in detail in Section 5.4.

Combination Function 1 - Capped summation

The first combination function uses a summation as described in Section 5.3.2, except that this time the result of the summation is *bounded* so that it cannot be greater than the largest element of the set C or less than the smallest element of the set C.

$$g(C) = bound\left(\min_{c \in C}(c)\ ,\ \sum_{c \in C} c\ ,\ \max_{c \in C}(c)\right)$$

where

$$bound(l, x, u) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } x < l \text{ then } l \\ \text{else if } x > u \text{ then } u \\ \text{else } x \end{array}$$

This bounded summation has some of the same properties of the simple summation: propagation of new rewards is fast because large changes contribute the most to the sum, but the variance of value estimates close to the true expected value may be increased. However, bounding the summation has two additional effects. Firstly, the increase in variance is bounded as the number of agents n is increased, since each combined weight change cannot exceed the greatest magnitude of weight change achieved by a single agent. Secondly, in a situation where several agents discover identical changes, the combined value will no longer greatly overshoot the expectation since the summation is bounded.

Combination Function 2 - Change of largest magnitude

The second combination function selects the change in set C which has the largest magnitude (absolute value). This change is then used as the combined change value.

$$g(C) = \arg\max_{c \in C} |c|$$

The effects are similar to those of function 1 despite the fact that the mechanism is quite different. New rewards are propagated quickly because they produce changes of high magnitude. However, there will be greater variance in the value estimates close to the true expected value, since choosing changes of the largest magnitude favours sampled values on the margins of the value distribution.

Combination Function 3 - Mean of the changes

The third combination function is a simple mean, used as the example in Section 5.3.1.

$$g(C) = \frac{1}{|C|} \sum_{c \in C} c$$

As indicated earlier, this function will perform well at reducing variance in the value estimates close to the true expected value. However, it is possible that rewards which are difficult to discover will propagate more slowly through the value function, because large changes from agents which do find the reward will be reduced in magnitude by agents which do not.

Combination Function 4 - Weighted Average

The fourth combination function is a weighted average, which is similar to the mean used in function 3 except that each member of the set has a contribution

weighted by the *magnitude* of the change. The weighted sum of the changes is normalized by dividing by the sum of the change magnitudes.

$$g(C) = \frac{\sum_{c \in C} c \cdot |c|}{\sum_{c \in C} |c|}$$

The result of using this weighted average as the combination function is that outliers in the set C have a much more significant effect on the result than when a simple mean is used. If one change c is much greater in magnitude than the others in set C, the combined change will be very close to c. This means that newly-discovered rewards propagate quickly through the value function. If, on the other hand, most of the changes in set C are of similar magnitude, the result $g(C)$ will be closer to the mean of the members of C, allowing value estimates to converge more quickly to the estimation. Note that for states where the future return has a very high variance, the tendency of $g(C)$ to favour outlying changes may interfere with reducing the variance of value estimates.

This combination function can be seen as making a *trade-off* between propagating new rewards quickly and improving convergence to the long term expected value of states.

5.4 Evaluation using the Cluster of Workstations

The evaluation carried out for the selective merging method involved testing the performance (with different numbers of agents) of each of the four combination functions in each of the evaluation domains defined in Section 4.3.1. The motivation behind each of these functions (given in Section 5.3.3) included an informal assessment of which functions would be most appropriate in some situations. However, the performance that can be achieved in practice depends on a range of factors including the stochasticity of the underlying domain, the form of the reward function used, and the choice of algorithm parameters p and f_{com}. The evaluation domains used here exhibit a range of different characteristics, allowing an assessment of the likely performance of the selective merging method in a broad range of situations.

Stochastic Grid World (low-difficulty)

The first results presented here examine the performance of the selective merging method in the low-difficulty Stochastic Grid World task. Figures 5.4–5.7 allow the performance of the four combination functions to be compared for groups of

2, 4, 8 and 16 agents. In this series of experiments, reward function #2 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. The parameters used for the selective merging algorithm were $p = 10000$ and $f_{com} = 256$. Recall that for this domain, the total number of features $f = 3600$. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 1 \times 10^{-8}$. These settings were also used for the earlier Stochastic Grid World experiment presented in Section 5.3.2.

The first observation that can be made about Figures 5.4–5.7 is that the results are very similar whichever combination function is used. This is an indication that the exact form of the combination function may not be as important to the success of the selective merging method as was initially thought (although preventing *overshooting* does seem to be a vital property). The only significant exception to this rule is the unreliable performance of combination functions 1 and 2 when there are 16 agents (see Figure 5.7). With combination function 1, the 16 agents initially converge quickly towards a good policy, but an increase in the overall variance of the value estimates prevents the agents from settling near the optimum until 1.5 seconds have elapsed. In addition, in at least one run (out of the 10 total runs) both combination functions 1 and 2 cause the 16 agent group to diverge from the optimum as decaying parameters α and ϵ approach zero.

Combination functions 1 and 2 both *increase* variance in the value estimates, but the effect is much less pronounced using combination function 2. In contrast, combination functions 3 and 4 are methods based on *averaging* the changes, resulting in a *decrease* in variance in most cases. Once the performance of the agents converges to around 500 steps per episode, the subsequent performance (of agents using the averaging combination functions) changes very little, even though exploratory actions and value function updates continue to be made.

Surprisingly, the results for combination function 3 (the mean of the changes) show convergence almost as rapid as for any of the other combination functions. This suggests that newly-discovered rewards propagate just as quickly through the value function. Propagation is not slowed by combination function 3 as was predicted earlier. One reason this could be the case is that during each merge operation, the agents each transmit only a small number (256) of changes out of the total possible (3600). This reduces the probability that one or more agents will broadcast small changes to a weight at the same time as another agent broadcasts a large change to the same weight (large changes are *much* more likely to be broadcast). It is reasonable to suppose that as f_{com} approaches the total number of features f, the degradation of convergence due to combination function 3 will

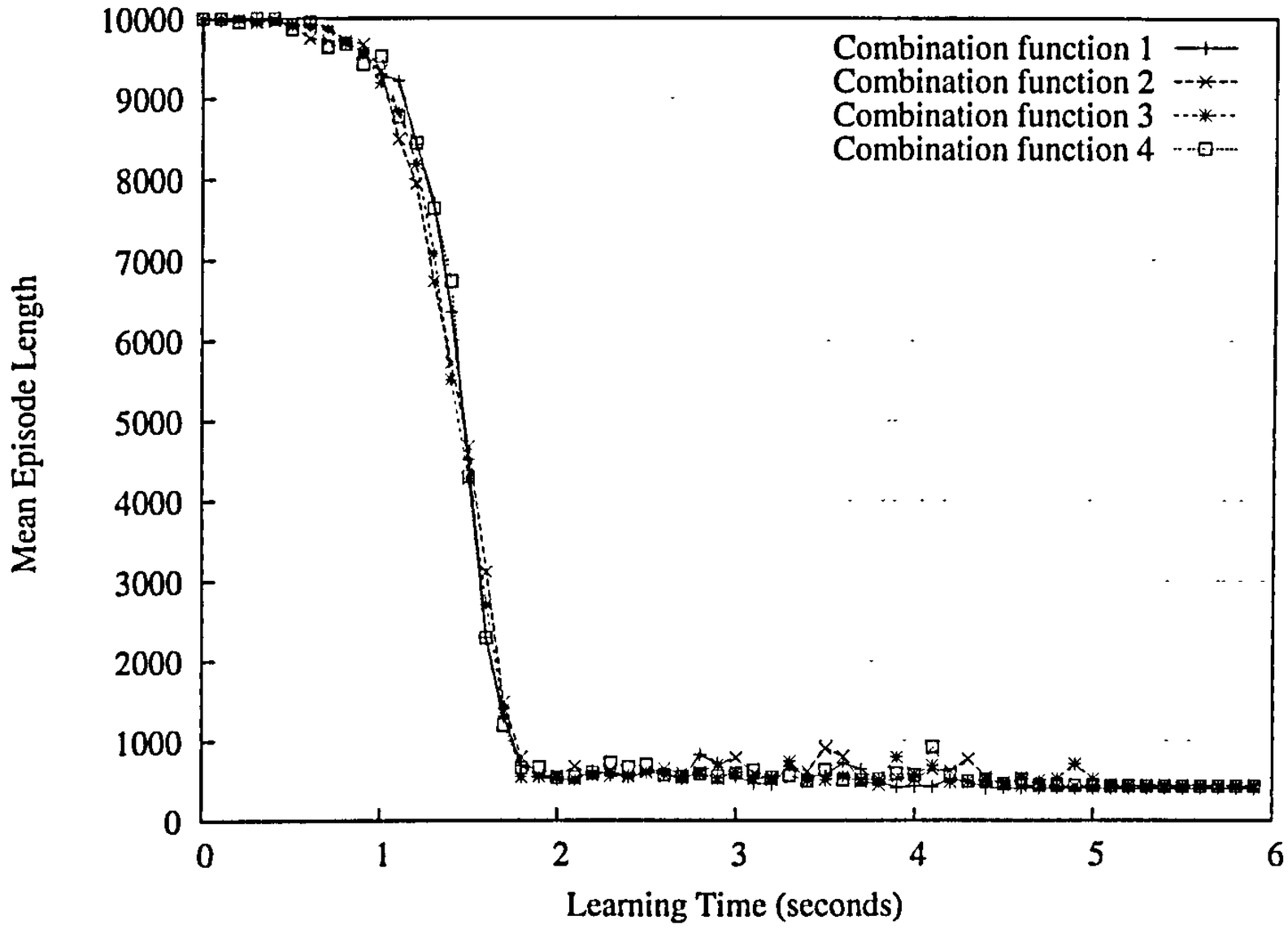


Figure 5.4: Comparing the combination functions using 2 agents in the low-difficulty Stochastic Grid World task.

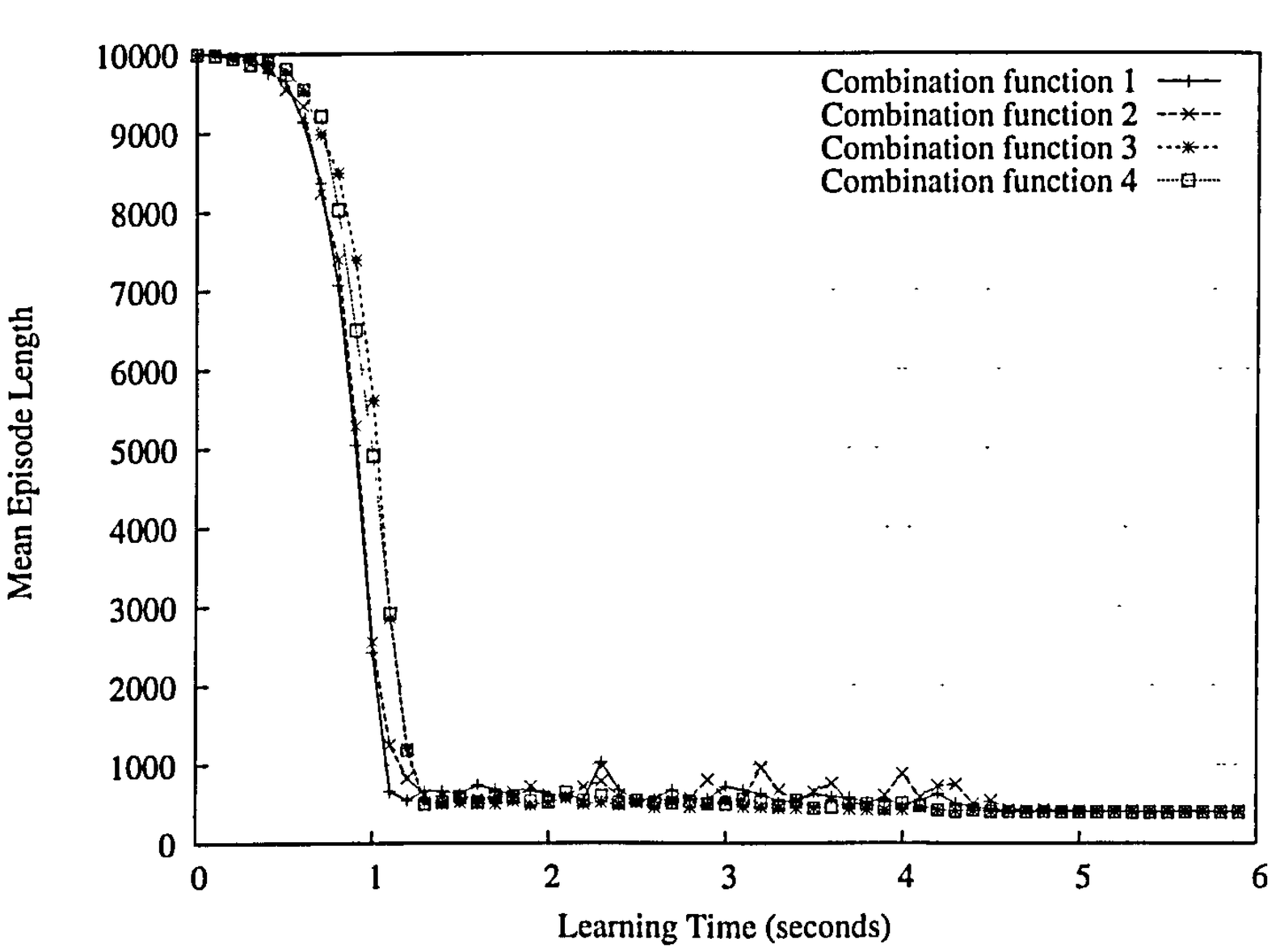


Figure 5.5: Comparing the combination functions using 4 agents in the low-difficulty Stochastic Grid World task.

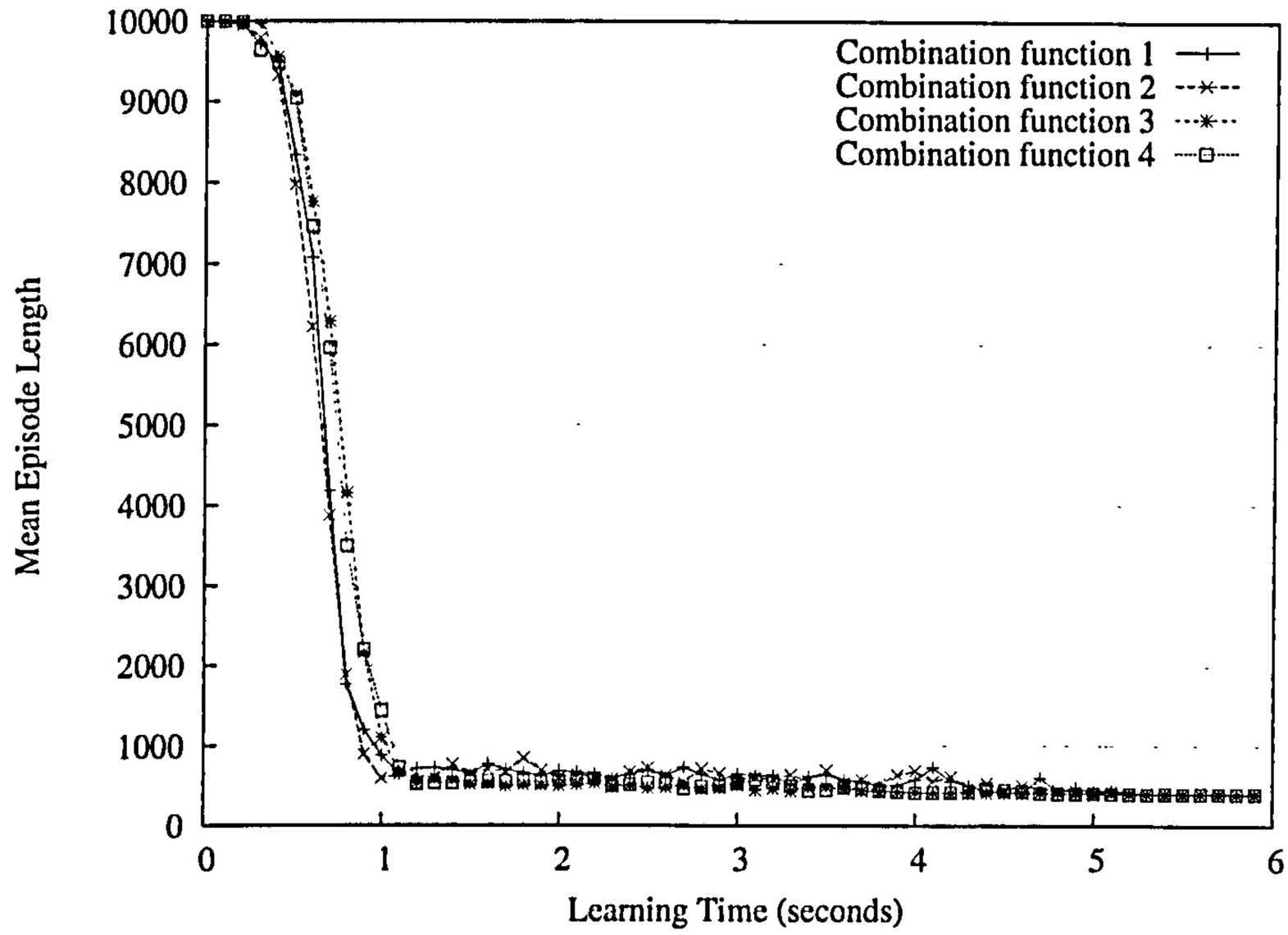


Figure 5.6: Comparing the combination functions using 8 agents in the low-difficulty Stochastic Grid World task.

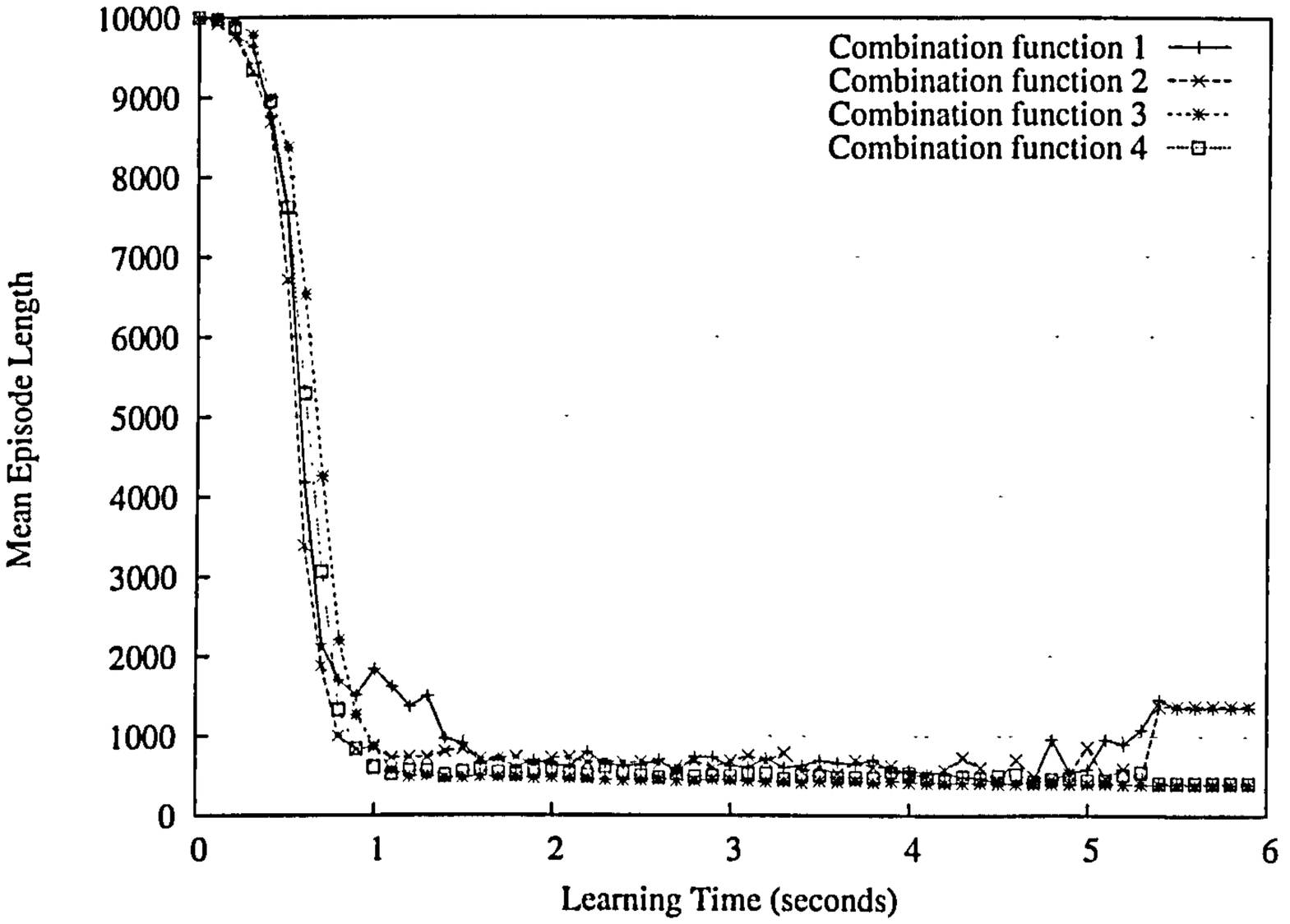


Figure 5.7: Comparing the combination functions using 16 agents in the low-difficulty Stochastic Grid World task.

be much greater.

In Figure 5.8 the performance of the selective merge method is compared directly with the performance of the visit-count merge method (which was described in Chapter 4). Combination function 3 was used with the selective method, since this produced very reliable results, as shown above. From the comparison it is clear that selective merging has a greater potential for achieving parallel speedups on the cluster of workstations. The visit-count merge method could not achieve convergence in a time less than 1.6s as the number of agents was increased. Increasing the number of agents from 4 to 16 made only a very small difference to the performance of the group.

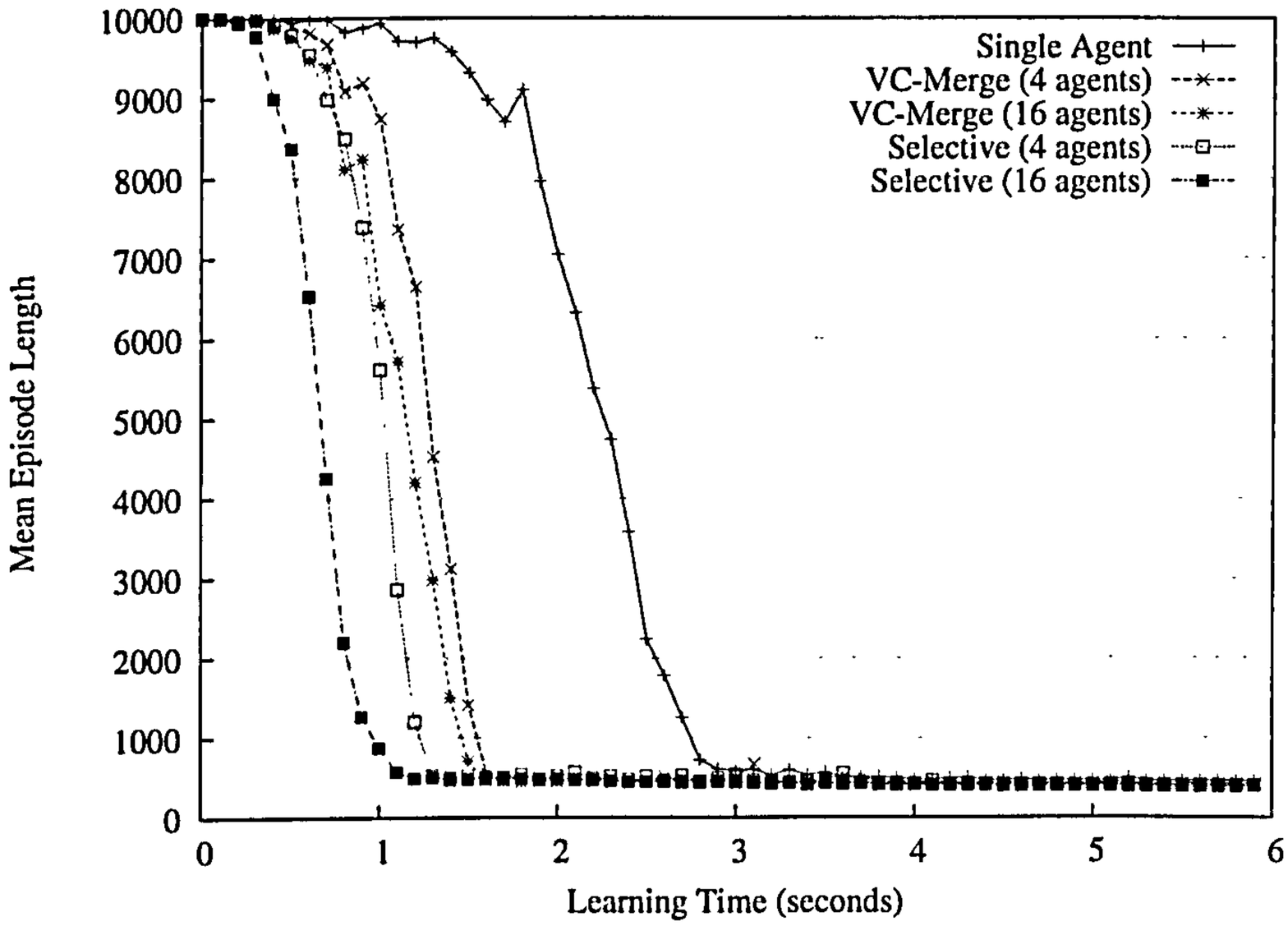


Figure 5.8: Comparing the performance of a single agent, the visit-count merge and the selective merge in the low-difficulty Stochastic Grid World task.

In contrast, using the selective merging algorithm, the rate of convergence continued to improve all the way up to 16 agents. With 16 agents convergence could be achieved in slightly over 1.0s. Even using only 4 agents, a significantly better speedup could be achieved than if 16 agents were used with the visit-count merge. Diminishing returns were still observed as the number of agents was increased, but for all numbers of agents that were tried the parallel speedup achieved was greater than that achieved using the visit-count merge.

Stochastic Grid World (high-difficulty)

Experiments were also carried out using the high-difficulty Stochastic Grid World task. These results were very similar in character to the results for the low-difficulty task. The speed of convergence is very similar no matter which combination function is used. Combination functions 1 and 2 still produce more variance in the value estimates. Graphs showing the results for the high-difficulty Stochastic Grid World task are given in Figures 5.9–5.12.

In the high-difficulty grid world reward function #1 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. The parameters used for the selective merging algorithm were $p = 100,000$ and $f_{com} = 1024$. Recall that for this domain, the total number of features $f = 16,384$. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining parameters were $\gamma = 1.0$, $\lambda = 0.95$ and $\theta_{init} = 0$.

While the rate of convergence is very similar for all the combination functions, Figures 5.11 and 5.12 show that (for 8 or 16 agents) combination functions 1 and 2 consistently converge faster in the early stages of a run. However, this is offset by the fact that in the later stages these combination functions have a greater probability of moving away from the optimum due to an increase in the variance of the value estimates. Combination functions 3 and 4 perform much better in this later stage, remaining quite close to the optimum after the initial phase of convergence.

A direct comparison between the selective merge method and the visit-count merge method (described in Chapter 4) is shown in Figure 5.13. The 4 and 16 agent groups using the visit-count merge converge in about the same time to a policy of good quality, even though in the initial stages of a run the 16 agent group appears to converge faster. There is therefore no advantage in using more than 4 agents for this task if the visit-count merge is used. Using the selective merge method (with combination function 3) the 4 agent group converges in a similar time, but the 16 agent group allows a significant parallel speedup to be achieved, with a good policy being found in under 30s.

Pole Balancing

Using the visit-count method from Chapter 4 it was not possible to achieve a real-time speedup for the Pole-Balancing task on the cluster of workstations. However, with the selective merging method it *is* possible to achieve such a speedup, or alternatively to learn a higher quality policy in the same amount of allotted real-

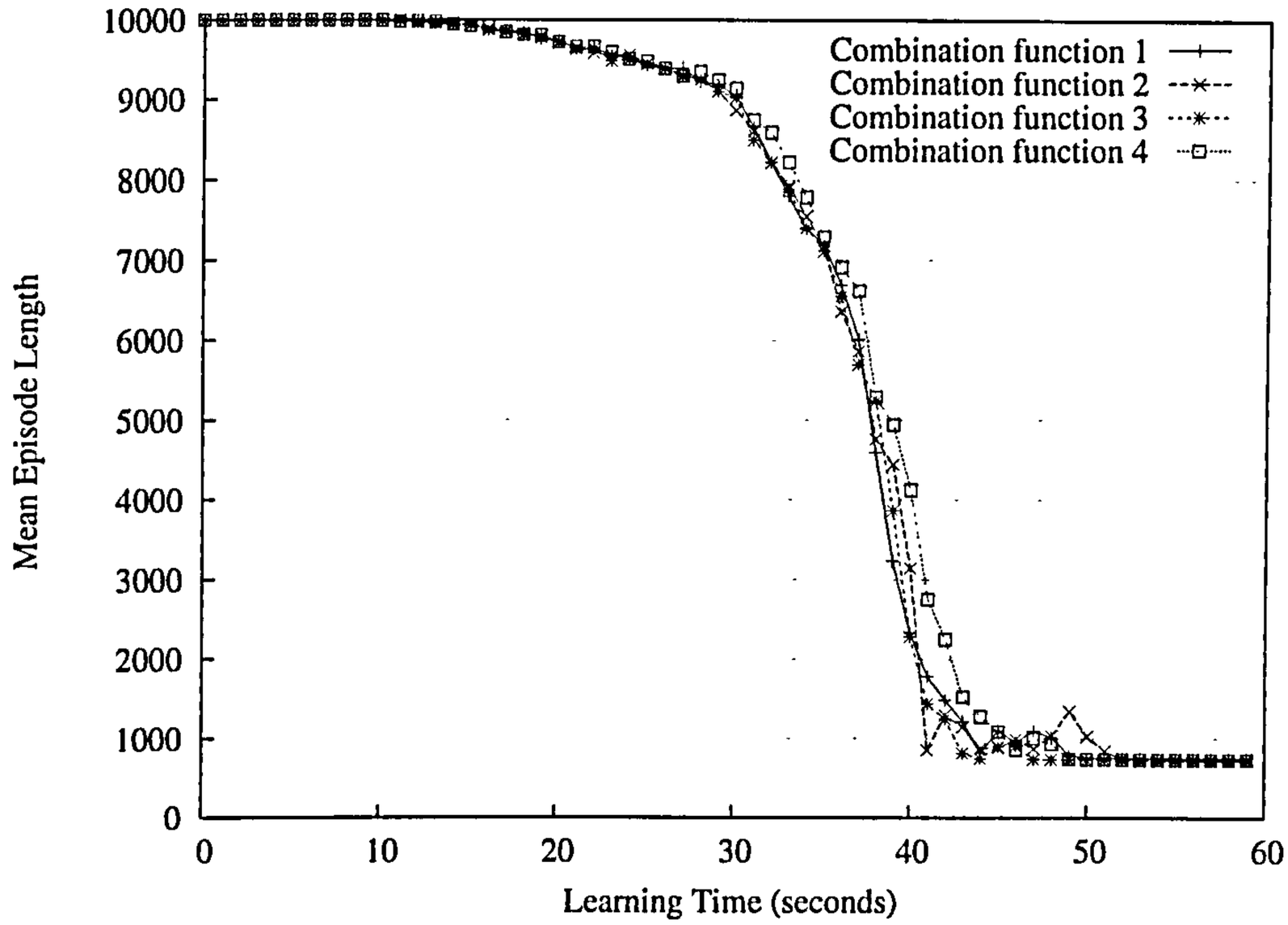


Figure 5.9: Comparing the combination functions using 2 agents in the high-difficulty Stochastic Grid World task.

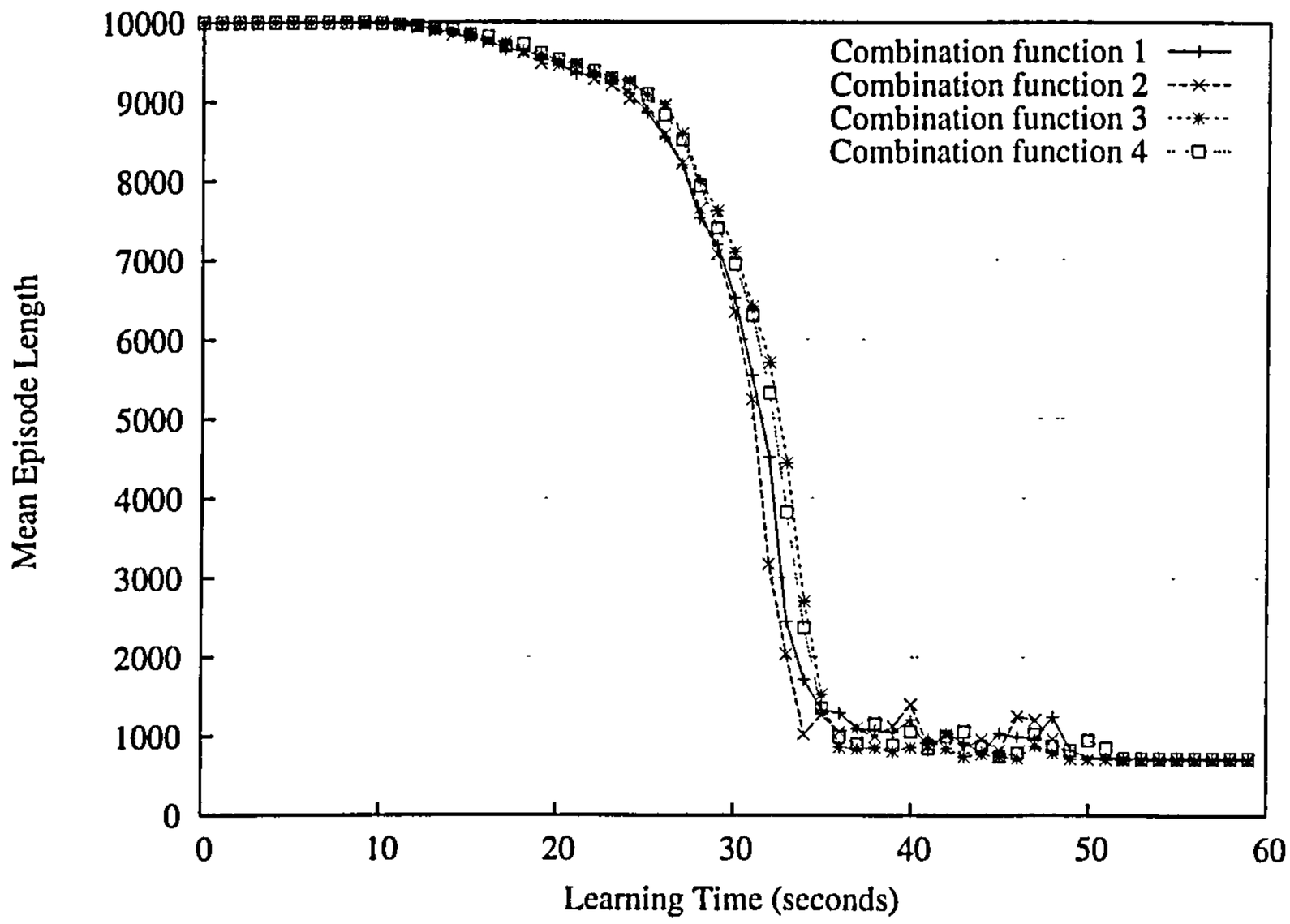


Figure 5.10: Comparing the combination functions using 4 agents in the high-difficulty Stochastic Grid World task.

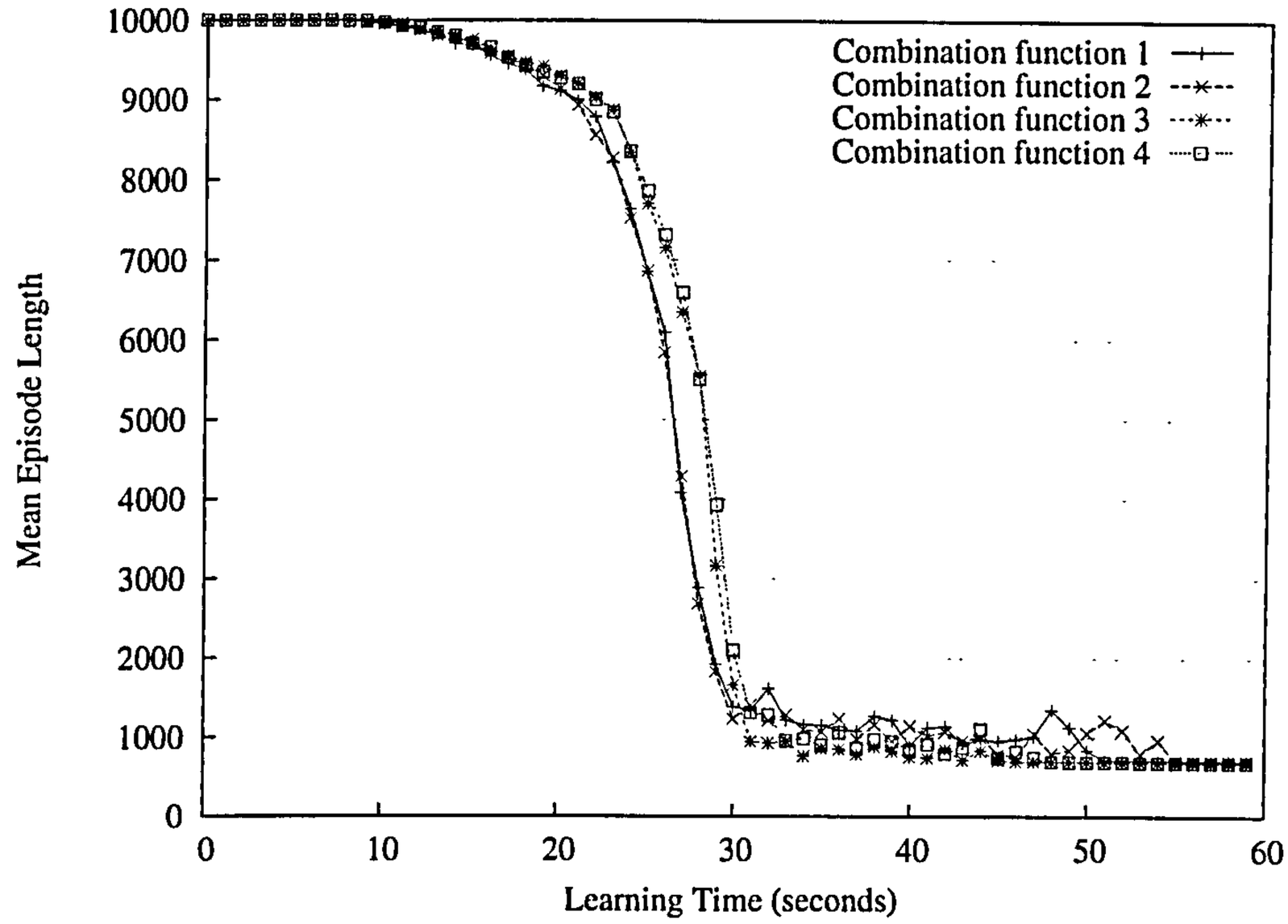


Figure 5.11: Comparing the combination functions using 8 agents in the high-difficulty Stochastic Grid World task.

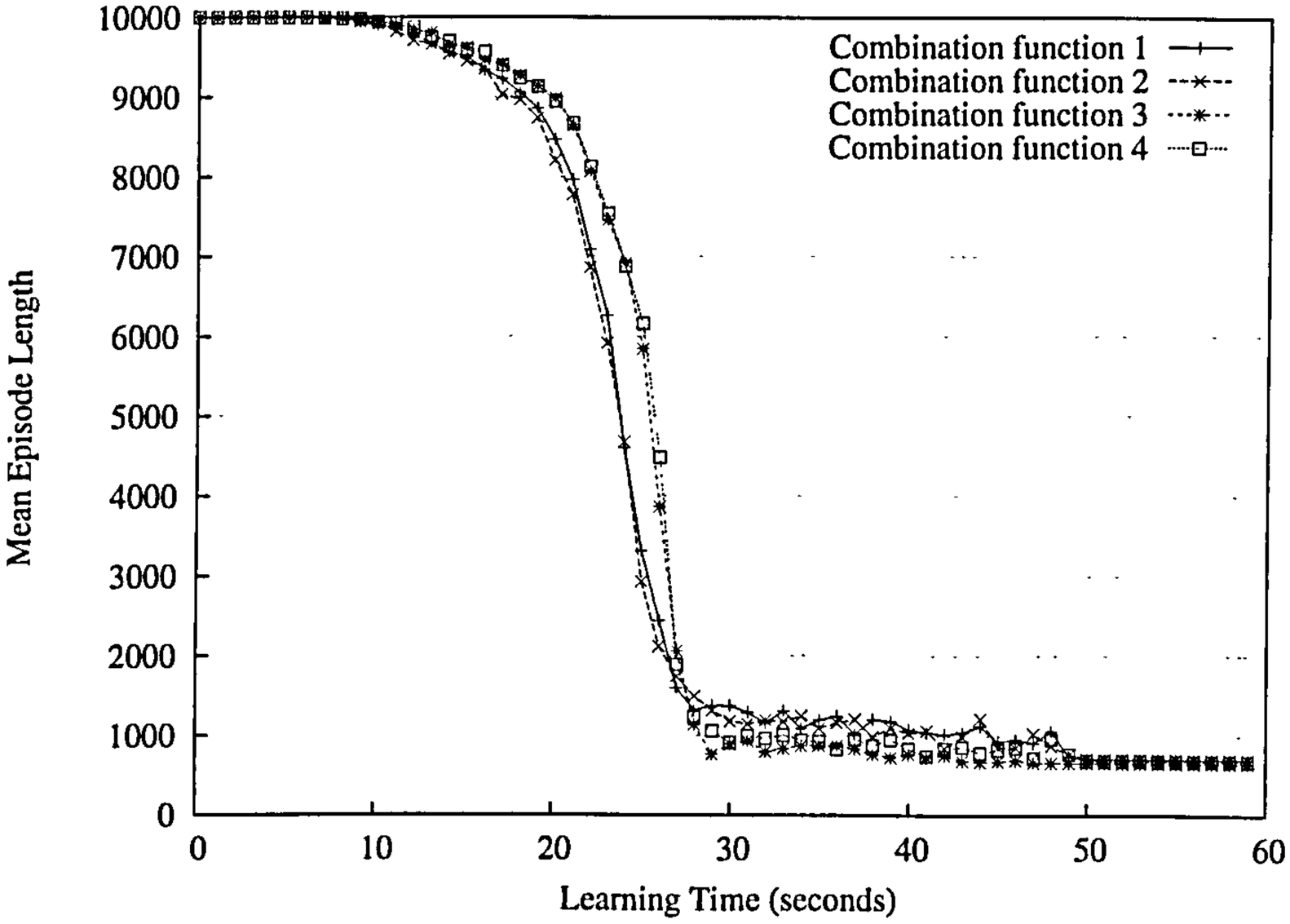


Figure 5.12: Comparing the combination functions using 16 agents in the high-difficulty Stochastic Grid World task.

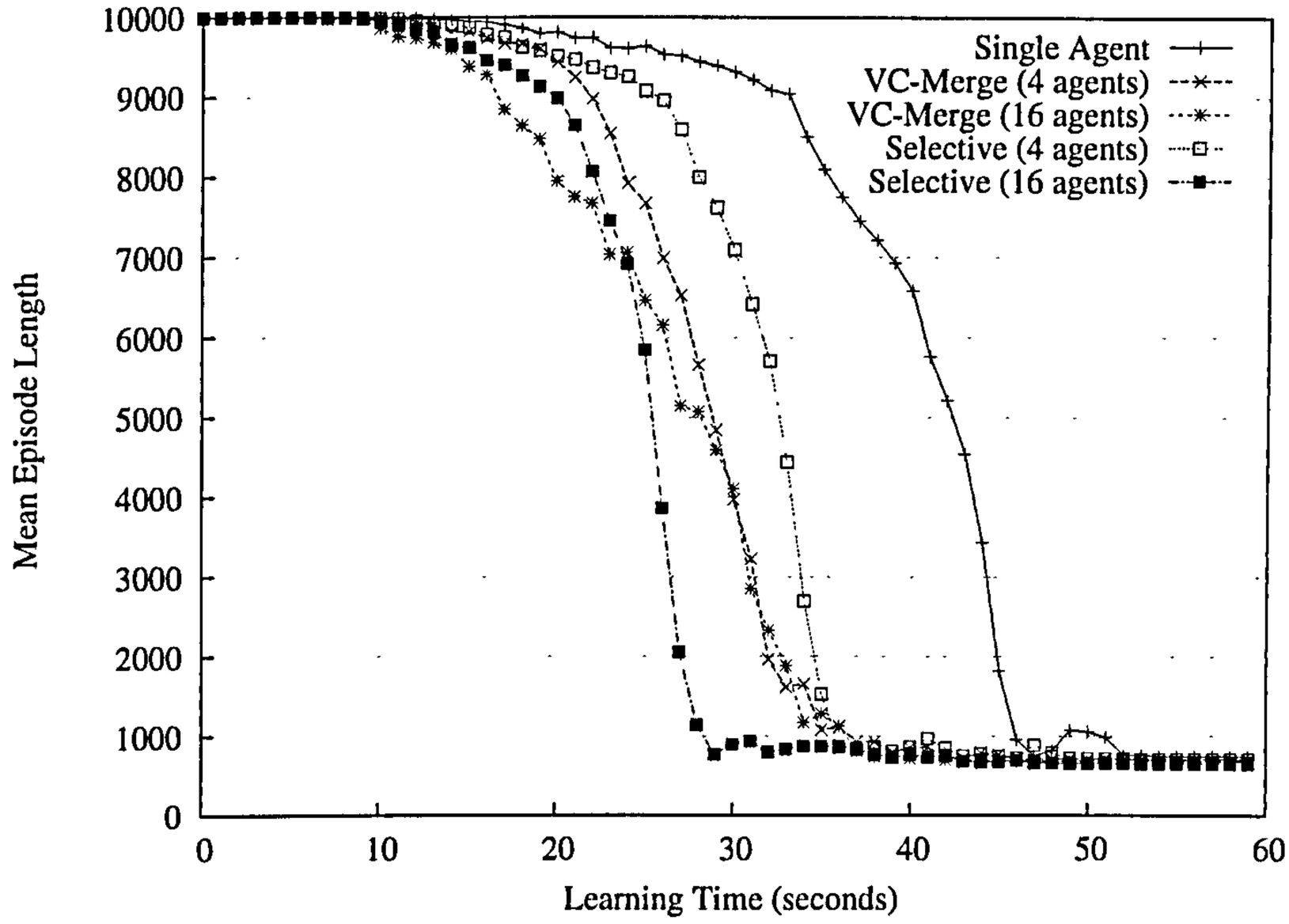


Figure 5.13: Comparing the performance of a single agent, the visit-count merge and the selective merge in the high-difficulty Stochastic Grid World task.

time. Figures 5.14–5.17 show results for groups of 2, 4, 8 and 16 agents, allowing the performance of the four combination functions to be compared for the different group sizes. The available learning time is fixed at 1.0s, and the groups of agents try to learn the highest quality policy that can be achieved in this time. In this series of experiments, reward function #1 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 20,000 steps. The parameters used for the selective merging algorithm were $p = 2000$ and $f_{com} = 128$. Recall that for this domain, the total number of features $f = 4096$. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.25$, $\epsilon_0 = 0.2$ and $t_{lim} = 0.9$. The remaining parameters were $\gamma = 0.99$, $\lambda = 0.5$ and $\theta_{init} = 0$. These settings were also used for the earlier Pole-Balancing experiment presented in Section 5.3.2.

The results for different combination functions in this task are much more varied than those obtained in the Stochastic Grid World task. There are clear differences in the quality that can be achieved in the available time. Combination functions 1 and 2 appear to perform best, with 16 agents achieving (on average) an episode length of around 16,000 in the time. Combination function 3, which uses a *mean*, performs particularly badly, with 16 agents only able to achieve an average episode length of 12,000. Combination function 4 performs a bit better for the larger numbers of agents, but using 2 agents the performance is almost

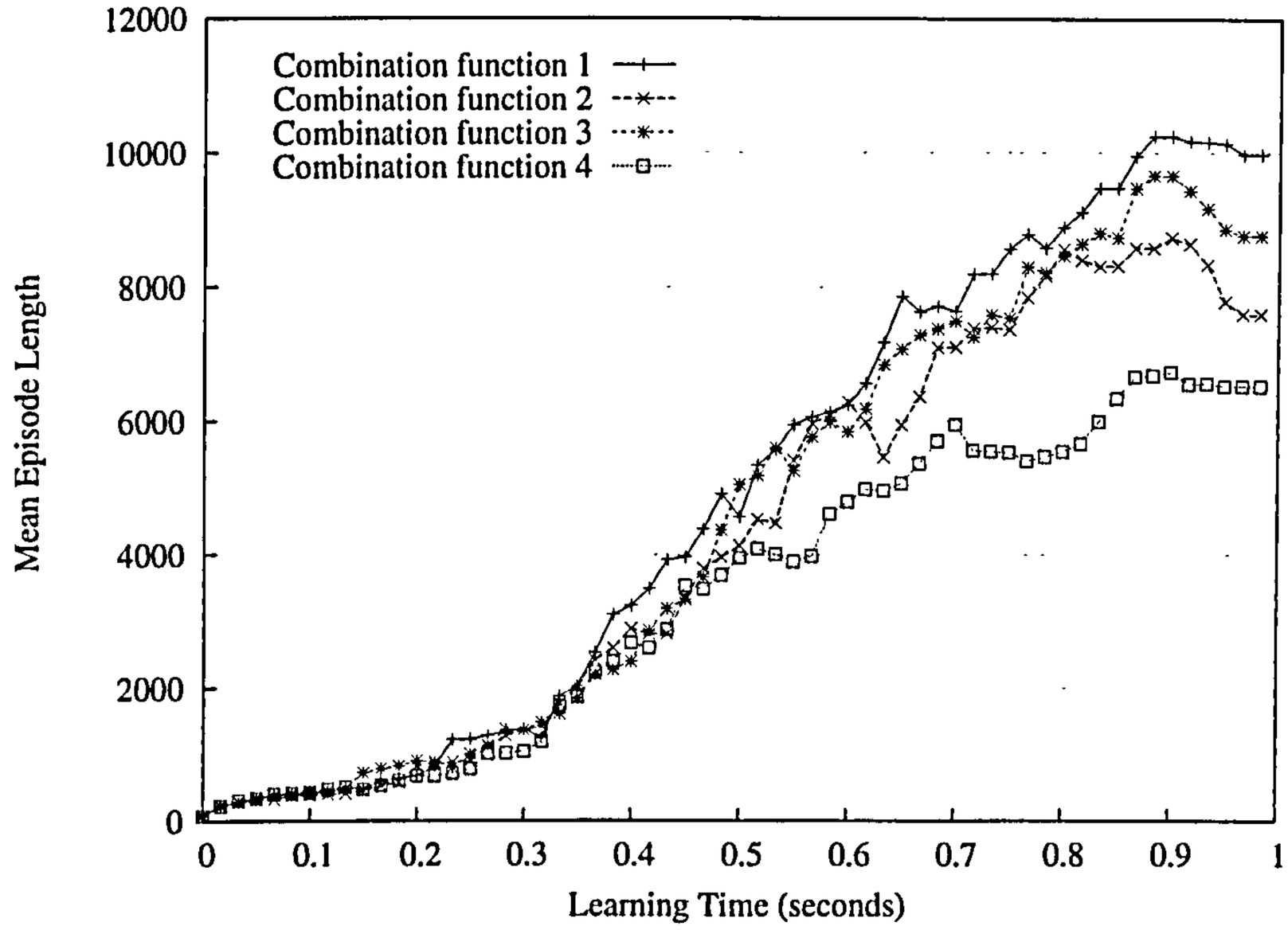


Figure 5.14: Comparing the combination functions using 2 agents in the Pole-Balancing task.

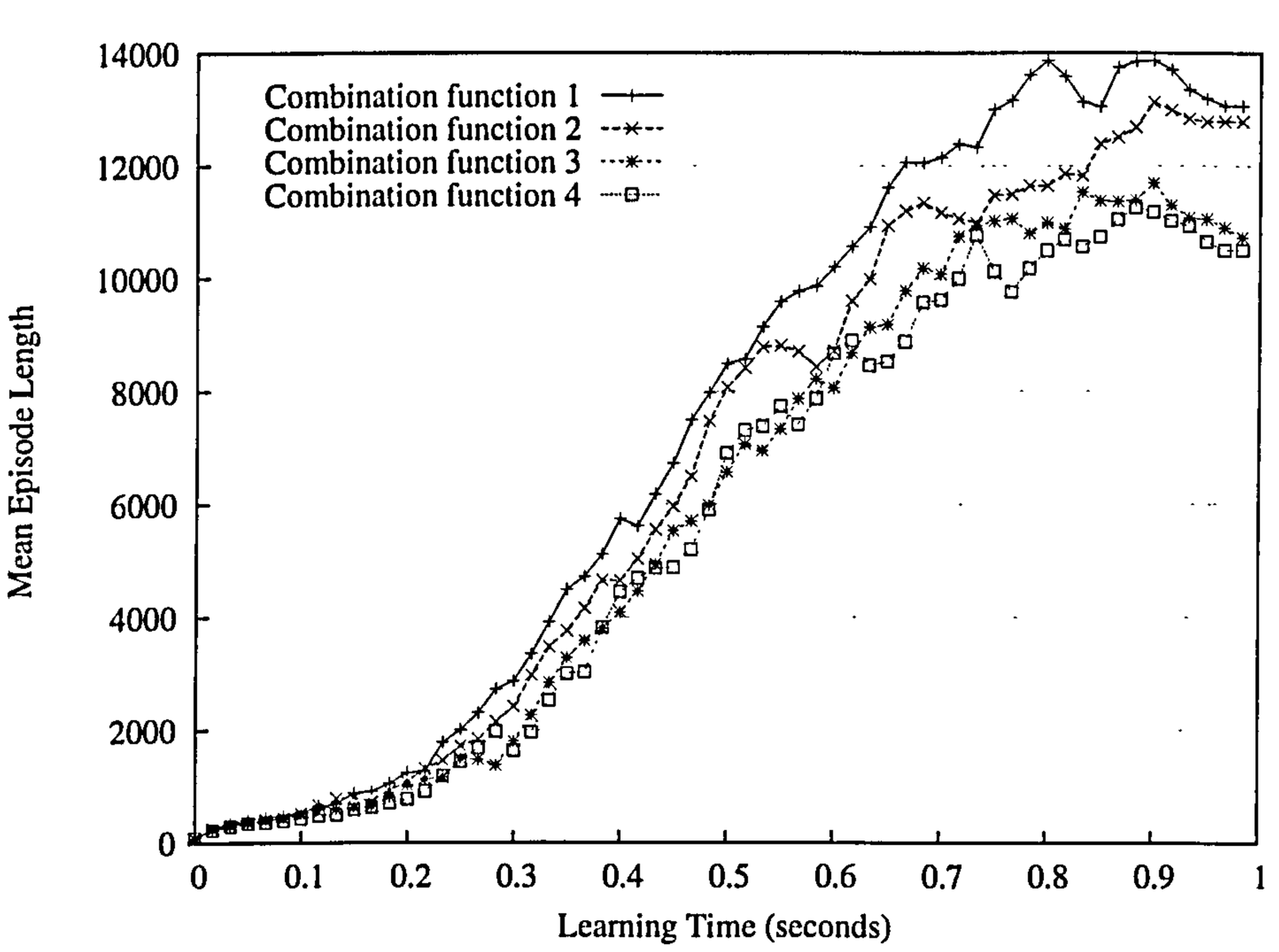


Figure 5.15: Comparing the combination functions using 4 agents in the Pole-Balancing task.

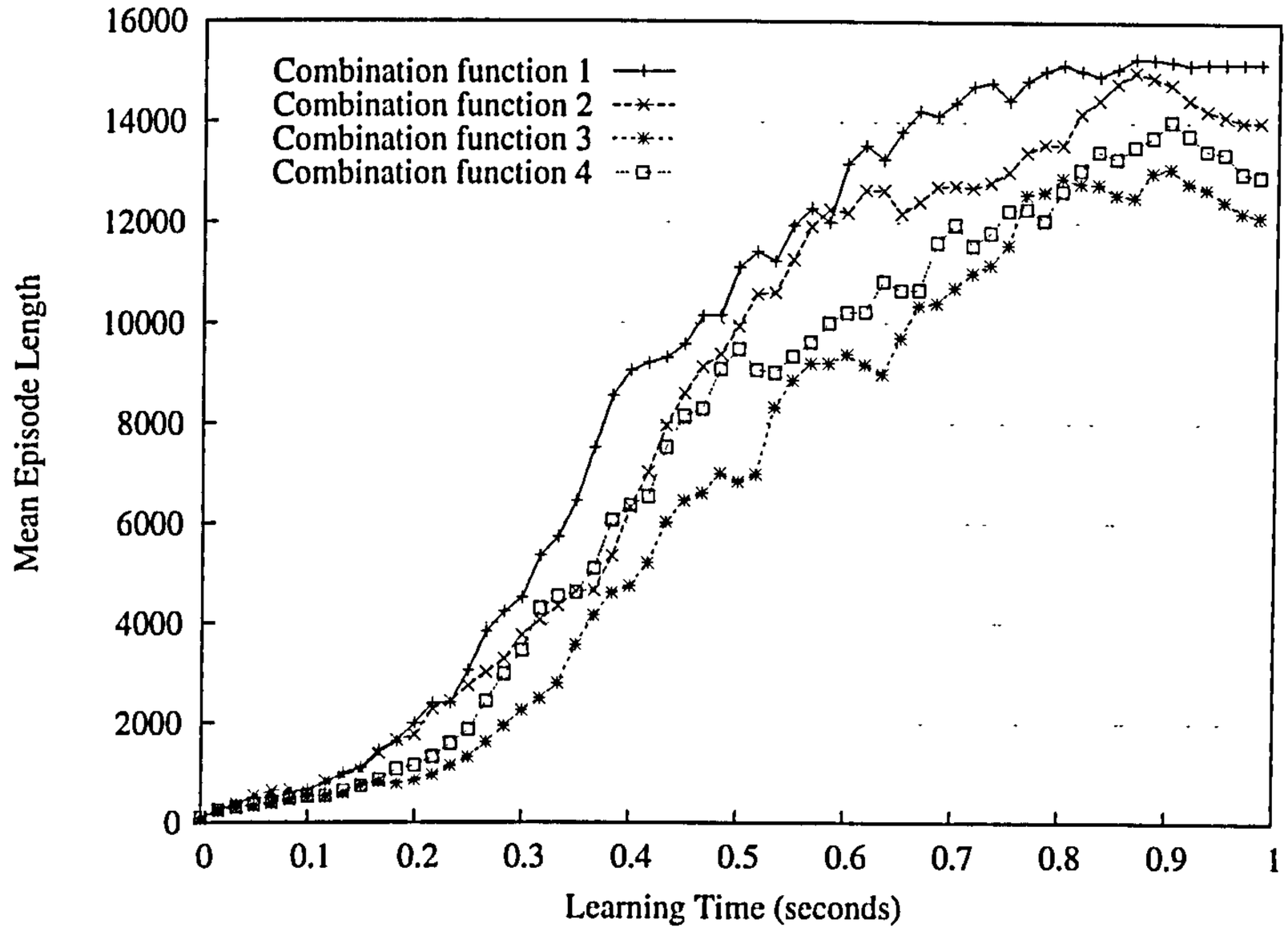


Figure 5.16: Comparing the combination functions using 8 agents in the Pole-Balancing task.

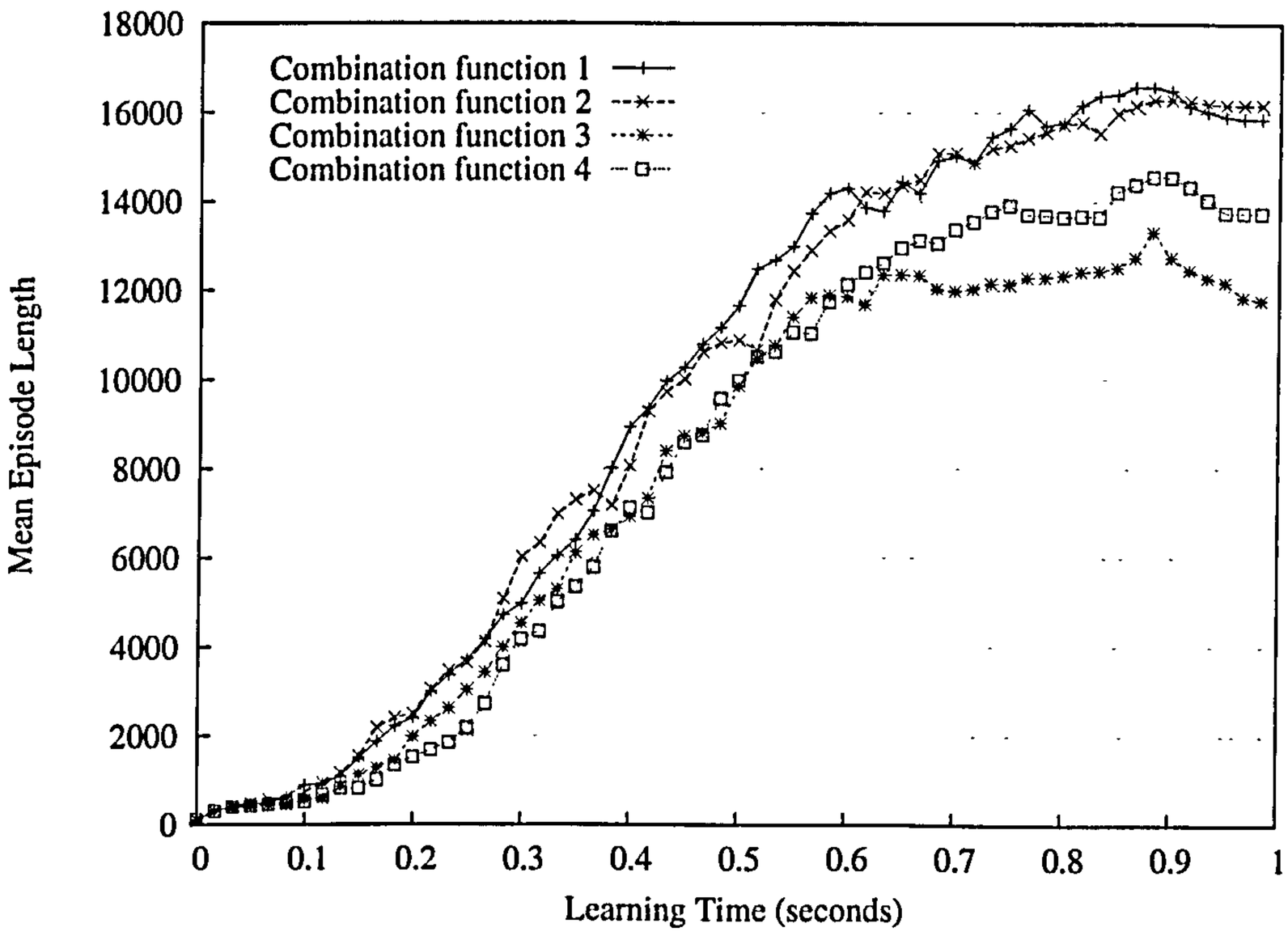


Figure 5.17: Comparing the combination functions using 16 agents in the Pole-Balancing task.

indistinguishable from that of a single agent.

The tendency of combination functions 1 and 2 to increase variance in the value estimates appears to be much less significant in the Pole-Balancing task than in the Stochastic Grid World task. This may be related to the fact that the Pole-Balancing task is a *deterministic* problem with a continuous state space. This means that variance in the value estimates now arises from the use of random (exploratory) actions and the effects of generalization error in the VFA. The poor performance of combination functions 3 and 4 show that *averaging* the changes received from a number of agents can result in reward information discovered by only one of the agents being degraded or lost. This degradation has the most significant effect when there are a large number of agents.

Since no speedup could be achieved for this domain using the visit-count merge method, the selective merge method represents a significant step forward in terms of the (comparatively) simple RL control problems considered in this thesis. While the improvements in policy quality achieved by the groups of agents are not massive, this remains an effective demonstration that parallelization will be useful for accelerating RL in a wide variety of domains, not just the largest or most complex problems.

Mountain-Car

Like in the Pole-Balancing task, it was not possible to achieve a real-time speedup in the Mountain-Car task using the visit-count merge method on the cluster of workstations. With the selective merging method it *is* possible to achieve such a speedup. Figures 5.18–5.21 show results for different numbers of agents, allowing the performance of the four combination functions to be compared.

In this series of experiments, reward function #2 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 500 steps. The parameters used for the selective merging algorithm were $p = 2000$ and $f_{com} = 128$. Recall that for this domain, the total number of features $f = 2430$. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.5$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 0.0001$. Since the policy quality over a given interval is strongly dependent on the exploration parameter ϵ, binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average quality under 145 over the set of 100 runs.

The results for the Mountain-Car task do not exhibit large variations with the use of different combination functions. Combination function 1 appears to produce the fastest convergence speed, achieving the requisite 145 average quality in around

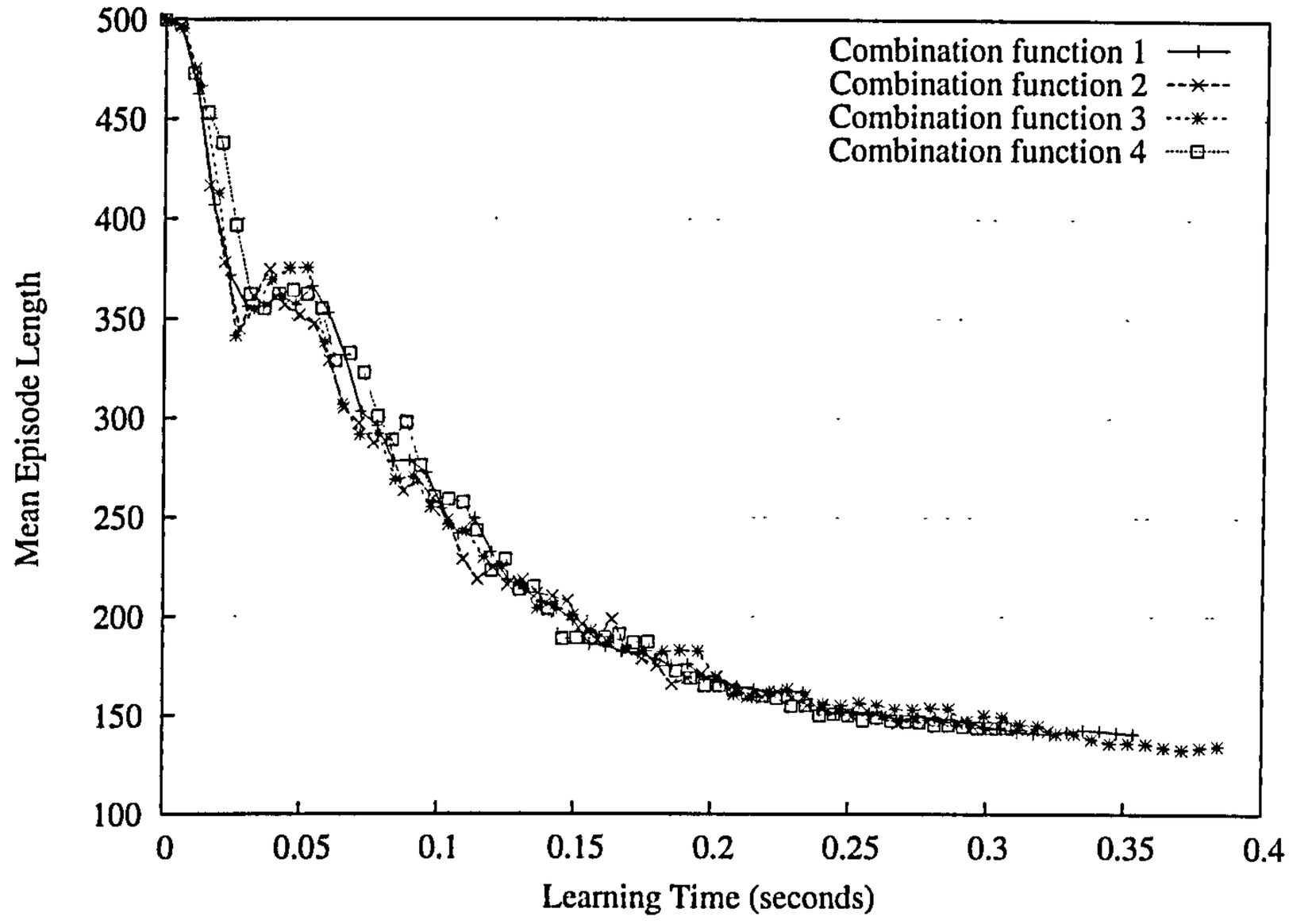


Figure 5.18: Comparing the combination functions using 2 agents in the Mountain-Car task.

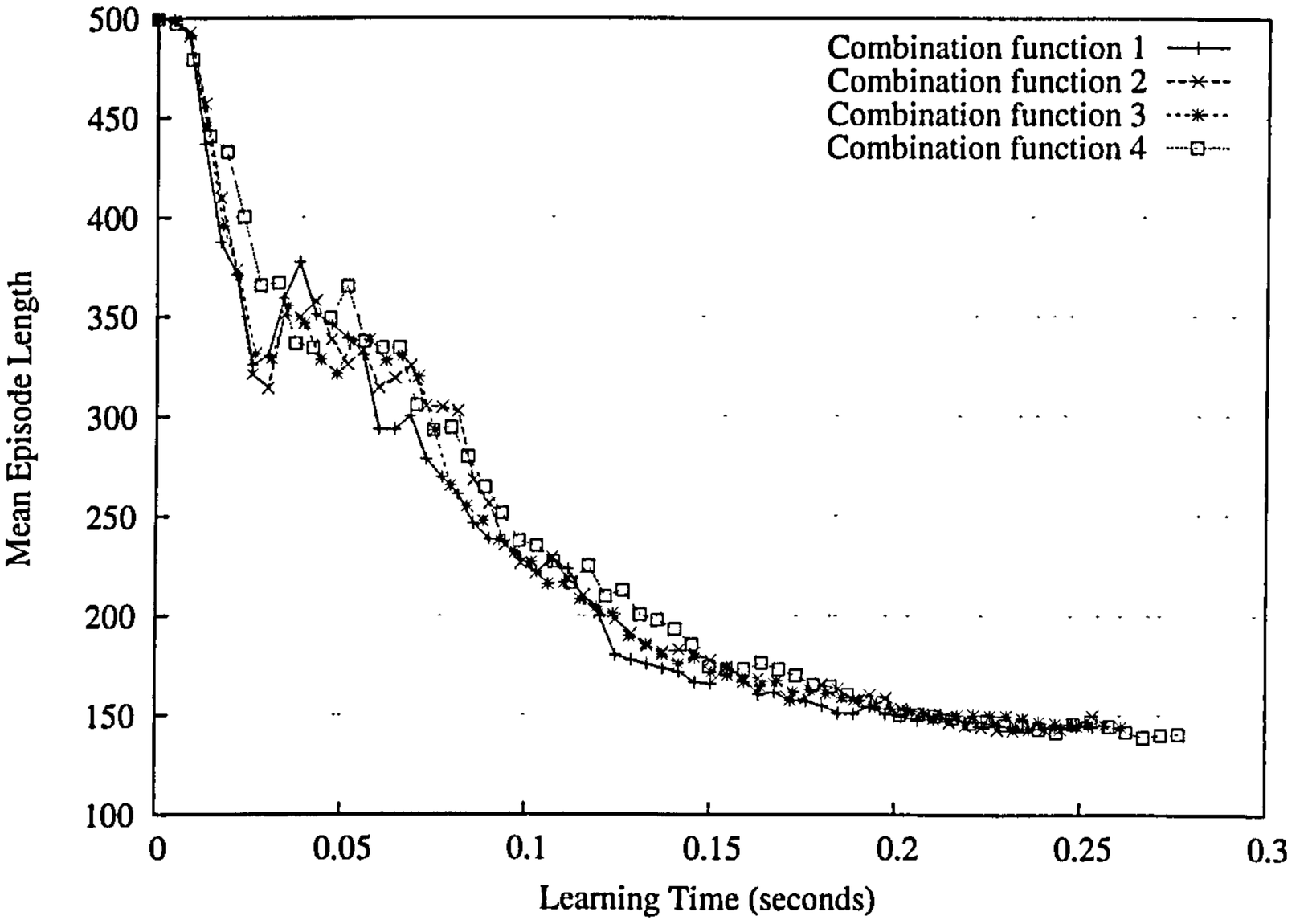


Figure 5.19: Comparing the combination functions using 4 agents in the Mountain-Car task.

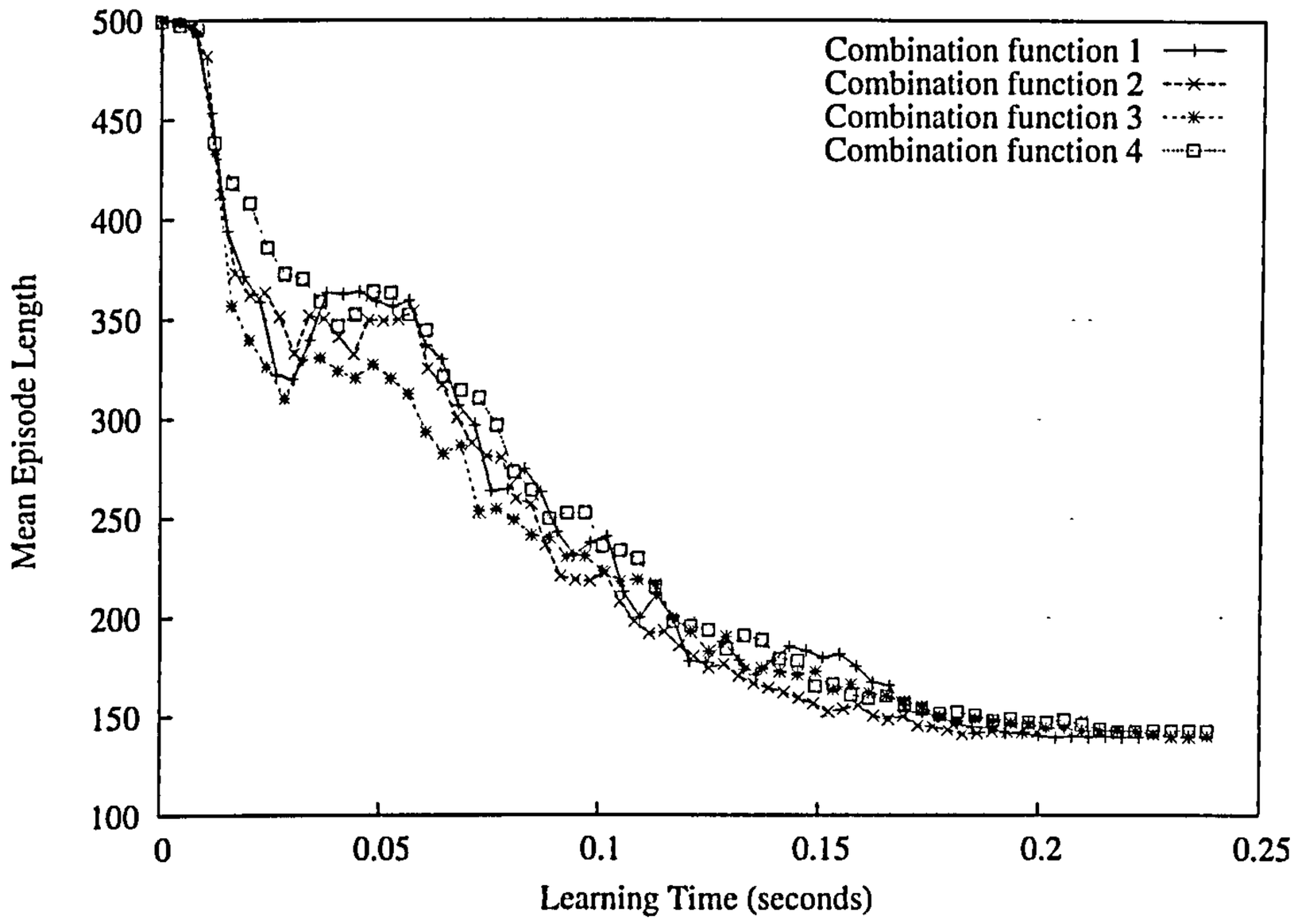


Figure 5.20: Comparing the combination functions using 8 agents in the Mountain-Car task.

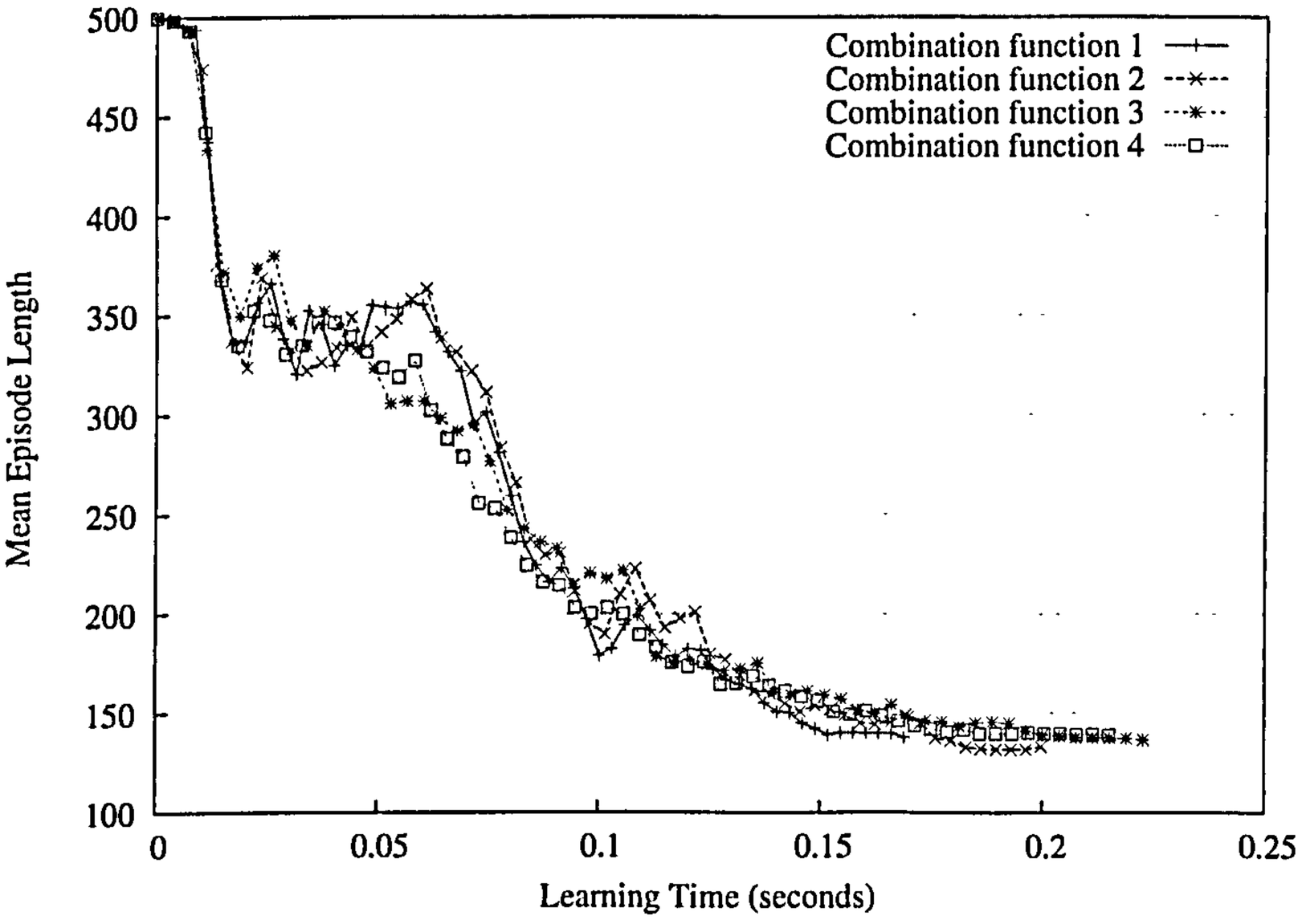


Figure 5.21: Comparing the combination functions using 16 agents in the Mountain-Car task.

0.17 seconds when 16 agents are used (see Figure 5.21). However, the use of binary search to produce these graphs introduces additional statistical uncertainty into these results. Each binary search terminates when the distance between the upper and lower bounds on the experiment time is less than 5% of the value of the lower bound. Suppose that two of the combination functions have a true expected quality of 145 at one of the boundaries tested by the search. Learners in the Mountain-Car task have a high variance in the quality achieved at the end of the experiment. Therefore the *sampled* mean at the binary search test points could be above or below the threshold of 145 on trials with different random seeds. This means that it is difficult to say with confidence that any one of the combination functions is clearly better given only the results presented above.

However, we can draw the general conclusion that with any of the combination functions we can achieve significant speedups over the single-agent's performance. This was not possible with the visit-count merge method. We may also observe that the additional speedup achieved by increasing the number of agents gradually diminishes.

Acrobot

The results for the Acrobot task were similar to those in the Mountain-Car task but with much smaller parallel speedups. A large number of VFA features are required for the Acrobot task, but relatively little experience in the domain is required to learn a high-quality policy. This means that communication costs are high in comparison to the learning time required by a single agent, making it difficult to achieve a large parallel speedup. A comparison of the performance of the four combination functions in the Acrobot task is shown in Figures 5.22–5.25.

The experiments in the Acrobot domain used reward function #1, with the results being averaged over 100 runs. Episodes were terminated if they reached 600 steps. The parameters used for the selective merging algorithm were $p = 1000$ and $f_{com} = 128$. Recall that for this domain, the total number of features $f = 18432$. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.1$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining parameters were $\gamma = 1.0$, $\lambda = 0.9$ and $\theta_{init} = 0$. Since the policy quality over a given interval is strongly dependent on the exploration parameter ϵ, binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average quality under 140 over the set of 100 runs.

As in the other domains evaluated here, the difference in performance between the combination functions is relatively small. However, as the number of agents in the group is increased, there appears to be a significant advantage in using com-

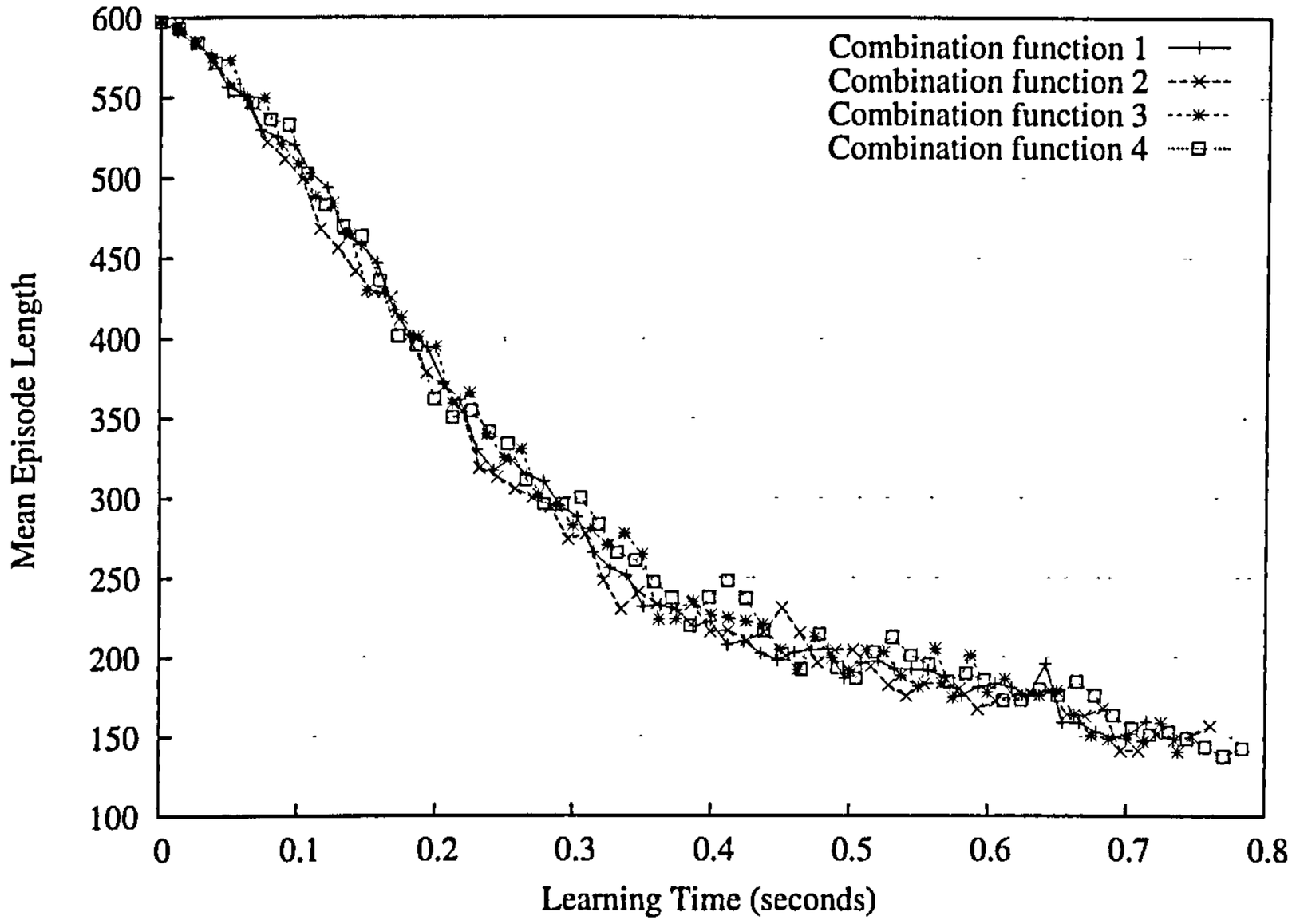


Figure 5.22: Comparing the combination functions using 2 agents in the Acrobot task.

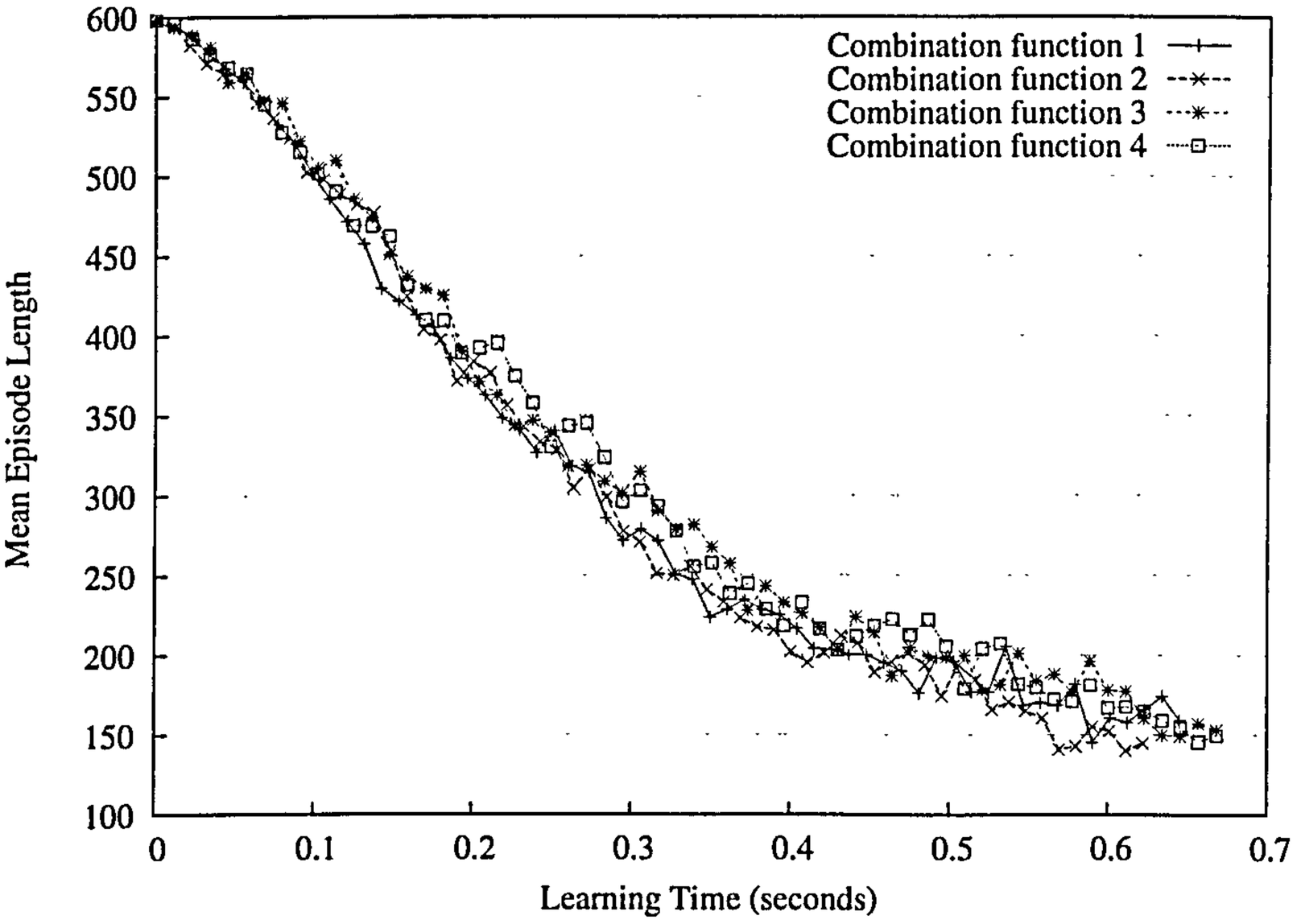


Figure 5.23: Comparing the combination functions using 4 agents in the Acrobot task.

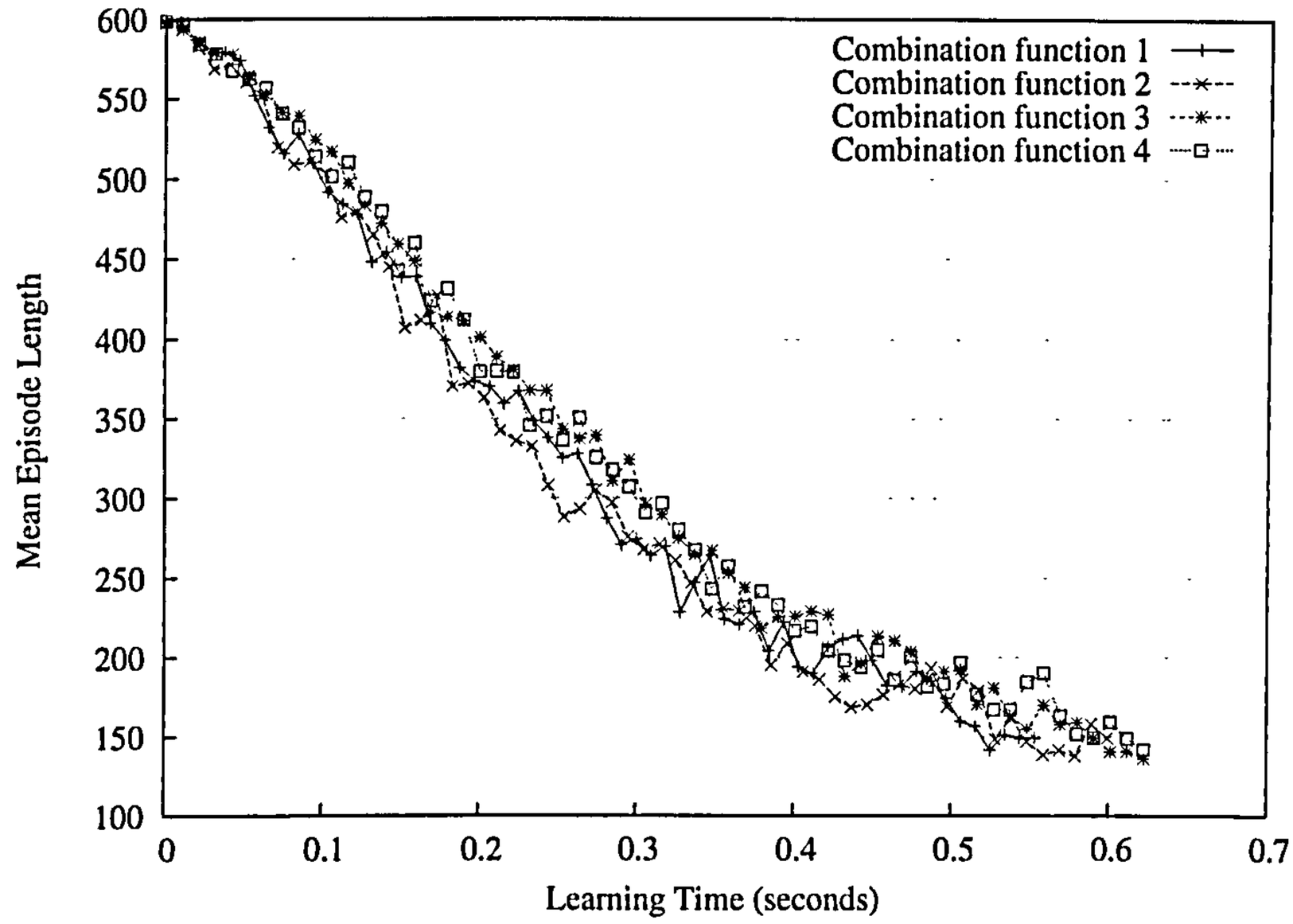


Figure 5.24: Comparing the combination functions using 8 agents in the Acrobot task.

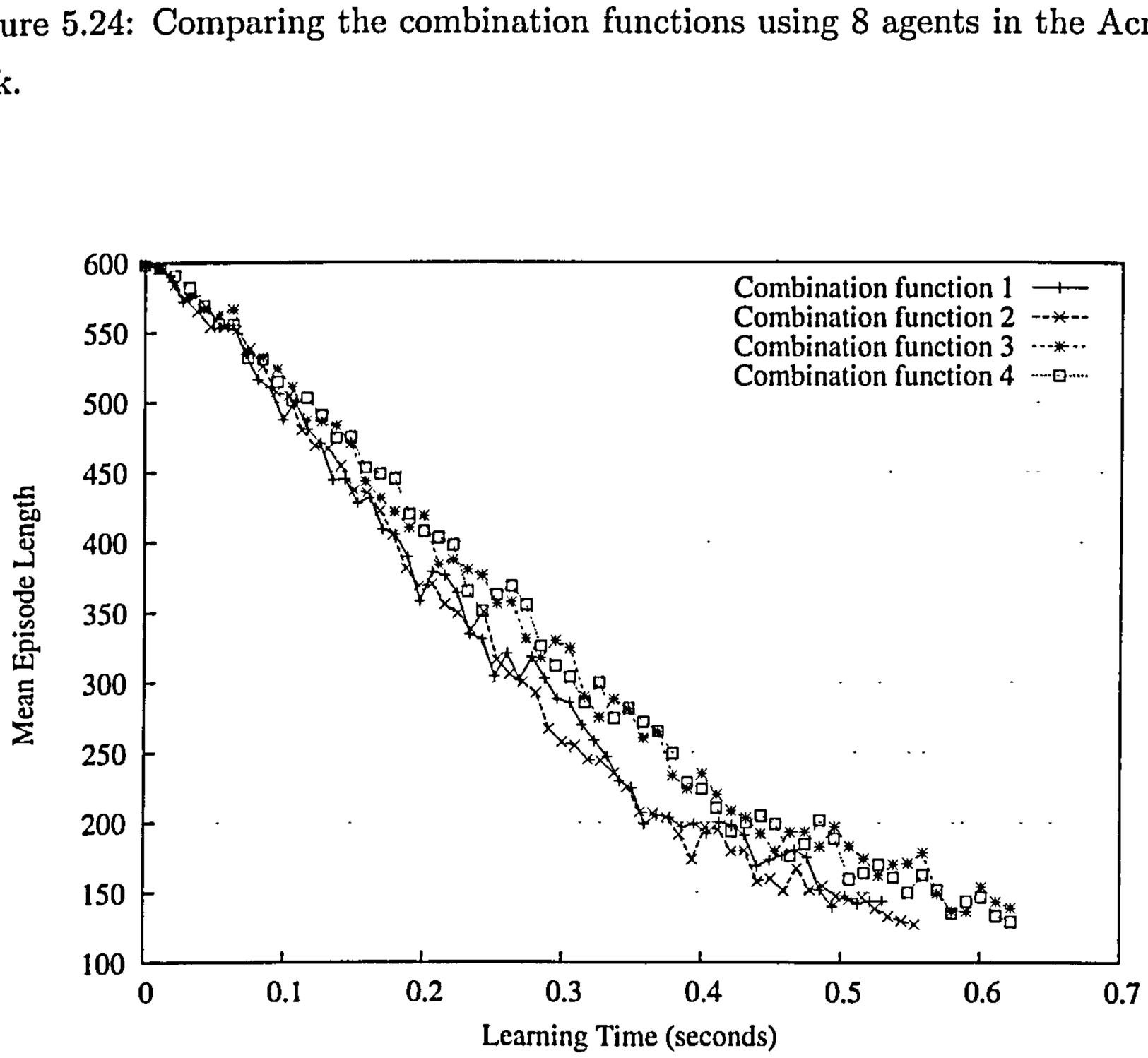


Figure 5.25: Comparing the combination functions using 16 agents in the Acrobot task.

bination functions 1 and 2. This suggests that the averaging effect of combination functions 3 and 4 is having a detrimental effect on performance as the number of agents is increased.

Summary of Evaluation

A qualitative summary of the results presented in this section is shown in Table 5.1. This table lists the best performing combination function(s) from each of the graphs shown in Figures 5.4–5.25. The best performing combination function is the one which makes the most rapid progress towards a high quality policy over the course of a parallel run, without compromising the final quality of the policy at the end of the run.

Domain	Number of agents			
	2	4	8	16
Grid world (low-difficulty)	*	#1/#2	#1/#2	#3/#4
Grid world (high-difficulty)	*	#1/#2	#1/#2	#1/#2
Pole-Balancing	#1	#1	#1	#1/#2
Mountain-Car	#4	#1/#2	#2	#1
Acrobot	#1	#2	#1	#1

Table 5.1: Lists the best performing combination function(s) for each possible domain and number of agents used. A star in the table indicates that a difference in performance could not be discerned from the relevant graph.

A clear winner from the combination functions does not emerge from the summary in Table 5.1. Combination function #1 appears most frequently as the best performer, followed closely by #2. In all of the cases enumerated here, combination functions #1 and #2 produce the best (or equal best) *initial* convergence rate. However, both of these combination functions *increase the variance* in the value function, making it more likely that the agents could move *away from* the optimal policy once they get close to it. This is most clear in Figure 5.7, where as ϵ and α are gradually decayed, the agents using combination functions #1 and #2 remain furthest from the optimum, and in one run out of the ten total runs the agents diverge completely from the optimum as ϵ and α both approach zero.

Therefore, in highly stochastic domains it may be preferable to favour combination functions #3 and #4, which are more stable in these circumstances. However, these more stable combination functions do degrade the overall performance to some extent. This is particularly clear in the Pole-Balancing and Acrobot domains.

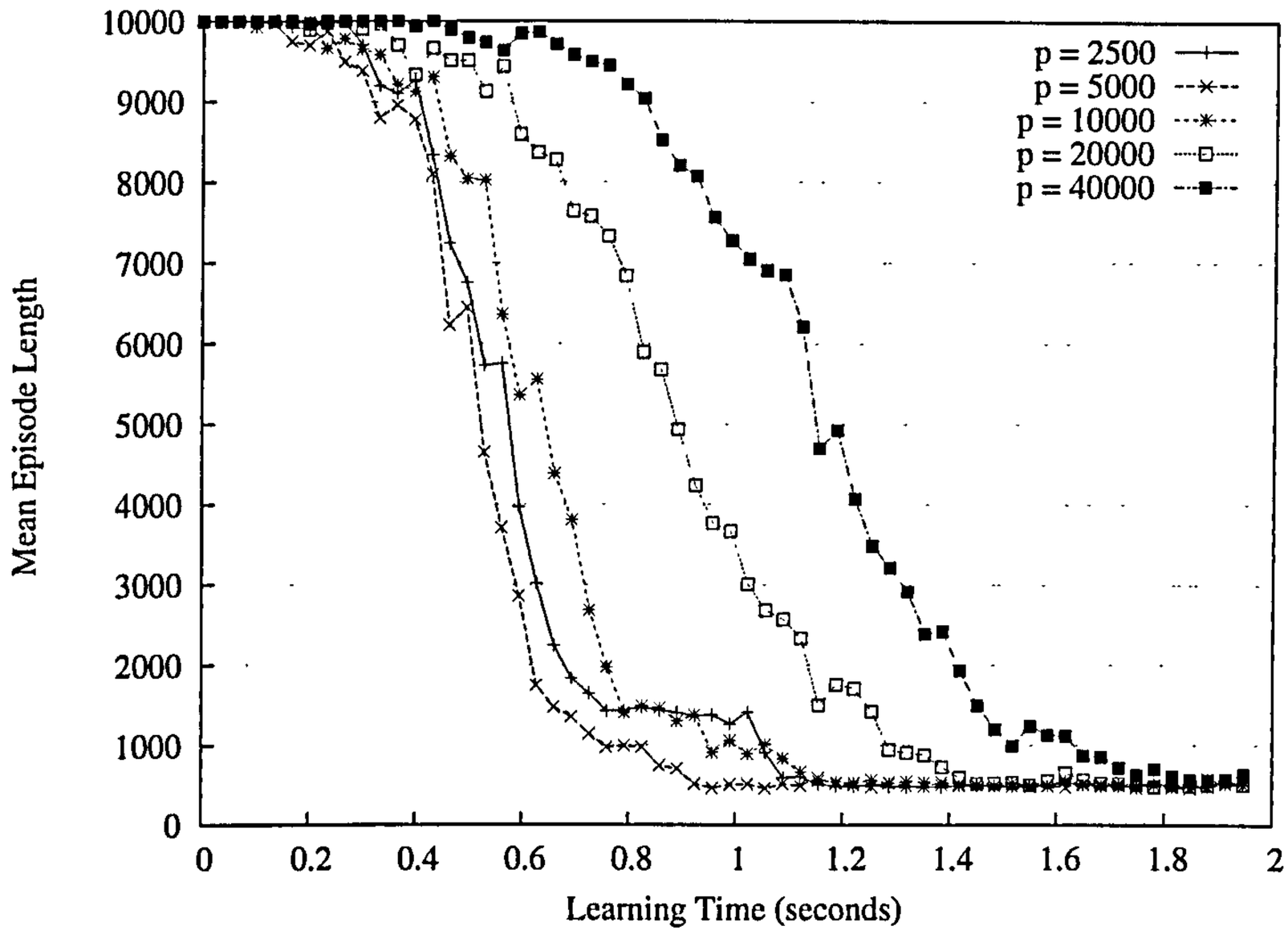


Figure 5.26: Experiment using 16 agents to solve the low-difficulty Stochastic Grid World task using selective merging (combination function 4). Several different values are tried for the merge period p.

5.5 Varying the Merge Period and Message Size

In the evaluation above, the parameter values for the merge period (p) and the number of weight changes per message (f_{com}) were selected using a trial and error approach. In this section I will conduct a closer examination of the effect these parameter choices have on the speedup that can be achieved. This study is similar to the one previously carried out (in Section 4.8) for the visit-count merge method. However, the *joint* effects of the p and f_{com} parameters on the performance of the selective merge method require further investigation.

The merge period parameter p has a similar purpose in this method as in the visit-count merge method (see Section 4.8). It controls *how often* the agents are able to share information, and a good choice for p represents a *trade-off* between the increase in sample efficiency as the agents share more often and the corresponding increase in communication overhead. A graph showing results for 16 agents using a variety of different values for p in the low-difficulty Stochastic Grid World task is shown in Figure 5.26. The selective merge method is used with combination function 4. The settings used for this experiment are the same as those given in Section 5.4, with the value of f_{com} remaining fixed at 256. The optimum choice of p for this particular number of agents appears to be a value close to 5000.

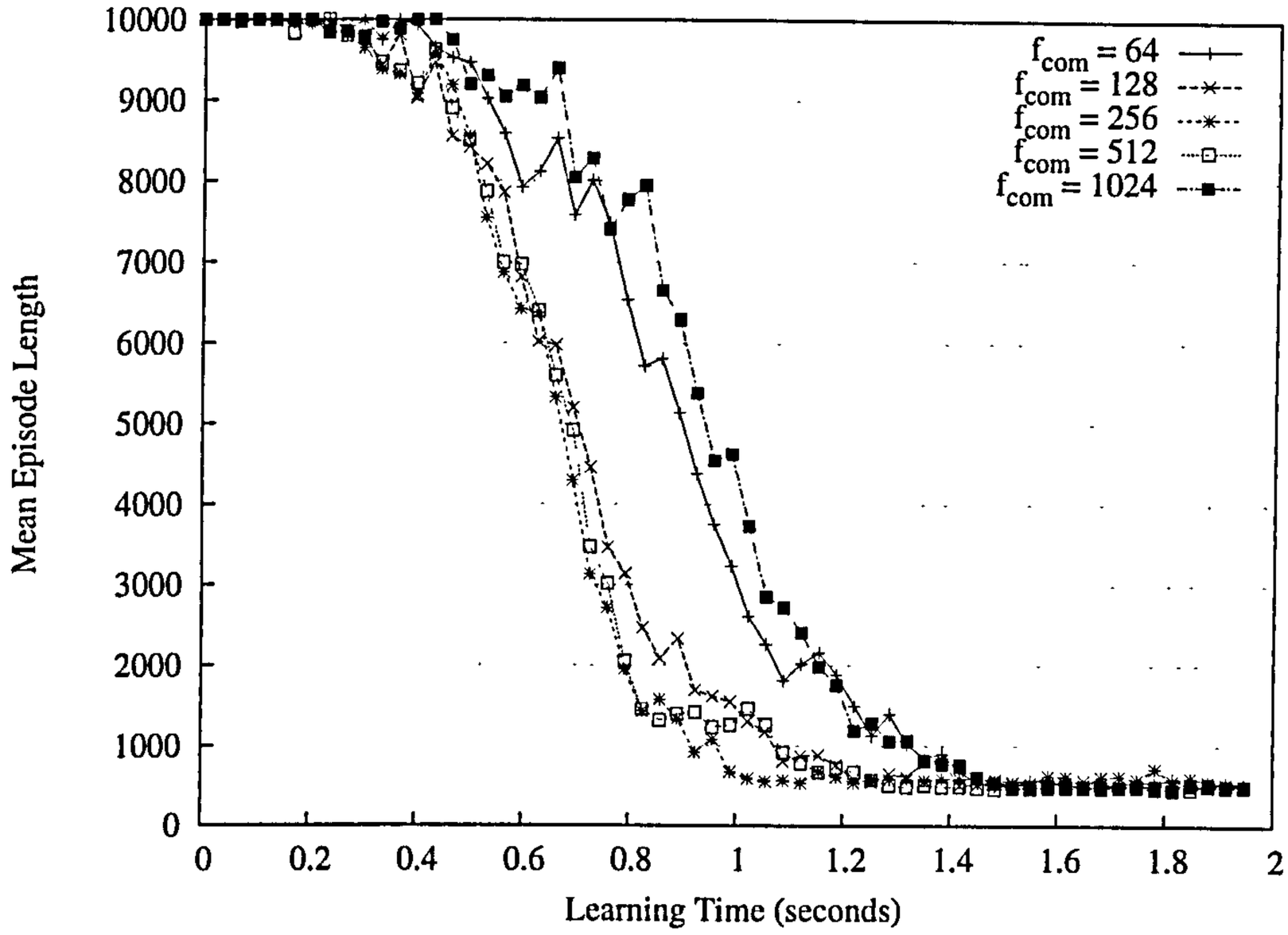


Figure 5.27: Experiment using 16 agents to solve the low-difficulty Stochastic Grid World task using selective merging (combination function 4). Several different values are tried for the number of communicated changes f_{com}.

The number of weight changes per message f_{com} controls *how much* information is transmitted by each of the agents during a single merge operation. Together with the merge period p, these two parameters control the *rate of information exchange* between the agents. The trade-off in the choice of parameter f_{com} is of a similar nature to the trade-off in the choice of p. As the value of f_{com} is increased, more information can be exchanged between the agents during each merge operation, allowing the resulting value function approximation to be a better combination of the knowledge of the whole group. This means that fewer simulation steps will be required to reach a near-optimal policy. However, increasing the value of f_{com} also increases the overall network bandwidth required, and hence the real-time required to complete the merge operation. A graph showing results for 16 agents using a variety of different values for f_{com} in the low-difficulty Stochastic Grid World task is shown in Figure 5.27. The settings are the same as those above, with the value of p being fixed at 10,000 (the same value used in the evaluation of Section 5.4). The optimum choice of f_{com} for this particular number of agents appears to be a value close to 256.

In the final experiment in this section, we examined the effect of varying the p and f_{com} parameters *at the same time.* The number of weight changes per message (f_{com}) is constrained so that it varies in *direct proportion* to the merge period. In

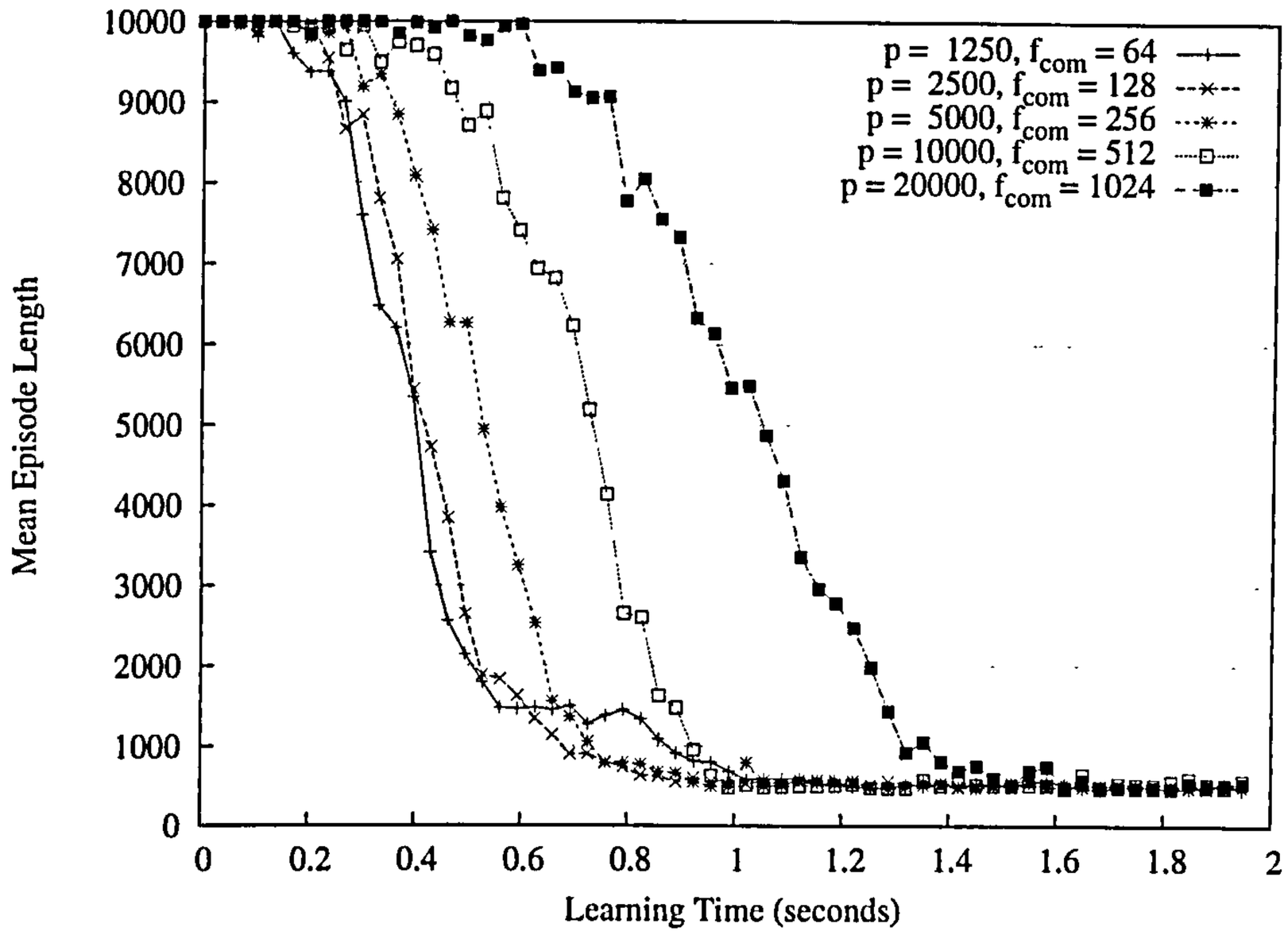


Figure 5.28: Experiment using 16 agents to solve the low-difficulty Stochastic Grid World task using selective merging (combination function 4). Each time the merge period p is doubled, f_{com} is also doubled.

this way the net rate of information exchange remains the same across all values of p. The choice of parameter p now primarily determines the size of the "chunks" into which the constant communication rate is divided. A graph showing results for 16 agents with various values of p and f_{com} in the low-difficulty Stochastic Grid World task is shown in Figure 5.28. The settings (other than p and f_{com}) used for this experiment are the same as those used above. The optimum values for p and f_{com} under these constraints are around $p = 2500$ and $f_{com} = 128$.

The trade-off in the choice of p is now affected by several different factors. The first is the *overhead per message sent*, i.e. the real time consumed by an agent broadcasting a message to the group as the number of changes per message $f_{com} \rightarrow 0$. This is affected by factors such as the *size of the headers* used by the MPICH library and the underlying TCP/IP protocol stack, and the *latency* inherent in sending a message between two agents over the communication network. However, the most significant contribution to the message overhead in our experiments was the *time required to rank the weights* according to the magnitude of the weight changes that had recently taken place. In other words, the properties of the underlying network stack and transmission medium were not as significant as the time required to identify each set of weight changes for broadcast.

The second factor affecting the trade-off is the likelihood of new reward infor-

mation being discovered independently by two or more agents before a merge can take place (replication of effort). If we use a large merge period while keeping the rate of information exchange constant, we will be able to transmit weight change information about a much larger set of weights than if we used a small merge period. However, there is a danger of the merge period becoming so long that many of the agents will discover the same changes between merges. This means that even though more weight changes can be transmitted, the usefulness of this information will be significantly degraded, as it may already be known by many in the group.

The third factor affecting the trade-off relates to the *diversity* of the sets of weight changes transmitted by the agents. In the RL problem there may be some states with a high probability of being visited by all the agents, or rewards that are particularly easy to discover from the initial state. If the transition and reward functions are stochastic, then many of the weight changes made by the agents will be associated with these highly-visited states. What we are more interested in are the rarely visited areas of the state space, with rewards that are difficult to find. If only a small set of changes can be transmitted during each merge, then the changes will be dominated by the highly-visited states, regardless of how small the merge period is. This means that if the "chunks" of information are too small, the diversity of the sets of changes will be too low, and there will be greater replication of effort in the group.

With such a large number of factors to consider, determining the optimum choices for p and f_{com} analytically is unlikely to be feasible. However, using a series of experiments (such as those reported in Figures 5.26–5.28) to find parameter values that are close to optimal is too time consuming to be practical. To improve the practicality of this method for speeding up learning on real parallel systems, a heuristic approach for selecting parameters p and f_{com} is likely to be required, although we have not identified such a heuristic during the course of this work.

5.6 Summary and Conclusions

The following material has been presented in this chapter:

- Motivation for the use of *selective merging* to eliminate much of the redundant information transmitted between the agents by the merging method of Chapter 4.
- A mechanism for ranking the weights of the VFA by the magnitude of recent accumulated change for each weight.
- A description of a selective merge operation, where each agent broadcasts a

message containing changes to the f_{com} highest ranked weights. This merge operation takes place after every p simulation steps.

- A description of the notion of a combination function, which is necessary to combine changes to the same weight received from different agents.
- The definitions of four candidate combination functions.
- An evaluation of four instances of the selective merging method, one based each of the combination functions. Each of the instances was evaluated in all of the example RL problems defined in Section 4.3.1. The reported results were generated using the implementation on the cluster of workstations.
- An analysis of the effect of parameters p and f_{com} on the parallel speedup which can be obtained using the selective merging method with 16 agents.

From this material we can draw the following conclusions:

- *Selective merging* can be used to achieve parallel speedups in a range of RL problem domains, even though much less information is exchanged between the agents compared to the merging method of Chapter 4.
- On the cluster of workstations, selective merging consistently outperforms the visit-count merge method of Chapter 4. In particular, selective merging achieves real-time speedups in each of the three control problems defined in Section 4.3.1. The visit-count merge method could not achieve any speedup in these domains.
- It was not possible to select a *combination function* $g(C)$ to produce the best performance of selective merging in all the evaluation domains. One reason for this may be that there is a trade-off between quickly propagating rewards through the VFA and reducing the variance in the value estimates. In spite of this, in most situations any of the combination functions will work fairly well.
- Selecting appropriate values for parameters p and f_{com} is vital for achieving good performance using the selective merging method. While the performance is not sensitive to small variations in these parameters, it is important that the parameters do not differ too greatly from their optimum values. There is not currently an analytic or heuristic method for determining the optima, so we have to use trial and error to select these parameters.

In Chapter 6 I will present an *asynchronous* parallel RL method which is based on the selective merging method presented in this chapter, but which eliminates the synchronization penalty exhibited by the latter method. This will allow greater parallel speedups to be achieved in the five evaluation domains.

Chapter 6

Asynchronous Merging

In the previous chapter, a parallel RL method was presented which was based on agents communicating *recent changes* to the weights of their value function approximators (VFAs). This method was described as *selective* merging, since each agent selects for broadcast only a small subset of changes which are large in magnitude. This method achieved parallel speedups (of varying size) in all of the evaluation domains. This was possible because the communication overhead of selective merging was many times smaller than that of the original merging method described in Chapter 4.

In this chapter I will present another method which is *selective* in the way described in the previous chapter, but which is also *asynchronous* in character. The selective merging method of Chapter 5 has distinct *computation* and *communication* phases. In the communication phase, messages from all the other agents must be received before the VFA can be updated and the next computation phase can begin. In contrast, the method in this chapter involves agents updating the VFA *as each message arrives* and performing additional computation (learning) in between the messages. Eliminating the synchronization penalty in this way means that the asynchronous method can achieve even greater speedups than those reported in Chapter 5.

The chapter begins with an examination of how the performance of selective merging can be improved with the use of asynchronous message passing. The basic procedure for asynchronous merging is then given. Three variants of the basic procedure are defined, each of which processes the incoming messages in a different way. We will also examine the issue of *when* each agent should schedule its communications, since agents are no longer restricted to simultaneous broadcast in the communication phase. The relative performances of the three variants of the asynchronous merge method are examined in each of the five evaluation domains in Section 6.3. This is followed in Section 6.4 by an evaluation comparing

the performance of the asynchronous merge method to the selective method from Chapter 5. Finally, in Section 6.5, the performance of the asynchronous method is compared to an alternative asynchronous method which uses messages containing absolute weight values and not weight changes.

6.1 The Benefits of Asynchronous Message Passing

At a high level, the selective merging method described in Chapter 5 is *synchronous* in character. The method is divided into alternate *computation* and *communication* phases. During the computation phase, each agent learns from a sequence of p actions taken in the local (simulated) environment. No synchronization is necessary during this phase, since the agents can operate entirely independently. However, during a communication phase each agent must broadcast a message to the group and wait to receive a message from every other agent. Once all the messages have arrived, the VFA can be updated and the next computation phase can begin.

The synchronous nature of the selective merging method results in a number of effects which can degrade performance:

- At the start of each communication phase, all the agents broadcast their messages simultaneously. This results in the interconnection network becoming congested, and overall the messages take longer to travel between the agents.

- If the interconnection network is slow (perhaps because of congestion), an agent may spend a significant amount of time idling while waiting for outstanding messages to arrive. The idle time could potentially be used to perform extra computation.

- Suppose that all the agents discover very similar changes at the start of the computation phase. Each agent cannot find this out until after the next communication phase. This could mean that many of the changes broadcast in the next communication phase will not be useful (because of the duplication of effort).

There exists the potential to eliminate these effects by basing a new parallel RL method on *asynchronous message passing*. This new method will no longer have distinct computation and communication phases. Instead, the periodic broadcasts are performed by the agents *at different times* (no longer simultaneous broadcasts). In addition, any messages received by an agent are *immediately processed* to incorporate changes into the VFA. When no messages are available for processing,

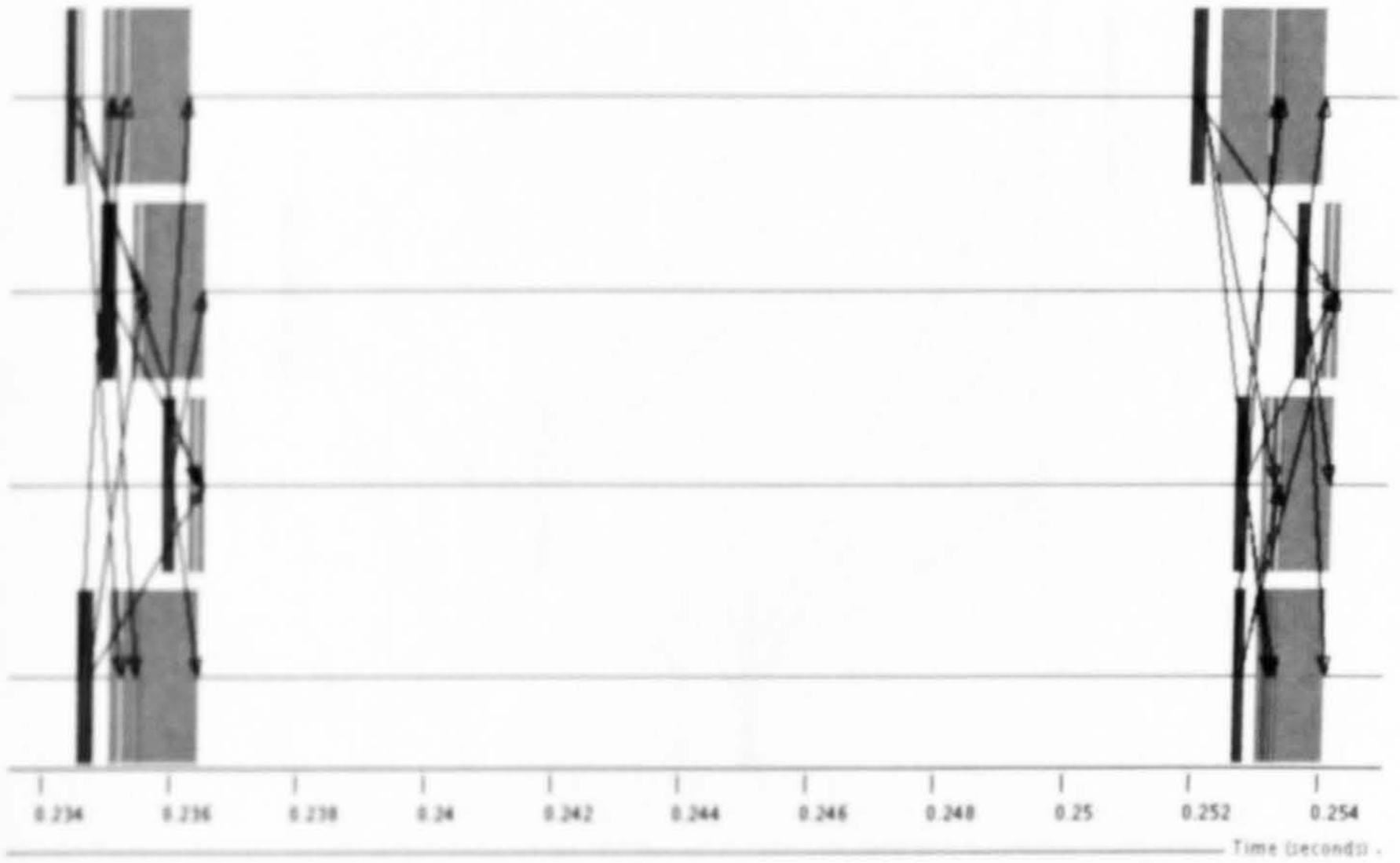


Figure 6.1: Messages exchanged between agents using the (synchronous) selective merging method in the Acrobot task.

an agent will continue to perform learning episodes in the simulation until a new message arrives.

The differences between the (synchronous) selective merging method and the asynchronous method considered in this chapter are illustrated by Figures 6.1 and 6.2. The two figures contain timelines for the two methods, which allow us to see when messages were sent and received by 4 agents during a short interval of an experiment using the Acrobot task. The charts were generated from data logged by MPICH during a number of experiments, using the JUMPSHOT program (version 4) included with MPICH. The two charts correspond to experiments where the merge period $p = 1000$ and the number of changes per message $f_{com} = 128$. In Figure 6.1, the light grey areas indicate times at which an agent was idle while waiting to receive a message from one of the other agents. These areas are not evident in Figure 6.2, since the agents process incoming messages as and when they arrive. Figure 6.2 also illustrates how the asynchronous method distributes the network traffic more evenly over time.

An asynchronous approach of this kind would eliminate the idling of agents waiting for messages, since any delay in message transmission can be used as extra learning time. Additionally, eliminating the requirement for agents to broadcast their messages simultaneously will reduce network congestion by distributing messages more evenly over time.

A less obvious consequence of the asynchronous approach is that very similar

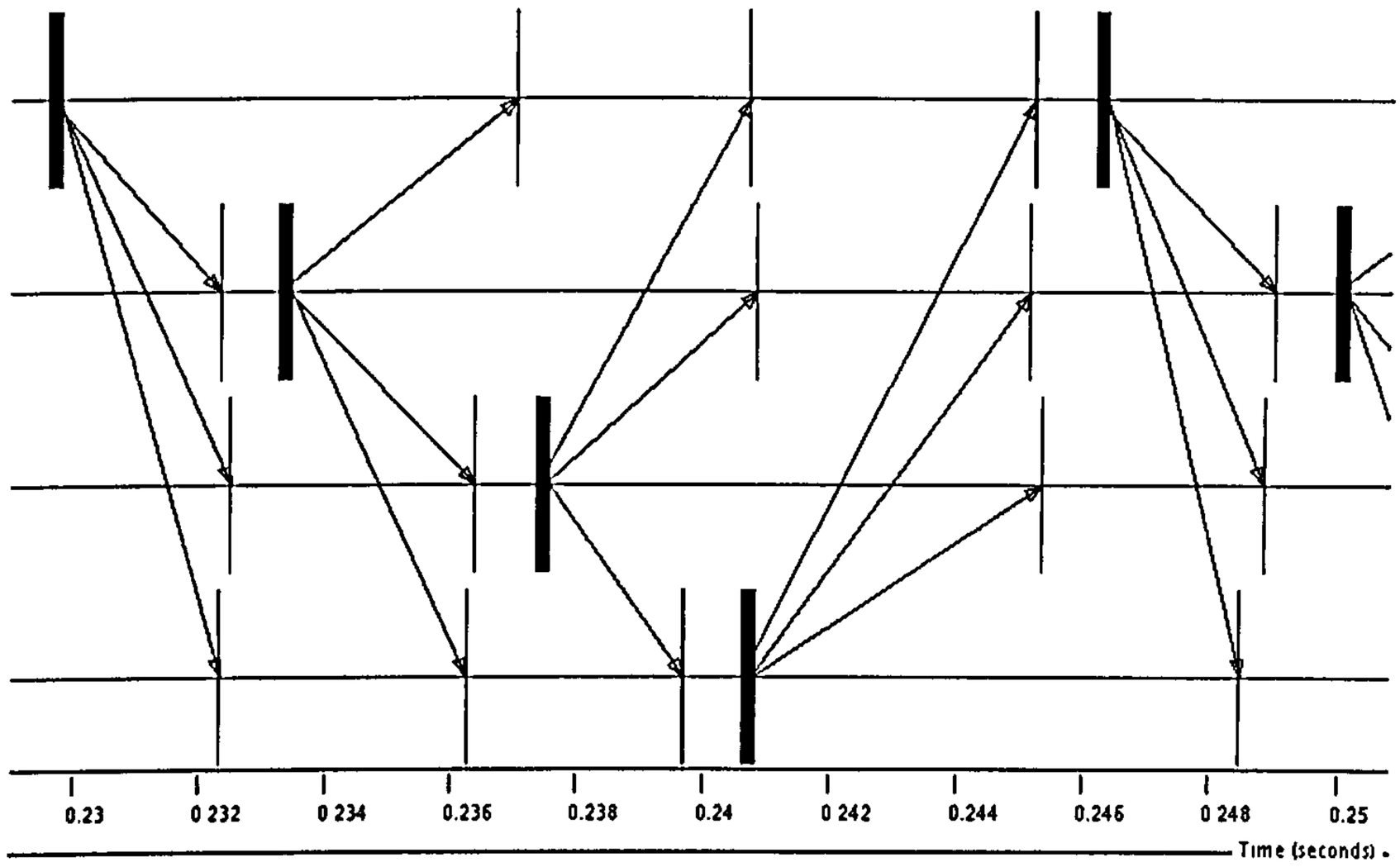


Figure 6.2: Messages exchanged in the Acrobot task by the *asynchronous* method to be defined in Section 6.2.

changes discovered independently by many in the group can be *identified more quickly* and will be *transmitted less frequently.* Suppose there are a total of n agents in the group, and all of these agents discover a large change $\Delta\theta_i$ almost immediately as a parallel run begins. With the selective merge method of Chapter 5, all the agents must complete p simulation steps before this change can be communicated, and since $\Delta\theta_i$ is large it is likely that *all* of the n agents will transmit this change in their messages.

Suppose that each agent using the asynchronous approach broadcasts *on average* every p simulation steps. This means that the overall communication overhead of the asynchronous approach will be similar to that of the selective merge method. Suppose also that these broadcasts are distributed fairly evenly over time, so that on average *one* of the agents will broadcast every p/n simulation steps (mechanisms to achieve this distribution are considered in Section 6.2.3). Under these assumptions, one of the agents will broadcast the change $\Delta\theta_i$ after only p/n simulation steps, which means that the other agents can assume that the change is known by the group. This in turn means that the remaining $(n-1)$ agents will not include $\Delta\theta_i$ in their broadcasts, freeing up space to transmit more useful information in the broadcast messages. The significance of this beneficial effect will become greater as the number of agents n is increased.

For these reasons, it is likely that an asynchronous approach to VFA merging will outperform the methods developed so far in Chapters 4 and 5. This is under the assumption that the communication overhead (including the time required to

incorporate the changes in messages into the VFA) is not significantly different from that of the selective merging method. In the next section, I will go on to define an asynchronous merging method which will be used to validate this claim empirically.

6.2 The Asynchronous Merging Method

The asynchronous method defined here shares several key properties with the selective merge method of Chapter 5. Agents still periodically broadcast their recent weight changes to the rest of the group. The particular weight changes that are sent are still decided by *ranking* the weights in terms of the *magnitude* of the recently observed change. The novel aspects of this new method arise because the high level synchronicity of the previous methods is now relaxed. Messages may now be sent and received at arbitrary times[1], and so the primary challenge in defining this asynchronous method is how to update the VFA weights in response to send and receive events. The core procedure for the asynchronous method is given in Section 6.2.1. In Section 6.2.2 a number of ways to incorporate changes from incoming messages are proposed, resulting in several variants of the core procedure. Finally, in Section 6.2.3, I will address the question of *when* each agent should schedule its message broadcasts.

6.2.1 The Basic Procedure

As was the case for the selective merge method of Chapter 5, the weight change vector $\Delta\vec{\theta}$ is not explicitly stored. Instead we store the vector $\vec{\theta}^{ref}$, which allows $\Delta\vec{\theta} = \vec{\theta} - \vec{\theta}^{ref}$ to be easily calculated whenever necessary. The advantage of this is that the agents' SARSA(λ) learning algorithm can continue to operate solely in terms of $\vec{\theta}$. The individual weight change $\Delta\theta_i$ in the context of the asynchronous merging method signifies *the accumulated local change yet to be communicated to the group.*

The method parameters p and f_{com} are retained from the selective merge method. Parameter f_{com} has an identical purpose in the asynchronous method. When each agent is required to broadcast a message to the group, the message will contain the f_{com} weights of highest rank. The rank of a weight θ_i is determined by the absolute weight change $|\Delta\theta_i|$.

Parameter p has a slightly different purpose. In the selective merge method, each agent would execute exactly p simulation steps between two successive (syn-

[1]Although, over time, the average rate of message transmission remains constant and can be specified in advance for a given experiment.

chronous) merge operations. In the asynchronous method, the agents now all broadcast at different times, but the *average period* between successive broadcasts for a single agent is controlled by parameter p. There are a number of different ways that broadcasts could be scheduled to achieve this average period. Several mechanisms for achieving this will be considered in Section 6.2.3. For now, we will assume that each agent can calculate (without synchronizing with its peers) when it is next due to broadcast a message to the group.

In the selective merge method, after an agent broadcast a message it did not immediately make changes to $\vec{\theta}$ or $\vec{\theta}^{ref}$. It was required to wait for a message from every other agent before any update to the local data structures was permitted. With the asynchronous method, learning must continue immediately after the message has been broadcast. This means that $\vec{\theta}$ and $\vec{\theta}^{ref}$ must be immediately updated to achieve a consistent state. In the absence of any information from the rest of the group, the best option is to assign the value of θ_i to θ_i^{ref} for each weight of index i that was included in the broadcast message. This effectively sets $\Delta\theta_i$ to zero for the subset of weights in the message, under the assumption that the other agents receiving the message will update their own data structures to reflect the change to θ_i.

A side-effect of these immediate updates to $\vec{\theta}^{ref}$ is that if two agents broadcast their messages in quick succession, it is possible that the local value of θ_i^{ref} when a message is sent may be *different* from the value of θ_i^{ref} at a remote agent when the message arrives. This means that if the remote value of θ_i^{ref} and the value of $\Delta\theta_i$ in the received message are used to calculate a new value for θ_i, this value may turn out to be much larger or smaller than the broadcasting agent intended. It is possible that this will result in the remote agent *overshooting* the true expected value of feature i. In order to detect and eliminate these effects, it is insufficient to send tuples of the form $(i, \Delta\theta_i)$ in the message. Instead, it is necessary to send 3-tuples of the form $(i, \Delta\theta_i, \theta_i)$.

Updating an agent's local values of θ_i and θ_i^{ref} in response to an incoming 3-tuple $(i, \Delta\theta_i', \theta_i')$ presents a number of challenges. If the change in the tuple represents new information for the agent, we want to incorporate this into the value function in order to accelerate convergence. However, if the change has already been discovered by the agent, we do not want to add in the same change again, since this could lead to overshooting and interfere with convergence. A number of mechanisms to achieve these goals are considered in Section 6.2.2, which leads to several variations of the core asynchronous method depending on which of the mechanisms is used. At this point, I will assume that a function *update* exists which takes 4 arguments (the local values of θ_i and θ_i^{ref}, and the message data $\Delta\theta_i'$

and θ_i') and returns a pair of values which can be used to update the local values of θ_i and θ_i^{ref}.

Given some choice of the *update* function and a mechanism for scheduling individual agent broadcasts, the procedure followed by an agent using the asynchronous merge method is given in Algorithm 5.

A set of q simulation steps is performed at the start of each iteration of the main loop. These q steps will be referred to as a *learning quantum.* In the remainder of the main loop, checks are made to see if a broadcast is due or if any incoming messages have arrived. If a broadcast is due, a message is constructed and sent to all the other agents. If any messages have arrived, they are processed and incorporated into local data structures using the *update* function.

The reason why a quantum $q > 1$ is necessary is because it is potentially expensive to check whether any messages have arrived. In the implementation on the cluster of workstations, asynchronous message passing was implemented using the MPICH functions `MPI_Isend`, `MPI_Iprobe` and `MPI_Recv`. The `MPI_Isend` function is a non-blocking function used here to send a message to all agents other than the sender, copying the message data in each case from a single buffer in user memory. The `MPI_Iprobe` function is a non-blocking function which can be used to check for the arrival of messages from other agents. Once a message has been detected with `MPI_Iprobe`, the message data can be retrieved using the `MPI_Recv` function. Our initial implementation used a quantum $q = 1$, which meant that a check was made for incoming messages after every simulation step. We discovered that with this initial implementation, each agent would spend a significant amount of its total running time executing the `MPI_Iprobe` function. Since it turned out to be so expensive, it was necessary to choose a larger quantum so that fewer calls would be made to `MPI_Iprobe` over the lifetime of the agent. However, q should not be too large, since this would mean that messages may arrive at the agent and not be processed for some time. In all the experiments reported in this chapter, it was found that a value of $q = 25$ allowed messages to be detected and processed quickly without there being an excessive number of calls to `MPI_Iprobe`.

In sections 6.2.2 and 6.2.3, I will go on to describe the elements of the asynchronous method which have been left undefined: how the local data structures are updated in response to incoming messages, and how each agent can determine when it is due to broadcast a message.

6.2.2 Updating after Message Received

To progress towards a complete definition of the asynchronous merge method, it is now necessary to define how an agent updates its data structures when a message

Algorithm 5 Agent pseudocode for the asynchronous merge method.

{Initialization}
for all i **do**
  $\theta_i \leftarrow \theta_{init}$
  $\theta_i^{ref} \leftarrow \theta_{init}$
end for

{Main loop}
while time elapsed $< t_{end}$ **do**

  {Learning quantum}
  for $step = 1$ to q **do**
    Execute a simulation step and update weight vector $\vec{\theta}$.
  end for

  {Scheduled Broadcasts}
  if scheduled broadcast is due **then**
    Calculate $\Delta\vec{\theta} = \vec{\theta} - \vec{\theta}^{ref}$.
    Rank each index i according to the value of $|\Delta\theta_i|$.
    $best \leftarrow$ { the f_{com} highest ranked indices }
    $m \leftarrow \{(i, \Delta\theta_i, \theta_i) \mid i \in best\}$
    Send message m to all other agents.
  end if

  {Message Receive}
  for each new incoming message m **do**
    for all $(i, \Delta\theta_i', \theta_i') \in m$ **do**
      $\vec{r} \leftarrow update(\theta_i, \theta_i^{ref}, \Delta\theta_i', \theta_i')$
      {Assign elements of result $\vec{r}$ to θ_i and θ_i^{ref}}
      $\theta_i \leftarrow r_1$
      $\theta_i^{ref} \leftarrow r_2$
    end for
  end for

end while

arrives from another agent. In other words, when a 3-tuple $(i, \Delta\theta'_i, \theta'_i)$ is received in an incoming message, we must define how local variables θ_i and θ_i^{ref} should be updated. This functionality is encapsulated in the *update* function, which will be defined in this section.

The simplest *update* function we can use is to simply add in the remote difference $\Delta\theta'_i$ while ignoring the resulting remote value θ'_i, as shown in Algorithm 6. The local change $\Delta\theta_i$ remains the same after the update. This means that the changes $\Delta\theta_i$ and $\Delta\theta'_i$ are combined by addition in the value function, but the agent still remembers the local change $\Delta\theta_i$ for later communication to the group.

Algorithm 6 An *update* function which simply adds in the remote change.

function UPDATE$(\theta_i, \theta_i^{ref}, \Delta\theta'_i, \theta'_i)$

return $(\theta_i + \Delta\theta'_i \ , \ \theta_i^{ref} + \Delta\theta'_i)$

end function

The simple *update* function given in Algorithm 6 works well if only one of the two agents involved (the sending and receiving agents) has discovered a significant change to θ_i in the recent past. The trouble arises when both of these agents have recently discovered a similar change to θ_i, as illustrated in Figure 6.3.

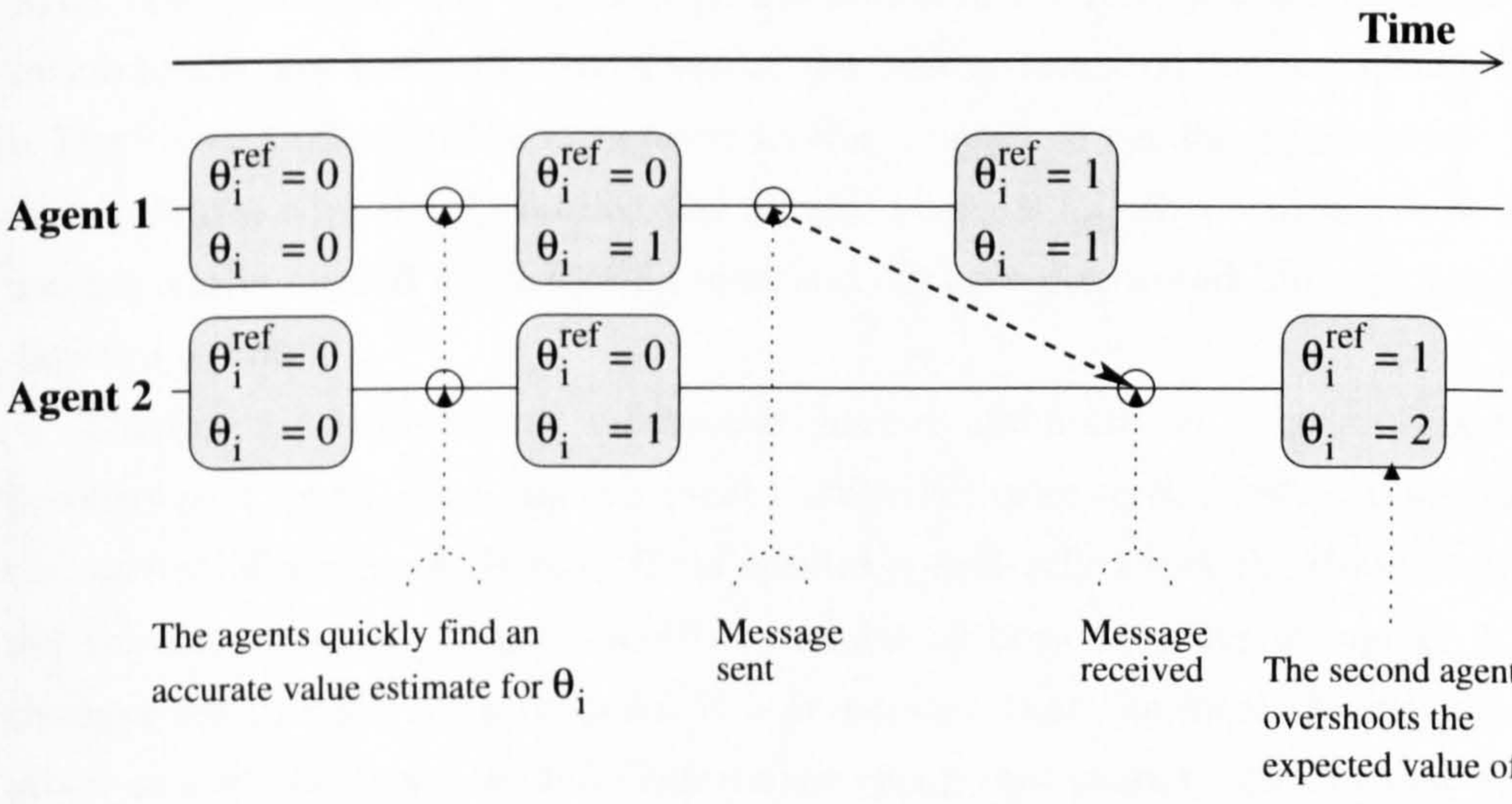


Figure 6.3: Example of how overshooting can occur when two agents simultaneously discover a change to weight. The simple *update* function given in Algorithm 6 is used.

Figure 6.3 focuses on the evolution of the weights corresponding to a single specific feature ϕ_i (i.e. i has some fixed value.) At the start of the timeline in the

Algorithm 7 The *cancel* function is used to cancel out part or all of a local change in response to a remote change.

function CANCEL($\theta_i, \theta_i^{ref}, \Delta\theta_i'$)
  $\Delta\theta_i \leftarrow \theta_i - \theta_i^{ref}$
  $b \leftarrow \theta_i^{ref} + \Delta\theta_i'$

  if $sign(\Delta\theta_i) \neq sign(\Delta\theta_i')$ **then** {keep the local change intact}
    $a \leftarrow \theta_i + \Delta\theta_i'$
  else if $|\Delta\theta_i| > |\Delta\theta_i'|$ **then** {cancel part of the local change}
    $a \leftarrow \theta_i$
  else {cancel all of the local change}
    $a \leftarrow b$
  end if

  return (a, b) {a and b contain new values for θ_i and θ_i^{ref}}
end function

figure, both agents still have θ_i and θ_{ref} set to their initial values of zero. Soon after the timeline begins, the two agents concurrently learn an accurate estimate of weight $\theta_i = 1$. The first agent then sends a message to the second agent. After the agent receiving the message has added in the remote change, its current estimate changes to $\theta_i = 2$, *overshooting* the best estimate of the two agents. This is likely to interfere with convergence as the number of agents is increased. This demonstrates why simply adding the agents' changes together will not suffice. A mechanism is needed for detecting identical changes discovered independently by different agents.

The first mechanism that will be used here to eliminate the overshooting effect involves part or all of an agent's local change being *cancelled out* in response to the arrival of a remote change. This applies specifically when the changes are in the same direction (i.e. both positive changes or both negative changes.) If the changes are in different directions, it is important that the local change remains intact so that the agent can later inform the group that there is some evidence that the expected value of θ_i lies in another direction. If the changes are in the same direction, however, broadcasting the local change later would result in the group overshooting the expectation, so in this case it is important to reduce the size of the local change so that the agent's value for θ_i remains consistent. The *cancel* function (given in Algorithm 7) implements the mechanism described above.

The *cancel* function returns a pair of values containing new values for θ_i and

θ_i^{ref}. Note that in all cases, the *cancel* function returns a value of $\theta_i^{ref} + \Delta\theta_i'$ (stored in temporary variable b) as the new value for θ_i^{ref}, i.e. the remote change is *always* added to the reference weight. The new value (stored in temporary variable a) to be assigned to local weight θ_i determines if part of the local change is cancelled. In the case where the local and remote changes are in opposite directions, a is assigned the value of $\theta_i + \Delta\theta_i'$, which means the local change is left *unmodified*. In the case where the remote change is in the same direction as the local change, but has a *smaller magnitude*, a is assigned the old value of θ_i. Since the remote change is added to θ_i^{ref}, what is left of the local change is the *difference* between the two. If the remote change has the same direction but *greater magnitude*, the new value of θ_i^{ref} will be greater than the old value of θ_i, so the local change must be completely cancelled by assigning the value of b to a.

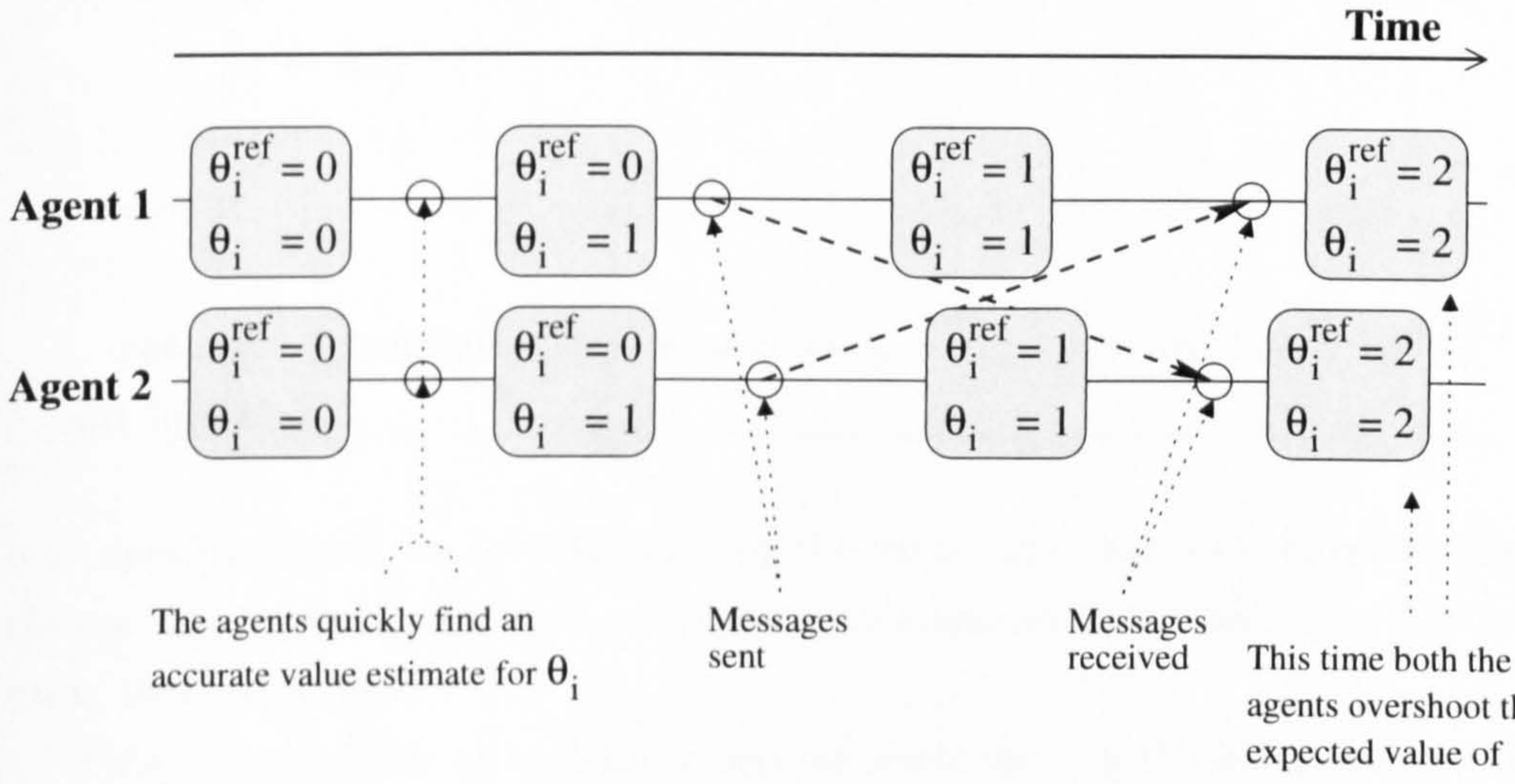


Figure 6.4: A second example of overshooting where two agents broadcast an identical change in quick succession. The simple *update* function given in Algorithm 6 is used.

An *update* function based on the *cancel* function will eliminate the overshooting effect illustrated in Figure 6.3. However, there are other situations where overshooting can occur which can not be corrected by eliminating part of the local change $\Delta\theta_i$. Such situations arise because of the asynchronous nature of the merge method, and the fact that it is possible for one agent to broadcast a message while an incoming message from another agent is still in transit. If both of these agents have discovered very similar changes to a single weight, it is likely that both the agents will overshoot the appropriate value for that weight. This process is illustrated in Figure 6.4. At the time the messages arrive in this Figure, neither of the agents has learned any local change since their last broadcasts. Since the

Algorithm 8 The *filter* function is used to exclude part or all of a remote weight change when it is detected that the change is inconsistent with the current value of the local weight.

```
function FILTER(θ_i, Δθ'_i, θ'_i)
    c ← 0
    Δθ†_i ← θ'_i − θ_i {the difference between local weight and new remote weight}
    {allow remote change only if it is in the same direction as Δθ†_i}
    if sign(Δθ†_i) = sign(Δθ'_i) then
        {filtered change minimizes magnitude of Δθ†_i and Δθ'_i}
        if |Δθ†_i| < |Δθ'_i| then
            c ← Δθ†_i
        else
            c ← Δθ'_i
        end if
    end if

    return c {c contains the filtered change, used later to modify θ_i and θ^ref_i}
end function
```

only operation that can be performed by the *cancel* function is to reduce the local change $\Delta\theta_i$ towards zero, it is not possible to eliminate overshooting of this type using this mechanism.

This example leads us to define a second mechanism to eliminate overshooting effects. The intuitive purpose of this second mechanism is to *filter out* incoming weight changes which are inconsistent given the current values of the relevant weights. To detect these inconsistencies, it is necessary to compare the value of the remote weight when the message was sent (θ'_i) to the local weight value (θ_i) of the agent receiving the message. This is why the messages sent by agents in the asynchronous merge method consist of a set of 3-tuples $(i, \Delta\theta'_i, \theta'_i)$. The additional value of θ'_i is necessary to detect inconsistent weight changes.

The *filter* function (given in Algorithm 8) implements the second mechanism. This function returns a single value, the *filtered change*. This change is set to zero if the incoming remote change is discovered to be inconsistent.

At the start of the *filter* function, the value of $\Delta\theta^{\dagger}_i = \theta'_i - \theta_i$ is calculated. The value of $\Delta\theta^{\dagger}_i$ represents the change that would be needed to move from the current value of weight θ_i to the remote agent's weight value at the time the message was sent. If the signs of $\Delta\theta^{\dagger}_i$ and the remote change $\Delta\theta'_i$ are *different*, this indicates

that the local agent has already moved θ_i in the direction of $\Delta\theta'_i$ *beyond* the value of θ'_i achieved by the remote agent. In this case, it is reasonable to ignore the incoming change by setting the filtered change c to zero.

If the signs of $\Delta\theta^{\dagger}_i$ and $\Delta\theta'_i$ are the same the incoming change will not be ignored. However, it may still be necessary to reduce the magnitude of the incoming change. This is achieved by returning whichever of the two values $\Delta\theta^{\dagger}_i$ and $\Delta\theta'_i$ has the smaller magnitude. If the current weight value θ_i lies within the range $(\theta'_i, \theta'_i + \Delta\theta'_i)$ this will cause the filtered change c to have a smaller magnitude than the incoming change $\Delta\theta'_i$. If the current weight value lies outside this range, the filtered change c will be identical to $\Delta\theta'_i$.

Using the *filter* function as the mechanism for eliminating overshooting will allow consistent values for θ_i to be maintained in *both* of the examples given in Figures 6.3 and 6.4. However, it also worth noting that the *filter* mechanism will produce *more* tuples to be broadcast in some situations. For example, consider the single message being sent in Figure 6.3. If the *cancel* mechanism is used, Agent 2 will have local change $\Delta\theta_i = 0$ after the message is received, correctly reflecting the fact that there is no further change worth communicating to the group. If the *filter* mechanism is used, however, the incoming change $\Delta\theta'_i = 1$ will be simply filtered out, leaving θ_i and θ^{ref}_i at their existing values, and therefore leaving the local change of $\Delta\theta_i = 1$ intact. This means that when the scheduled broadcast of Agent 2 occurs, a 3-tuple for weight θ_i is likely to be included, despite the fact that Agent 1 is already well aware of this change. Sending this extra tuple does not affect consistency, since Agent 1 will simply filter it out as inconsistent. However, the extra tuple does take up space in the message which could be taken up by more informative weight changes, so this will have an impact on the overall performance.

Having motivated and defined the *filter* and *cancel* functions, it is now possible to define the update functions which will form the basis of three variants of the asynchronous merge method. Update function 1 is defined in Algorithm 9, and is based on the *cancel* function only. Update function 2 is defined in Algorithm 10, and is based on the *filter* function only. Update function 3 is defined in Algorithm 11, and uses a *combination* of the *filter* and *cancel* functions.

The first two update functions delegate most of their work to the *cancel* and *filter* functions respectively, both of which were described in detail above. Update function 3 requires some additional explanation. The motivation for combining the two mechanisms is to create a method which has the robustness of *filter* when messages from different agents are transmitted almost simultaneously, but which also eliminates some of the redundant tuples sent by *filter* by using the *cancel* mechanism to eliminate local changes that are already known by the group.

Update function 3 begins by calling the *filter* function, and storing the result (the filtered change) in temporary variable c. If the signs of the filtered change and the local change $\Delta\theta_i$ are the same, this indicates that the remote change has overtaken the local change, which should be completely cancelled out. In this case the *cancel* function is called, passing the value of $c + \Delta\theta_i$ for the remote change[2]. If the signs of the filtered change and the local change are different, the filtered change is simply added to both θ_i and θ_i^{ref}, since changes in opposite directions do not cancel each other out.

Algorithm 9 Update function 1. Uses only the *cancel* function.

```
function UPDATE1(θ_i, θ_i^ref, Δθ'_i, θ'_i)
    return CANCEL(θ_i, θ_i^ref, Δθ'_i)
end function
```

Algorithm 10 Update function 2. Uses only the *filter* function.

```
function UPDATE2(θ_i, θ_i^ref, Δθ'_i, θ'_i)
    c ← FILTER(θ_i, Δθ'_i, θ'_i)
    return (θ_i + c , θ_i^ref + c)
end function
```

Algorithm 11 Update function 3. Uses both the *filter* and *cancel* functions.

```
function UPDATE3(θ_i, θ_i^ref, Δθ'_i, θ'_i)
    Δθ_i ← θ_i − θ_i^ref
    c ← FILTER(θ_i, Δθ'_i, θ'_i)
    if sign(c) = sign(Δθ_i) then
        return CANCEL(θ_i, θ_i^ref, c + Δθ_i)
    else
        return (θ_i + c , θ_i^ref + c)
    end if
end function
```

A comparison of these three update functions will allow the relative importance of the *filter* and *cancel* mechanisms to be examined, as well as an assessment of how well the two mechanisms are combined in update function 3. Three variations of the asynchronous merge method, each based on one of these update functions, will be fully evaluated in Section 6.3.

[2]The value of $c+\Delta\theta_i$ arises because the filtered change c measures the change from the current weight value θ_i. The *cancel* function, on the other hand, expects any change to be measured from the reference weight θ_i^{ref}, so the local weight change $\Delta\theta_i$ must be added to c before passing this value to the *cancel* function, ensuring the results are consistent.

Before the evaluation can take place, however, the definition of the asynchronous merge method must be completed by specifying how the agents determine when to broadcast messages to the other agents.

6.2.3 Scheduling the Message Broadcasts

The definition of the asynchronous merge method in Algorithm 5 (see Section 6.2.1) indicated that agents would periodically construct a message (of 3-tuples) and send it to every other agent in the group. It was indicated that these broadcasts would occur each time "a scheduled broadcast is due." In this section, mechanisms for scheduling the agents' individual broadcasts in a decentralized manner will be presented.

The following basic properties were used as tenets for selecting a scheduling mechanism:

1. The *mean* period between two consecutive broadcasts of an individual agent in the group is p simulation time steps.

2. The agents do not need to exchange messages to synchronize their scheduling mechanisms.

3. The broadcasts of the group should be well distributed over time, not clustered together in short intervals.

The reasoning behind these properties is as follows. Requiring that the average period between an agent's broadcasts is p time steps means that the overall network bandwidth consumed by the method can be controlled by parameters p and f_{com} (the number of 3-tuples per message). This is important for tuning the performance of the method on different parallel systems. It also will allow the asynchronous method to be compared with the selective method of Chapter 5.

Specifying that the agents do not synchronize their scheduling mechanisms conforms to the asynchronous character of the algorithm, and allows all the available bandwidth to be dedicated to the exchange of weight changes. This specification also *simplifies* the range of mechanisms we can consider. It would not be difficult, for example, to mark each message with a timestamp and then use the series of timestamps to detect when the agents are drifting out of sync with each other. Lightweight time synchronization methods of this kind were not considered as part of this thesis.

Distributing the agents' broadcasts widely over time is necessary to exploit the full potential of the asynchronous merge method. In Section 6.1 the motivation behind the asynchronous merge method was given. Two of the most advantageous

properties of this method only arise if the broadcasts are well distributed. The first of these is minimizing network congestion. The second is quickly identifying and eliminating very similar changes discovered independently by different agents.

Adhering to these basic properties does exclude some interesting alternative scheduling mechanisms. For example, having both p and f_{com} remain fixed implies that each agent will consume the same bandwidth over its entire lifetime. However, as learning progresses it becomes increasingly unlikely that an agent will discover any new information about the environment. It might therefore be advantageous to decrease communication in the later stages and devote more time to computation. One way to achieve this would be to link the probability of a broadcast to the *total size* of the weight changes yet to be communicated. However, such an approach makes the overall bandwidth required more difficult to predict, making it harder to tune the algorithm to a particular domain and parallel computer. For this reason, variable-bandwidth scheduling is not considered in this thesis.

Throughout the research for this thesis, we experimented with three mechanisms to determine when each agent should broadcast a message.

Uniform Schedule

The *uniform schedule* uses the simplest mechanism, which corresponds closely to the way broadcasts occur in the (synchronous) selective method of Chapter 5. Counting the total number of time steps t experienced by a single agent from the start of a parallel run, each agent broadcasts a message at $t = p$, $t = 2p$, $t = 3p$... etc. In other words, for every agent a broadcast occurs at $t = kp$ for all $k \in \mathbb{Z}$, $k > 0$. No communication is required between the agents. Each agent simply monitors how many local simulation steps have been observed, and broadcasts a message when the appropriate interval has elapsed. The mean period between broadcasts is clearly p simulation steps in this case.

If all the agents took exactly the same time to run a simulation step and send/receive messages, the broadcasts would occur at exactly the same time for all the agents. In practice this is not the case. Complex simulations may require quite different amounts of computation on different time steps, and since the agents explore randomly to some degree it is likely they will differ in this regard. The agents will also observe different patterns of processor cache misses, page faults and other operating system interrupt events. Finally, network congestion and the underlying TCP transport mechanism may introduce significant variance in the time for a message to travel over the network.

The end result of these variations is shown in Figure 6.5. This Figure depicts timelines for 16 agents using the asynchronous merge method with a uniform

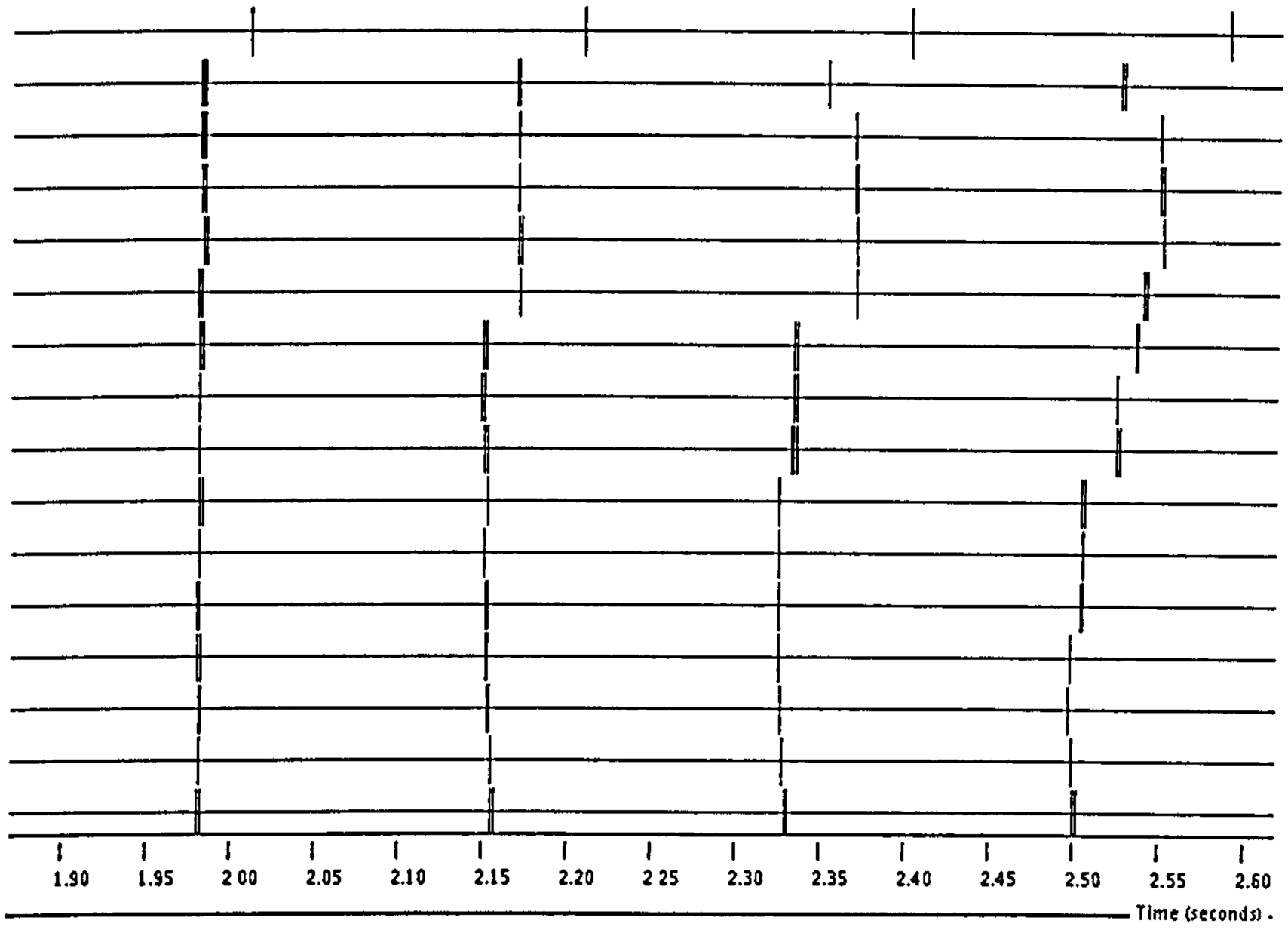


Figure 6.5: *Uniform schedule.* Message send events for 16 agents in the early stages of the Stochastic Grid World task (high difficulty).

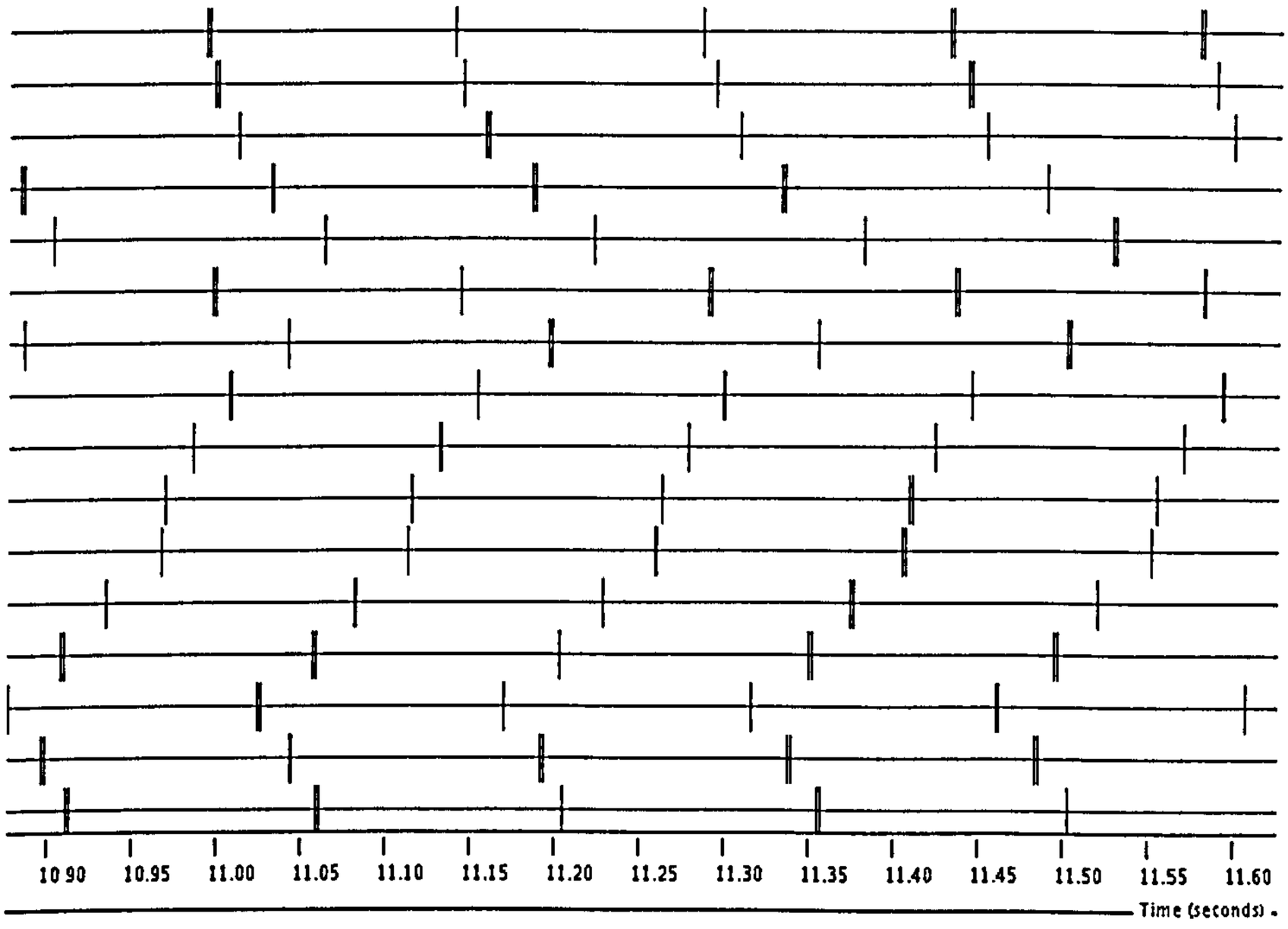


Figure 6.6: *Uniform schedule.* Message send events for 16 agents in the later stages of the Stochastic Grid World task (high difficulty).

schedule, and was generated (using the JUMPSHOT program) from MPI logging information. The times at which a broadcast message was sent are shown on the timelines. The first four broadcasts in a single parallel run are shown. While the agents' first broadcasts all occur simultaneously, variations in processing time cause the individual broadcasts to begin to spread out. Figure 6.6 shows the timeline much later in the same experiment. By this point, the initial synchronization of the agents' broadcasts has been almost entirely lost, and overall the broadcasts exhibit a more unpredictable distribution.

It is clear that the simultaneous broadcasts in the early stages of the parallel run are less than ideal. To what extent this affects the overall performance of the method is not obvious, and so it was decided that it would be valuable to compare the performance of this simple mechanism with the two mechanisms defined below.

Staggered schedule

The *staggered schedule* is closely related to the uniform schedule. In both cases, each agent completes a fixed period of p simulation steps between consecutive broadcasts. However, in the staggered schedule the very first broadcast takes place after a different number of steps for each agent. Each of the n agents has a rank which identifies it uniquely within the group. The ranks are integers which run from 0 to $(n-1)$. Counting the total number of time steps t experienced by a single agent from the start of a parallel run, the agent with rank r broadcasts a message at $t = \lfloor p(k + \frac{r+1}{n}) \rfloor$ for all $k \in \mathbb{Z}$, $k \geq 0$.

Figure 6.7 shows the first few broadcast events for 16 agents using the staggered schedule. The first set of broadcasts occur in sequence, distributed quite uniformly over the time interval. As in the case of the uniform schedule, this initial uniformity is quickly affected by variance in the processing time, and the broadcasts soon tend towards a more random pattern. After some time the distribution of the agents' broadcasts reach a similar pattern to that reached using the uniform schedule in Figure 6.6.

The staggered schedule avoids the simultaneous broadcasts that occur in the early stages using the uniform schedule, so it is reasonable to expect that the performance of the staggered schedule will be better.

Exponential schedule

In contrast to the previous two schedules, agents following the *exponential schedule* do not execute a fixed number of simulation steps between consecutive communications. Instead, the occurrence of broadcast events is modelled using a *Poisson*

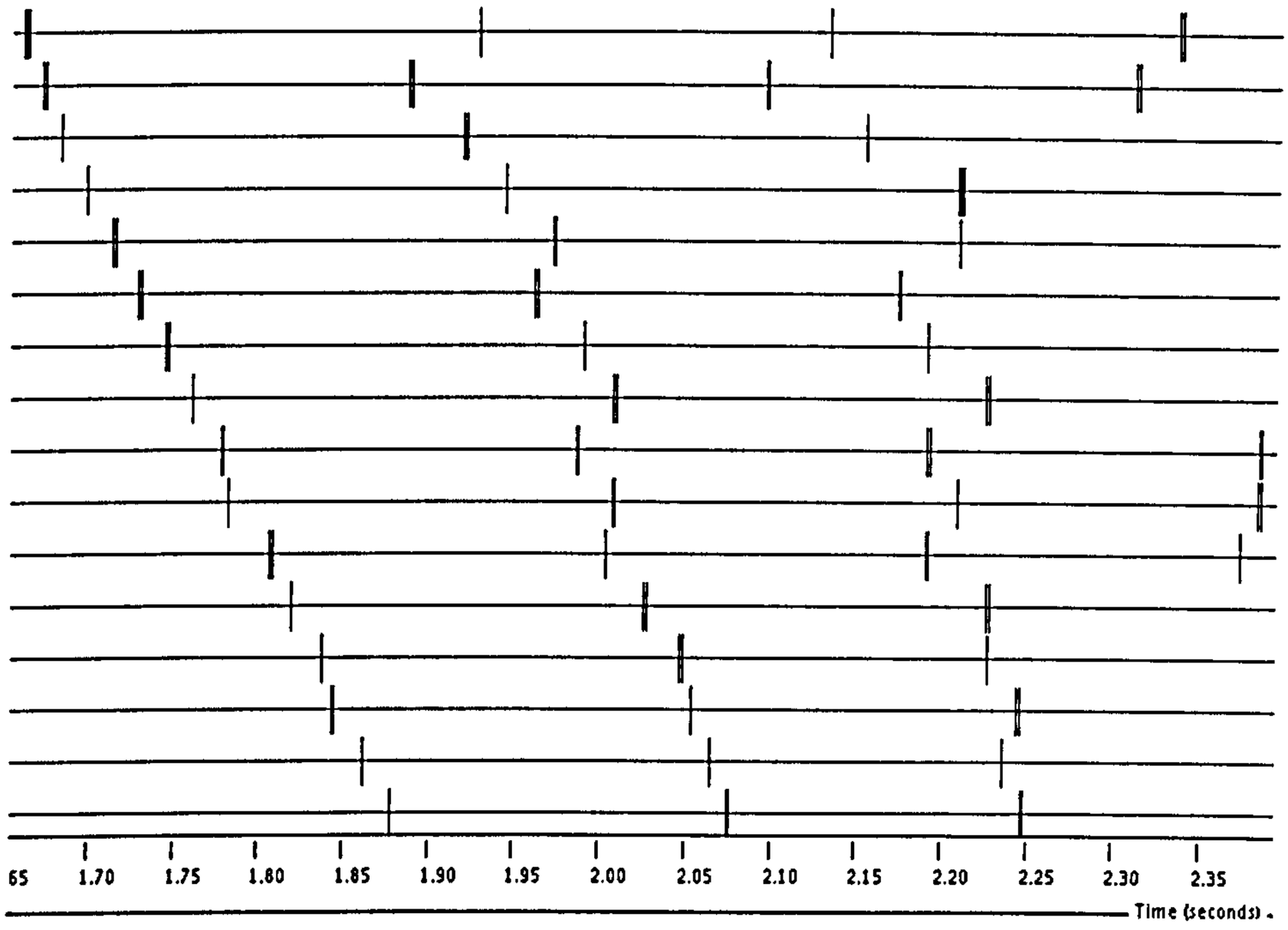


Figure 6.7: *Staggered schedule.* Message send events for 16 agents in the early stages of the Stochastic Grid World task (high difficulty).

process. This means that the time between successive broadcasts can be modelled using an *exponential distribution* of mean p. In our implementation, the number of simulation steps which must be taken before the next broadcast is calculated at the start of a parallel run and after every subsequent broadcast. The number is sampled from a pseudo-random variable which draws numbers using a distribution defined by the following probability density function:

$$f(x) = \begin{cases} \frac{1}{p}e^{-\frac{x}{p}} & \text{, if } x \geq 0, \\ 0 & \text{, if } x < 0. \end{cases}$$

In essence, this results in significant variation in the period between consecutive communications, to the extent which there may be several communications by one agent in the time where another agent makes no communication at all. Unlike the previous two schedules, the pattern of the first few broadcasts does not differ significantly from the pattern achieved later in the experiment. Both exhibit quite a random pattern, such as that shown in Figure 6.8. The other major difference from the other two schedules is that here the variation in the period between broadcasts is dominated by the the variance of the exponential distribution, rather than the small variance introduced by the processor, operating system and interconnection network of the parallel system.

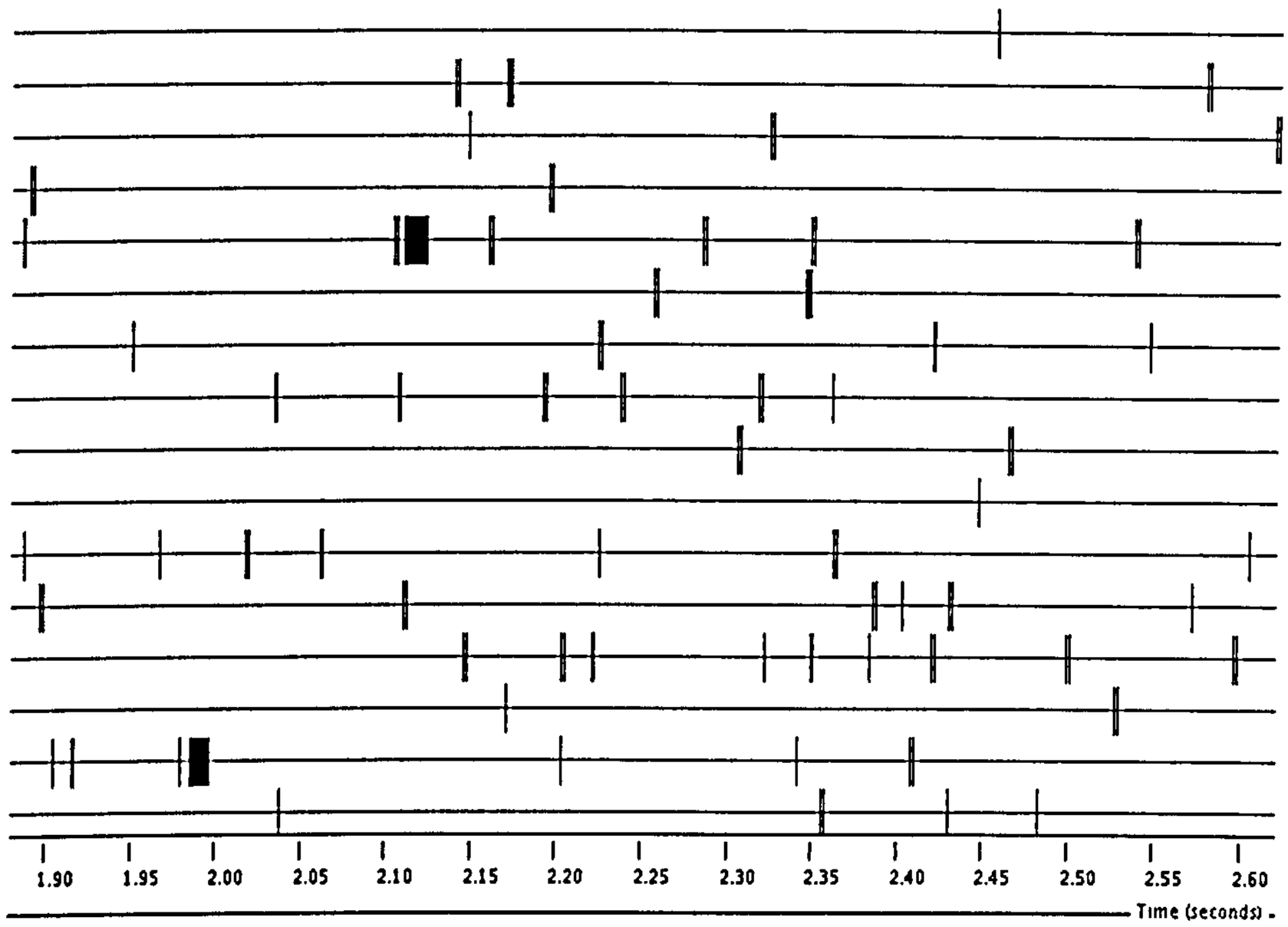


Figure 6.8: *Exponential schedule.* Message send events for 16 agents in the early stages of the Stochastic Grid World task (high difficulty).

Comparing the scheduling mechanisms

A full evaluation of the effect of the scheduling mechanism on performance is not included in this thesis, since the bulk of the experiments carried out for this chapter were focused on establishing a successful method to update the VFA asynchronously (discussed in Section 6.2.2). In a series of preliminary experiments, it was discovered that the staggered schedule consistently produced the best performance out of the three schedules across a variety of domains and update functions.

A graph comparing the performance of the three schedules in one experiment is given in Figure 6.9. In this experiment, 16 agents using update function #1 were evaluated in the high-difficulty Stochastic Grid World task. The merge period was $p = 100,000$, and the number of tuples per message was $f_{com} = 1024$. Reward function #1 was used, and parameters α and ϵ were decayed linearly according to $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The other parameters were $\gamma = 1.0$, $\lambda = 0.95$, $\theta_{init} = 0$. The results were averaged over 10 runs, and episodes were terminated if they reached 10,000 steps.

In Figure 6.9 we can see that the 16 agents using the staggered schedule approach the optimal policy at the fastest rate. The exponential schedule is the next best option, converging slightly less quickly to the optimum. The agents using the uniform schedule initially converge at a similar rate, but ultimately the group only

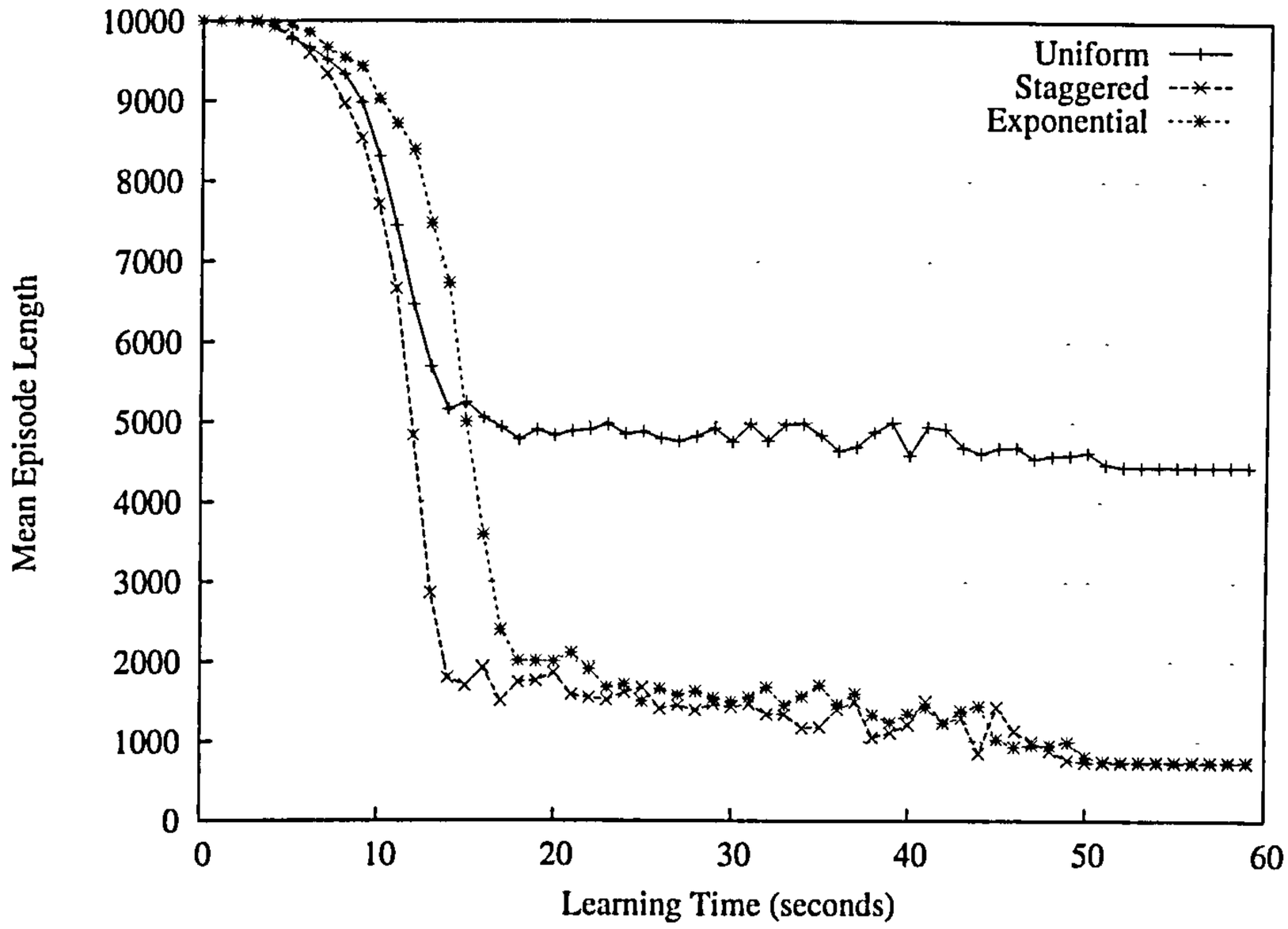


Figure 6.9: Comparison of the performance of three scheduling mechanisms using 16 agents in the Stochastic Grid World task (high difficulty).

converged to a near optimal policy on about half the runs. This makes the uniform schedule seem greatly inferior, but the main reason for this poor performance is the pairing of the uniform schedule with update function #1. Recall from Section 6.2.2 that update function #1 has no mechanism to eliminate overshooting as a result of two agents broadcasting an identical change simultaneously. Using the uniform schedule, the first few broadcasts are *all* simultaneous, and the overshooting has a seriously detrimental effect on convergence.

When there is a mechanism to deal with this kind of overshooting (such as that of update functions #2 and #3) the performance of the uniform schedule is not nearly so bad, although it is still consistently outperformed by the staggered schedule.

The exponential schedule performs worse than the staggered schedule in all the experiments we have carried out. The main reason for this seems to be the relatively high variance produced in the period between an agent's successive broadcasts. The nature of the exponential distribution is such that often a pattern can be observed in the MPI logs where an agent will broadcast several times quickly in succession, then wait for a time up to 2 or 3 times greater than p before the next broadcast takes place (this can be observed in Figure 6.8). As far as the asynchronous merge method is concerned, the resulting performance would be much better if these broadcasts occurred more uniformly. It is likely that a schedule based on a

random distribution with less variance than the exponential distribution (such as a Gamma distribution) could equal (or possibly exceed) the performance achieved with the staggered schedule.

The remainder of the results reported in this chapter are based on the staggered schedule mechanism, since out of the proposed mechanisms this was found to consistently produce the best results.

6.3 Evaluation of Asynchronous Merging

In the previous section, a detailed description of the asynchronous merging method was given. Details were provided about how each of the agents schedules its broadcasts to the other agents. In addition, three candidate mechanisms (known here as *update functions*) for updating the local VFA in response to incoming messages were proposed. In this section, an evaluation of the asynchronous merging method is reported. The main purpose of this evaluation is to compare the performance that can be achieved using each of the three update functions. A secondary outcome of this evaluation is that the performance of the asynchronous merge method in general may be compared with the results previously obtained for the selective method of Chapter 5 and the merging method of Chapter 4.

The relative performance of each of the update functions is shown in each graph in this section, as two key dimensions are varied. The first dimension is the number of agents n used in each experiment. Experiments were performed for 2, 4, 8 and 16 agents. The second dimension is the evaluation domain being used. Experiments were performed for each of the evaluation domains defined in Section 4.3.1. Whenever possible this evaluation will use the same experimental settings that were used in the evaluation of the selective merge method (see Section 5.4) so that the results can be compared directly.

Stochastic Grid World (low-difficulty)

The first evaluation domain considered here is the low-difficulty Stochastic Grid World task. Each individual graph shown in Figures 6.10–6.13 shows the resulting performance of the asynchronous merge method with each of the three update functions. The different graphs correspond to different numbers of agents, as the number of agents is increased from 2 up to 16.

In this series of experiments, reward function #2 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$,

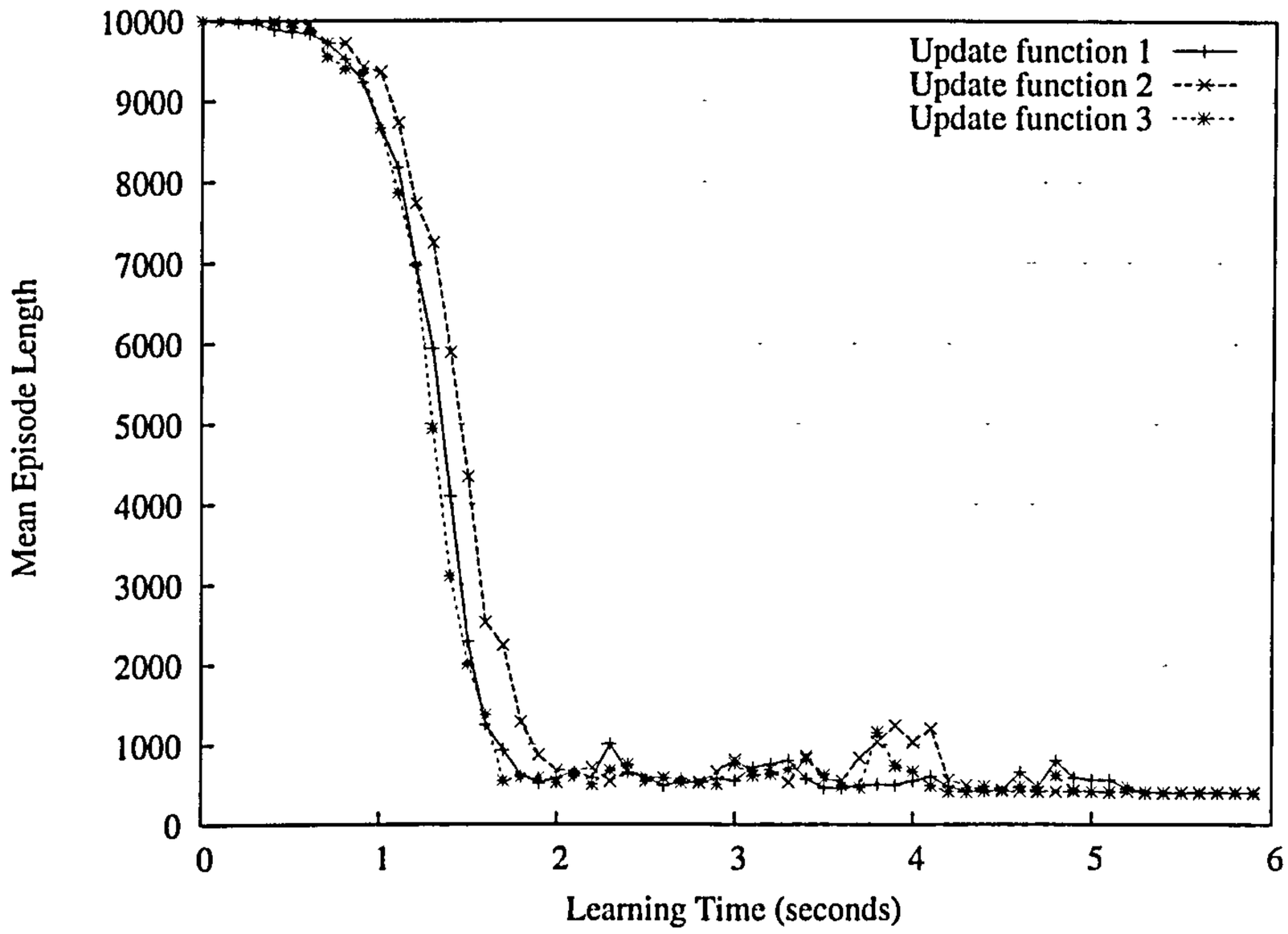


Figure 6.10: Comparing asynchronous update functions with 2 agents in the low-difficulty Stochastic Grid World task.

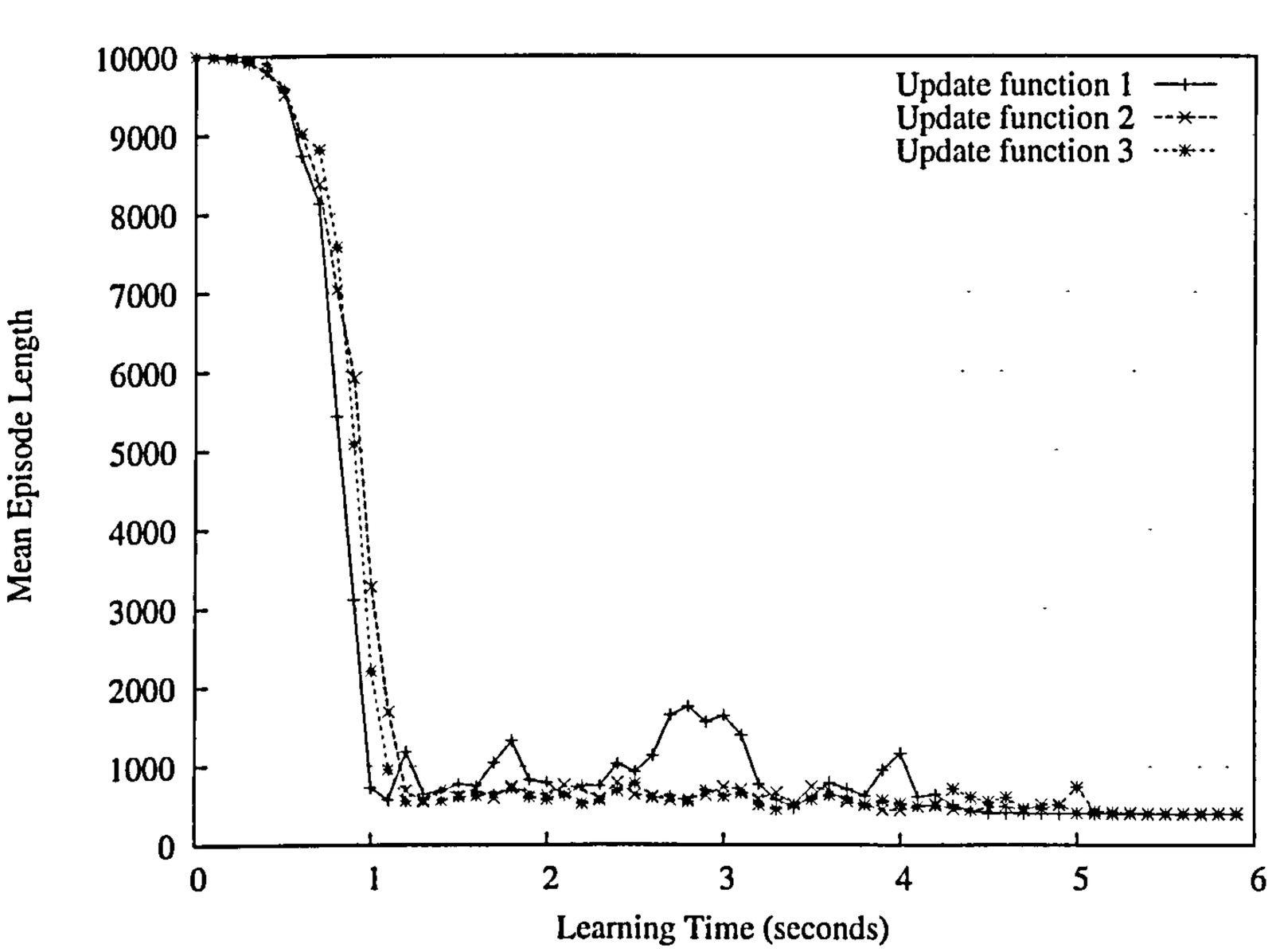


Figure 6.11: Comparing asynchronous update functions with 4 agents in the low-difficulty Stochastic Grid World task.

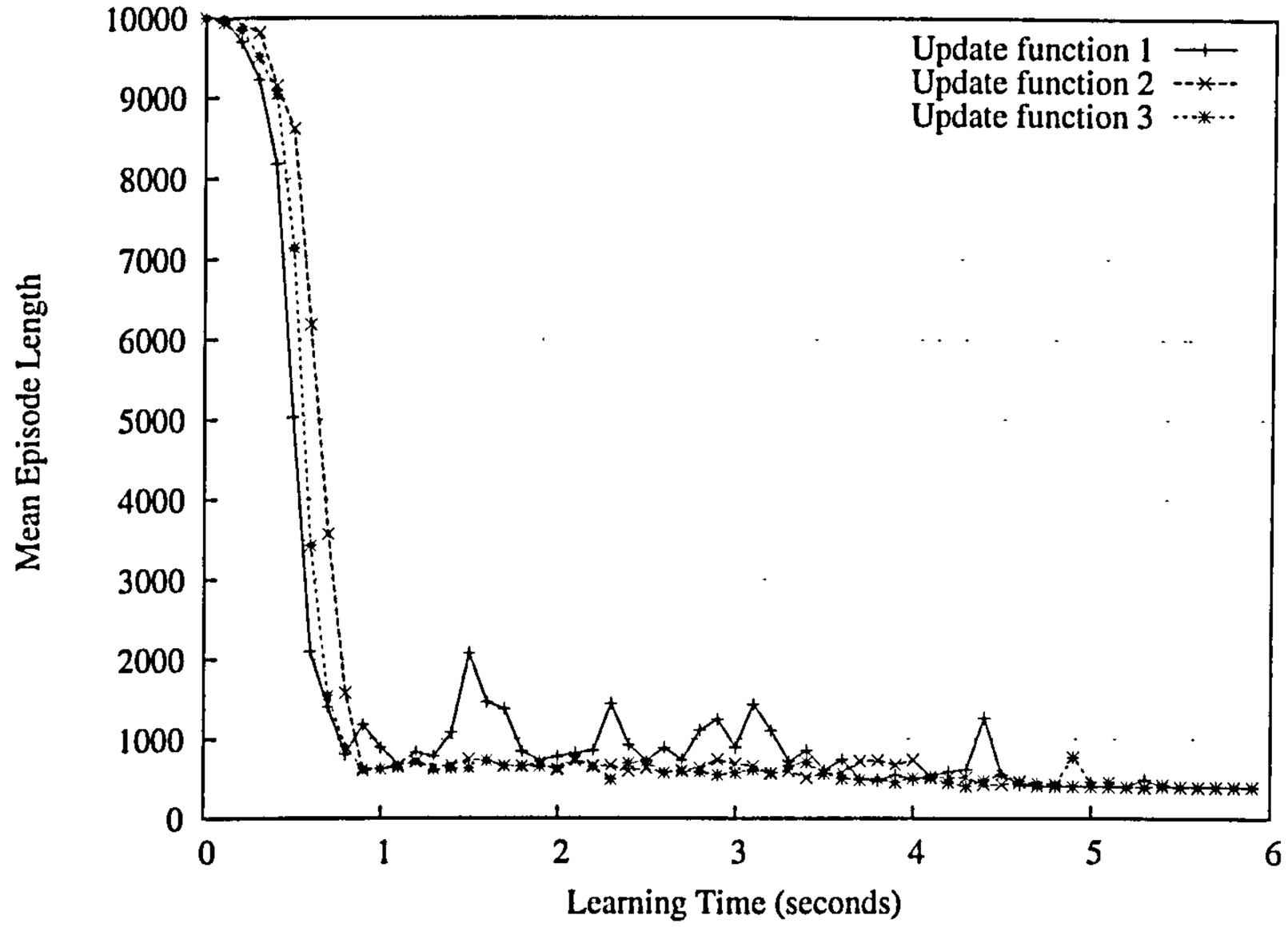


Figure 6.12: Comparing asynchronous update functions with 8 agents in the low-difficulty Stochastic Grid World task.

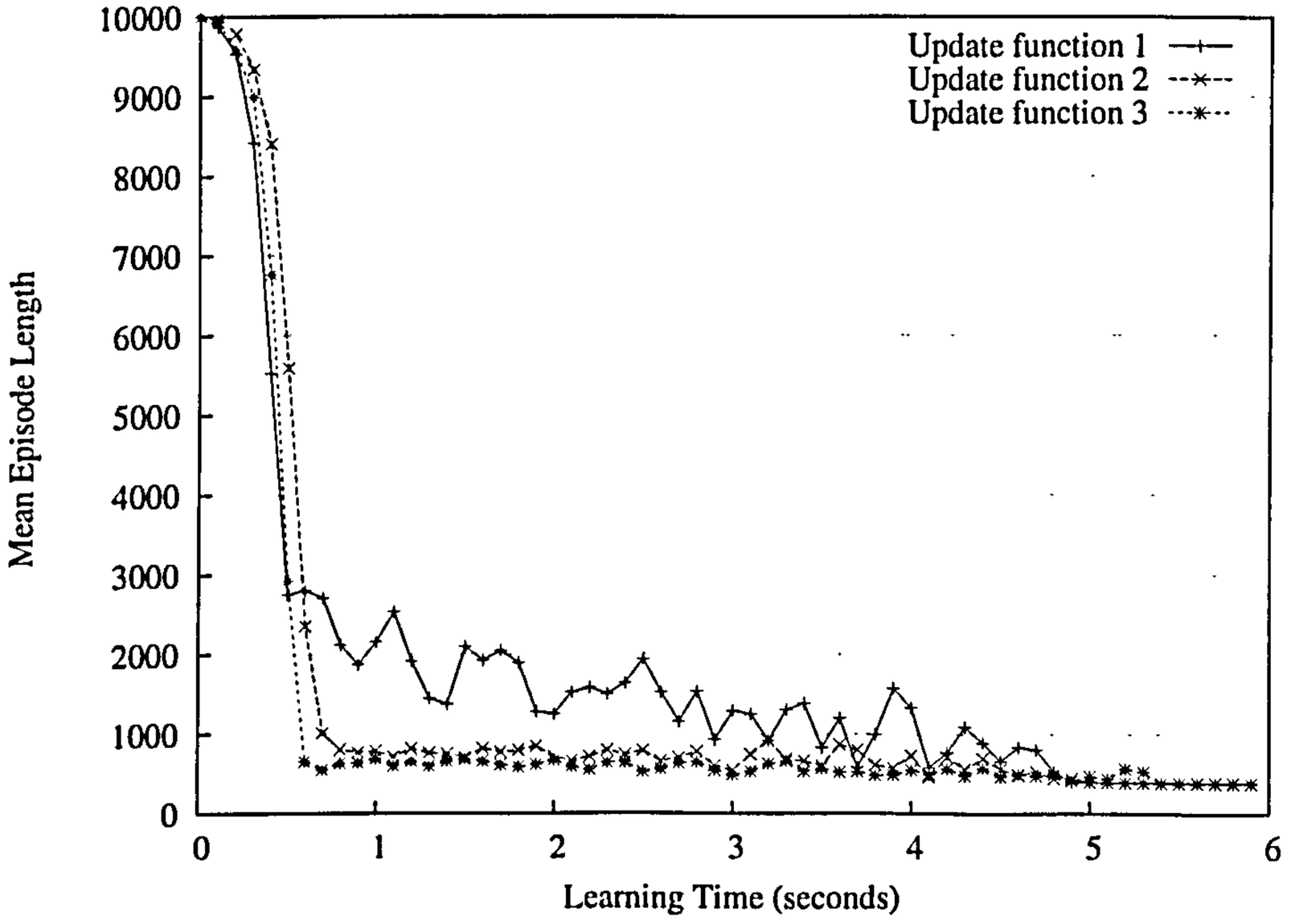


Figure 6.13: Comparing asynchronous update functions with 16 agents in the low-difficulty Stochastic Grid World task.

$\lambda = 0.9$ and $\theta_{init} = 1 \times 10^{-8}$. The parameters used for the asynchronous merge method were $p = 10000$ and $f_{com} = 256$. These experimental settings are identical to those used in the evaluation of the selective merge method. The *staggered* schedule was used by the agents to time the asynchronous broadcasts.

The graphs show that learning in this domain divides clearly into two distinct stages. There is an initial stage where the agents start from zero knowledge about the environment and rapidly improve the group's performance. Then there is a stage where the improvement in performance is much more gradual, as parameters α and ϵ gradually decay towards zero, and the agents gradually settle into policies close to the optimum.

Let us examine the initial stage first. Whichever update function is used there is a similarly rapid initial improvement in performance, the rate of which increases as the number of agents is increased. There are small differences between the update functions though. Generally update function #1 produces the most rapid improvements, and update function #2 is the least rapid, although the difference between them in this area is not great.

In the latter, more gradual stage of improvement, we can observe markedly different behaviour using update function #1. As the number of agents is increased to 8 agents, and then to 16 agents, the performance of update function #1 becomes more noisy and erratic. With 16 agents in particular, there seems to be an increased probability that the learned policy will move *away* from the optimum after the initial phase of rapid convergence is over. The most likely explanation for this behaviour is that as the number of agents is increased, the likelihood of two or more agents broadcasting simultaneously increases. Since update function #1 does not eliminate overshooting caused by simultaneous transmission of identical changes, convergence towards the optimum is badly affected. In contrast, the agents using update functions #2 and #3 remain very close to the optimal policy during the latter phase of improvement.

Stochastic Grid World (high-difficulty)

Experiments were also carried out using the high-difficulty Stochastic Grid World task. Graphs showing the results for these experiments are given in Figures 6.14–6.17.

In the high-difficulty grid world reward function #1 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 1.0$, $\lambda = 0.95$ and $\theta_{init} = 0$. The parameters used for the asynchronous

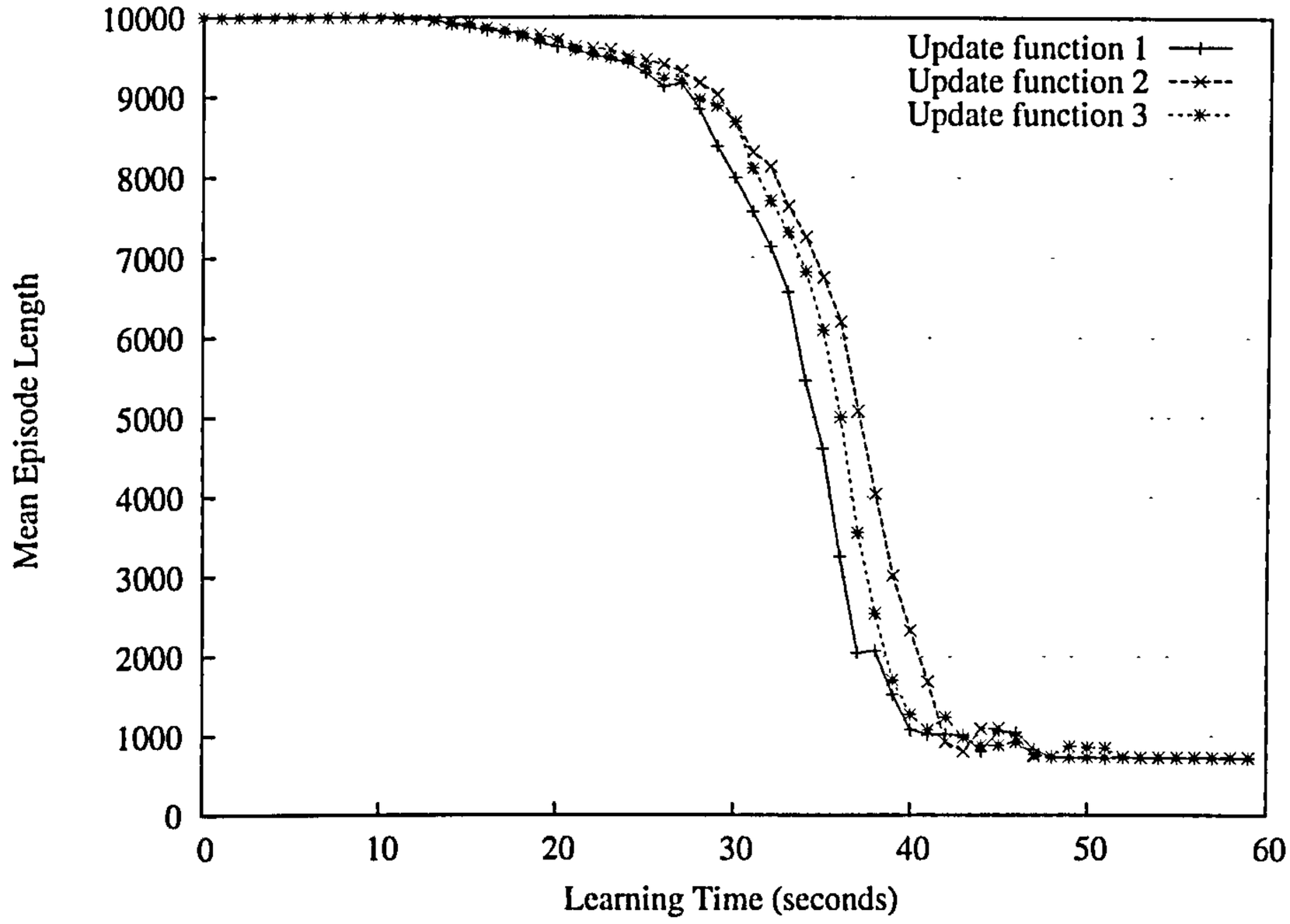


Figure 6.14: Comparing asynchronous update functions with 2 agents in the high-difficulty Stochastic Grid World task.

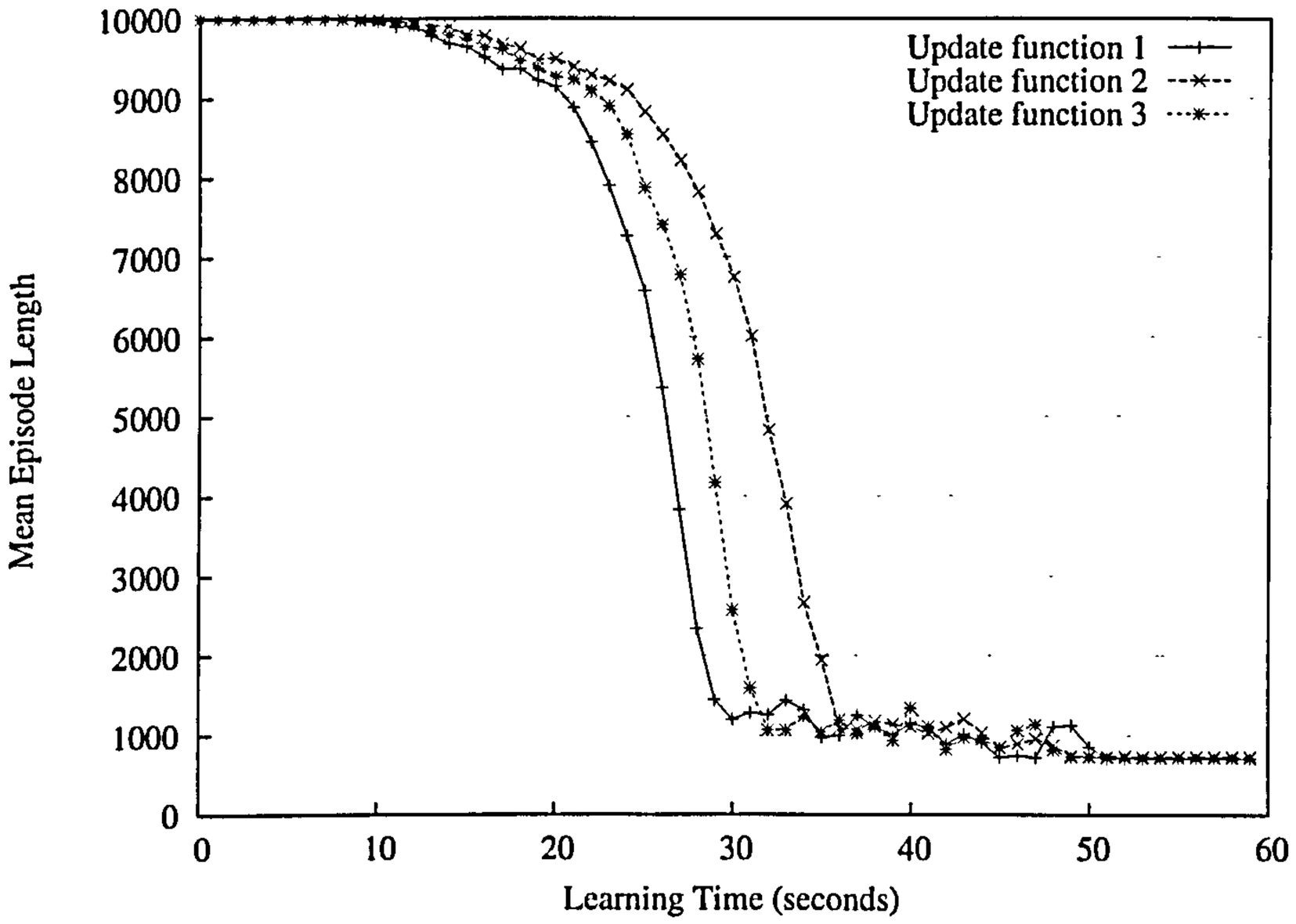


Figure 6.15: Comparing asynchronous update functions with 4 agents in the high-difficulty Stochastic Grid World task.

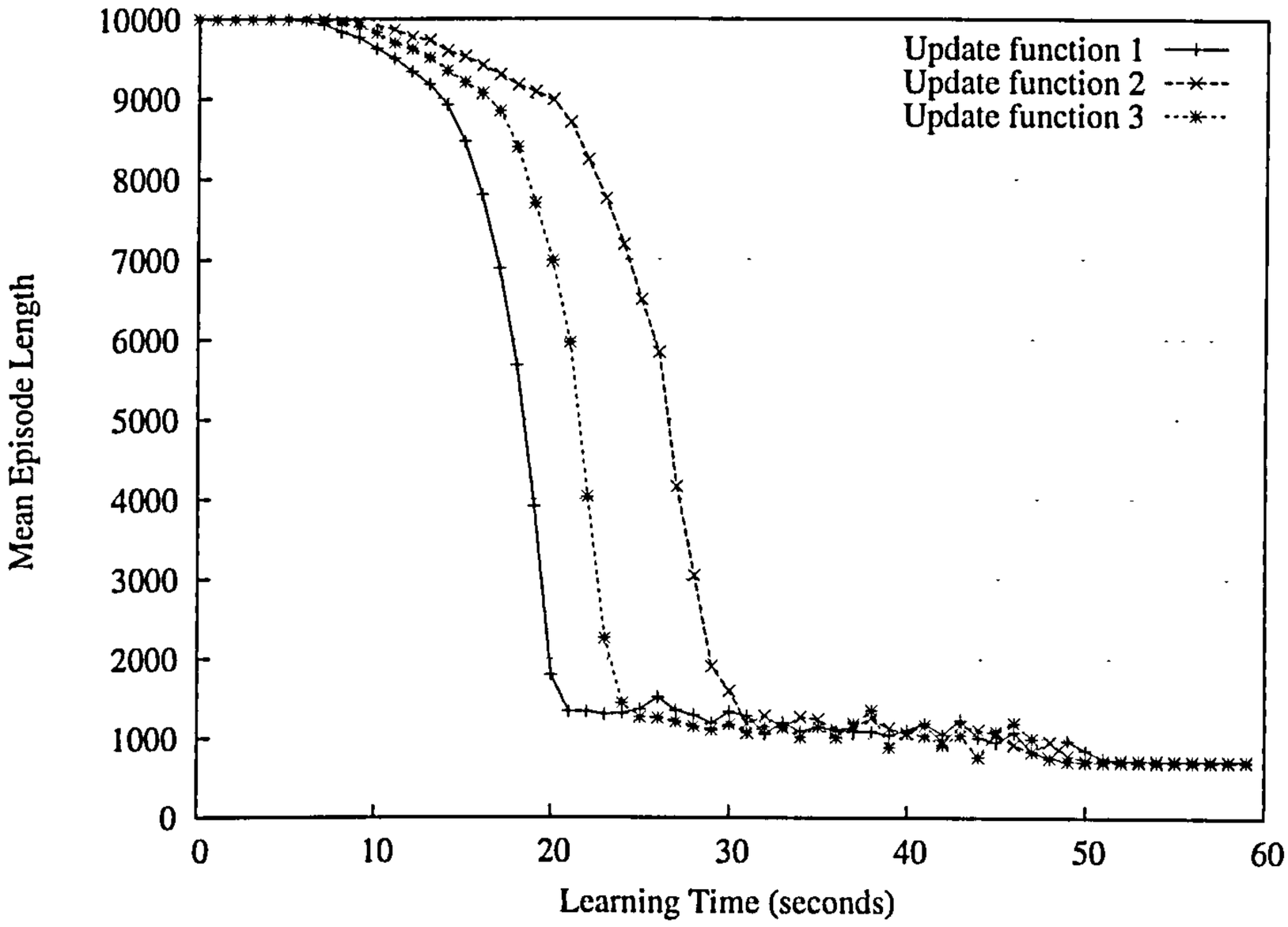


Figure 6.16: Comparing asynchronous update functions with 8 agents in the high-difficulty Stochastic Grid World task.

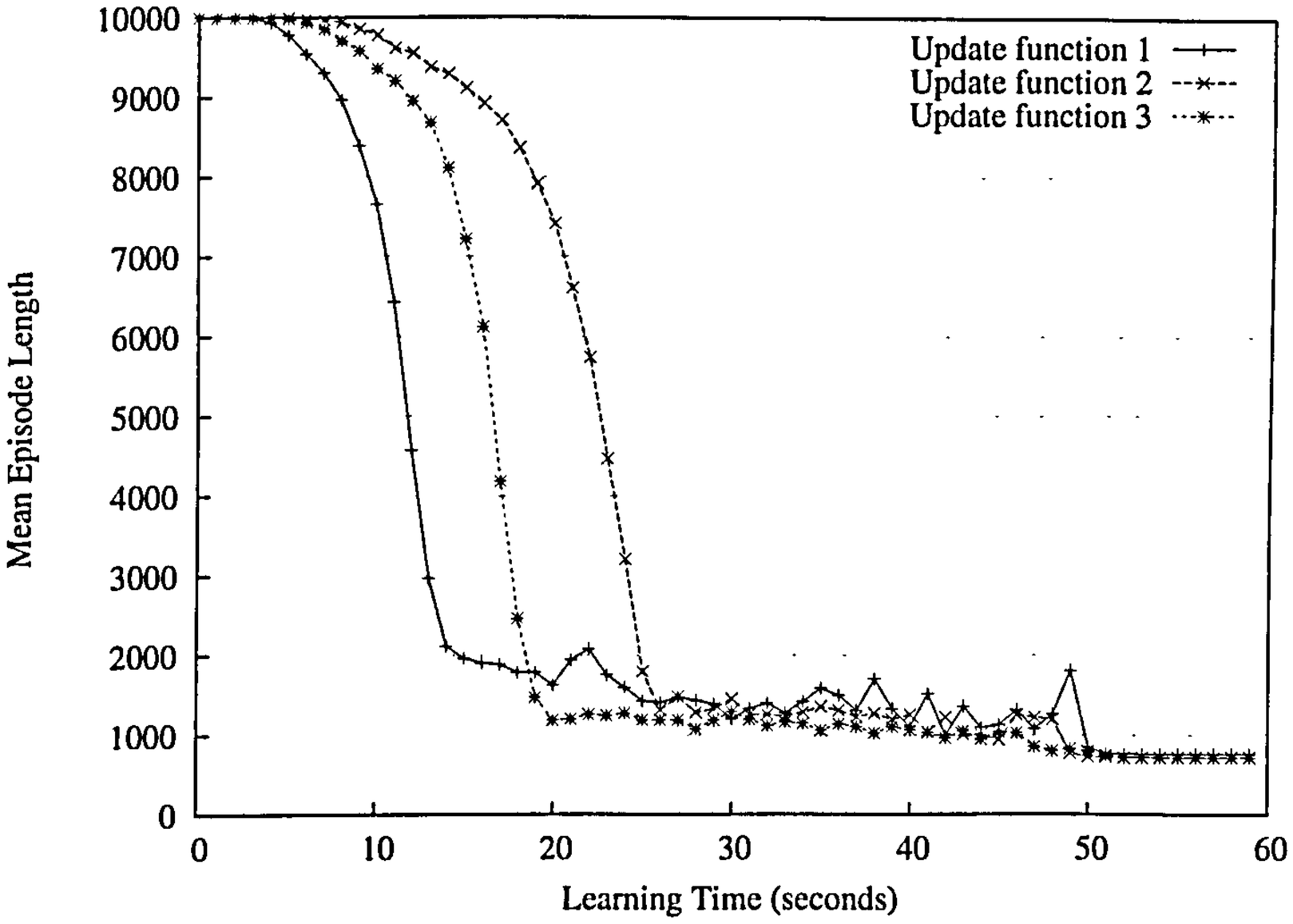


Figure 6.17: Comparing asynchronous update functions with 16 agents in the high-difficulty Stochastic Grid World task.

merging algorithm were $p = 100,000$ and $f_{com} = 1024$. These experimental settings are identical to those used in the evaluation of the selective merge method. The staggered schedule was used by the agents to time the asynchronous broadcasts.

The results for the high-difficulty grid world follow pattern which is similar to that observed for the low-difficulty grid world, but the size of the differences between the update functions are different in two key aspects.

In terms of the initial, rapid improvement in performance, there are much larger differences between the update functions now, and the separation of the learning curves is much clearer than it was for the low-difficulty grid world. Update function #1 produces the most rapid improvement, with a performance that is clearly better than update function #3. Update function #2 produces the least rapid improvement, performing significantly worse than both of the others. The differences between the update functions are relatively small for 2 agents, but become much greater as the number of agents is increased.

The increased variation during the latter stage of gradual improvement performance is now only clearly evident for update function #1 when 16 agents are used. Even so, the increased variation seems to have a much lesser effect, and does not appear to prevent a good policy being achieved after 50 seconds. It is unclear why the size of this effect is reduced. It could be simply that learning in the larger problem size is affected less by overshooting the expected weight values. Alternatively, it could be related to the fact that we used different reward functions in the two experiments (in the low-difficulty experiment, the only non-zero reward is given when the goal is reached.)

Pole Balancing

The graphs in Figures 6.18–6.21 allow the results for the three update functions to be compared in the Pole-Balancing task. In contrast to the Stochastic Grid World experiments, the agents do not all reach the same near-optimal policy quality. Instead the available time is fixed at 1.0s, and the group of agents tries to learn the highest quality policy that can be achieved in the available time. In this series of experiments, reward function #1 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 20,000 steps. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.25$, $\epsilon_0 = 0.2$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$, $\lambda = 0.5$ and $\theta_{init} = 0$. The parameters used for the asynchronous merge method were $p = 2000$ and $f_{com} = 128$. These experimental settings are identical to those used in the evaluation of the selective merge method. The staggered schedule was used by the agents to time the asynchronous broadcasts.

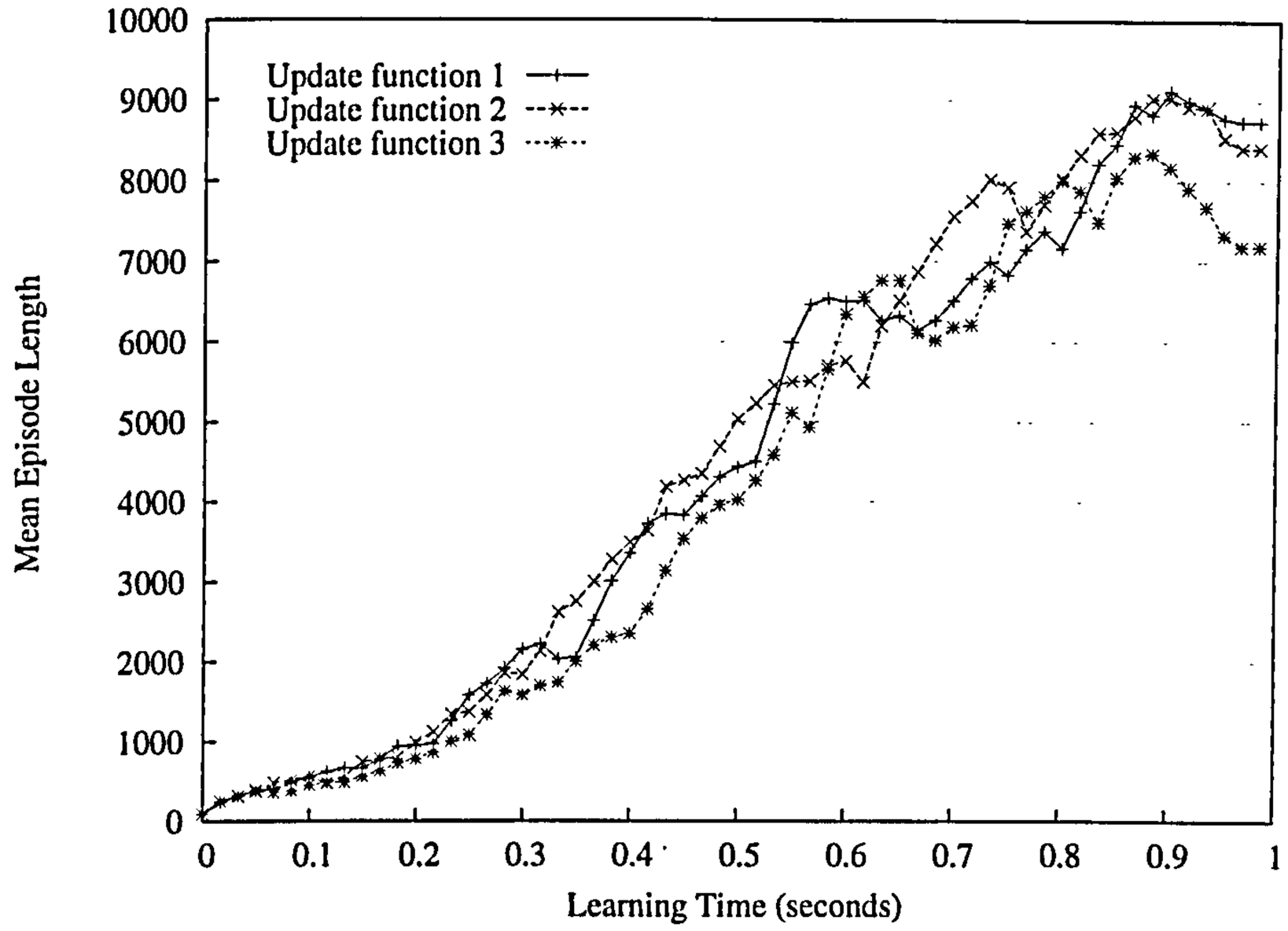


Figure 6.18: Comparing asynchronous update functions with 2 agents in the Pole-Balancing task.

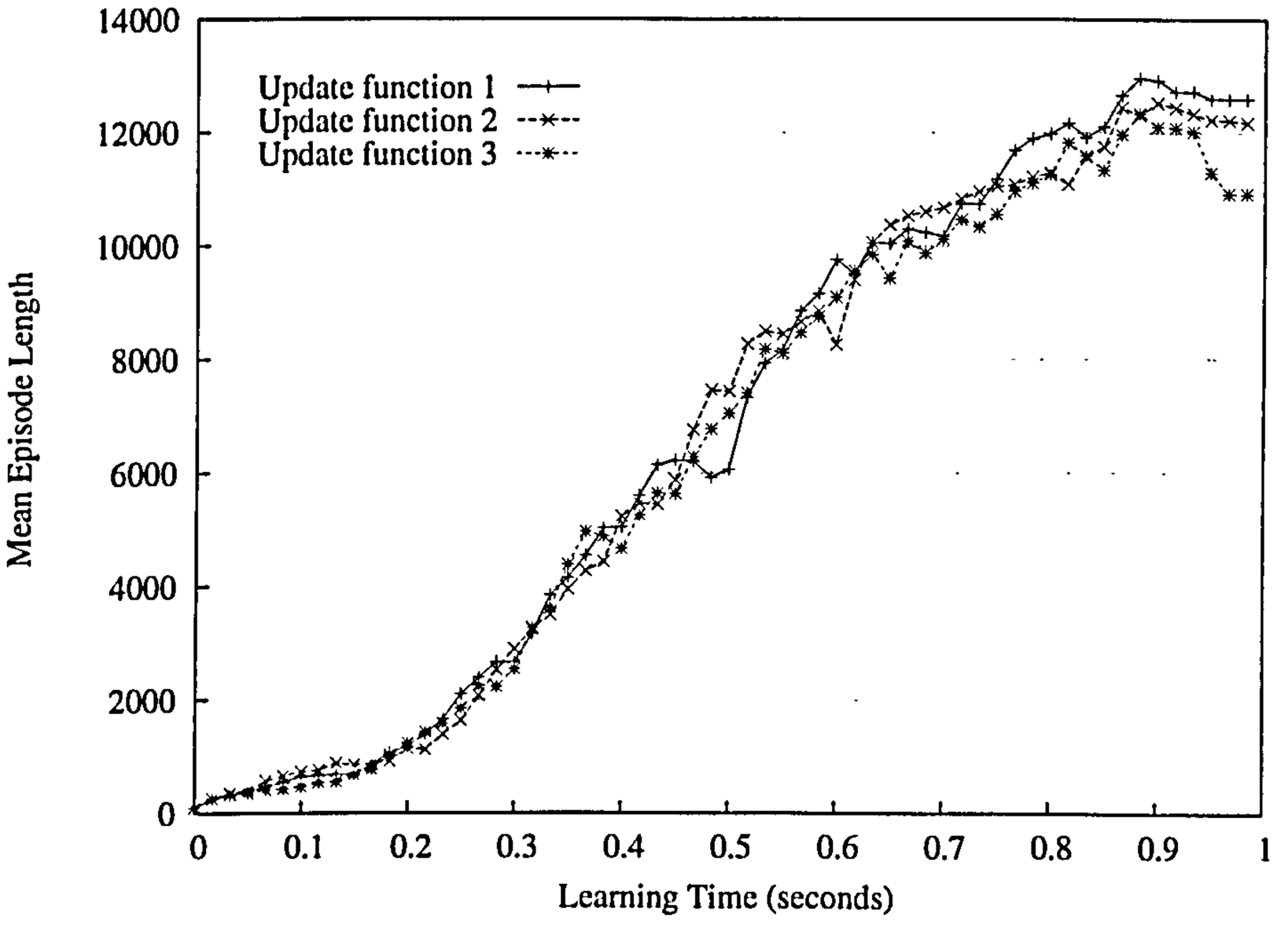


Figure 6.19: Comparing asynchronous update functions with 4 agents in the Pole-Balancing task.

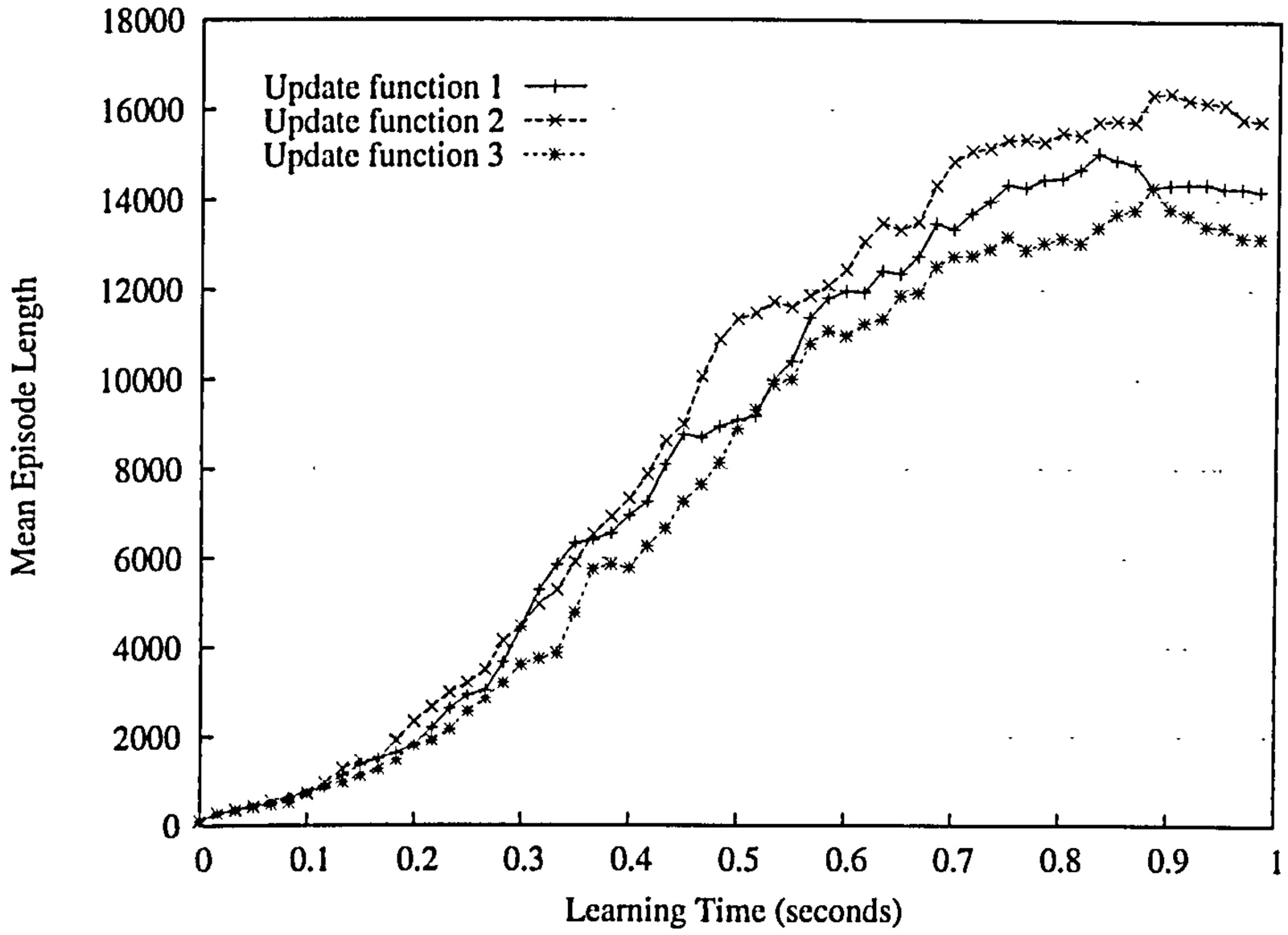


Figure 6.20: Comparing asynchronous update functions with 8 agents in the Pole-Balancing task.

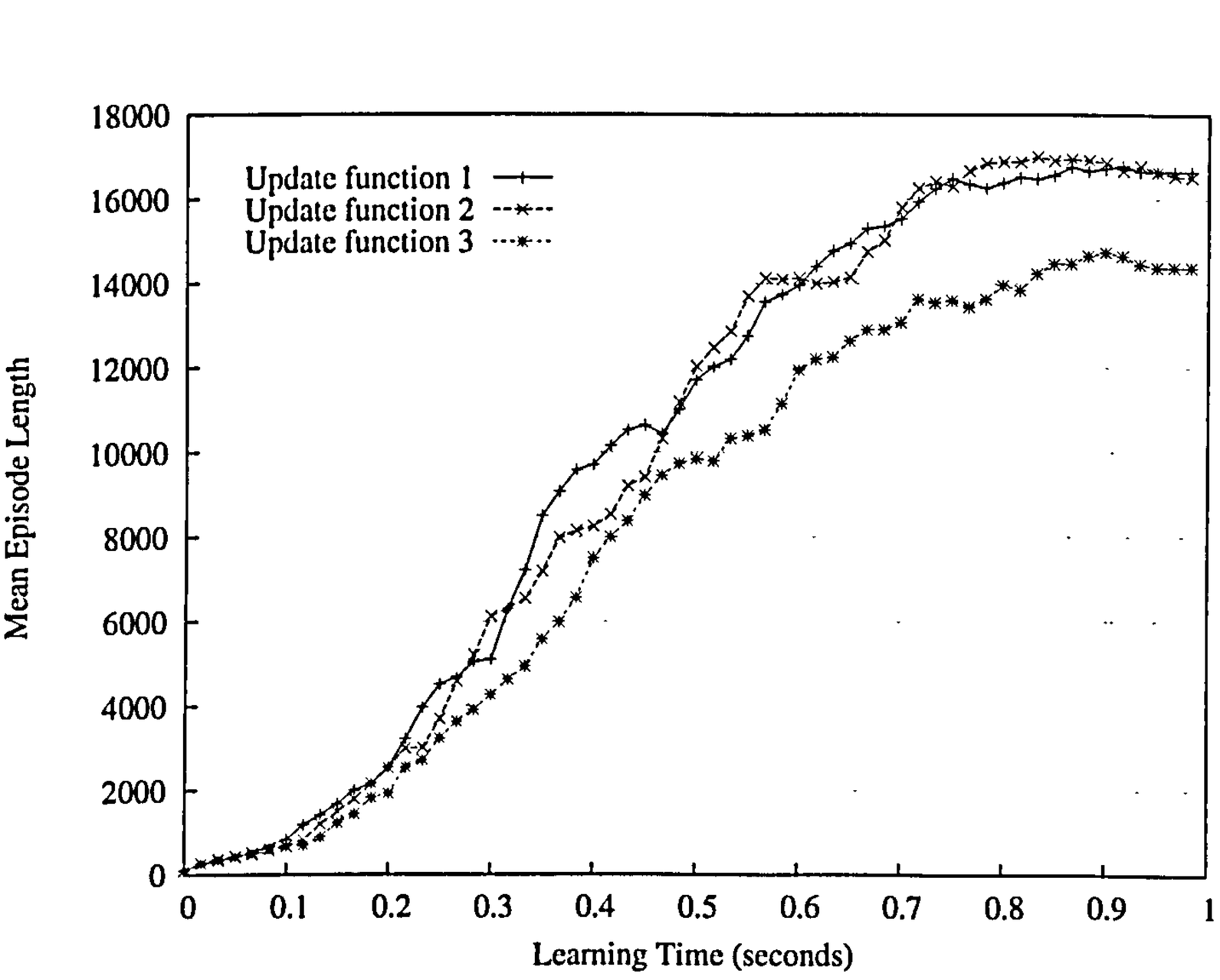


Figure 6.21: Comparing asynchronous update functions with 16 agents in the Pole-Balancing task.

The relative performances of the 3 update functions in the Pole-Balancing task exhibit a different pattern than that seen in the Stochastic Grid World tasks. Update function #3 is now consistently the worst out of the three functions, although the difference in the average quality achieved at the end of the run is not huge. It is difficult to determine which of the other update functions is the better performer. It is only in the case of 8 agents (in Figure 6.20) that one of them, update function #2, clearly outperforms the other. For other numbers of agents, there is not a significant difference between them. No evidence of the increased variance produced by update function #1 in the previous experiments is observed here, but it is still possible that overshooting effects are affecting the overall performance of update function #1.

Mountain-Car

Figures 6.22–6.25 show results for the three update functions for different numbers of agents in the Mountain-Car task. In this series of experiments, reward function #2 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 500 steps. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.5$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 0.0001$. The parameters used for the asynchronous merge method were $p = 2000$ and $f_{com} = 128$. Since the policy quality over a given interval is strongly dependent on the exploration parameter ϵ, binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average quality under 145 over the set of 100 runs. These experimental settings are identical to those used in the evaluation of the selective merge method. The staggered schedule was used by the agents to time the asynchronous broadcasts.

In the results for the Mountain-Car task, we observe a pattern in the relative performance of the update functions that is closer to what was observed for the Stochastic Grid World task than what was observed in the Pole-Balancing task. Since the gradual improvement in performance is so strongly tied to the decay of the exploration parameter, it is difficult in some of these graphs to identify which of the update functions has performed the best. However, we can draw some general conclusions from the graphs. For all numbers of agents, the asynchronous method using update function #2 requires the most time to converge to a policy of the specified quality. Most often it is update function #1 which requires the least time, although in the 4-agent case it is update function #3 which produces the best performance. The performance of update function #3 seems relatively poor for small numbers of agents, but almost as good as update function #1 for larger

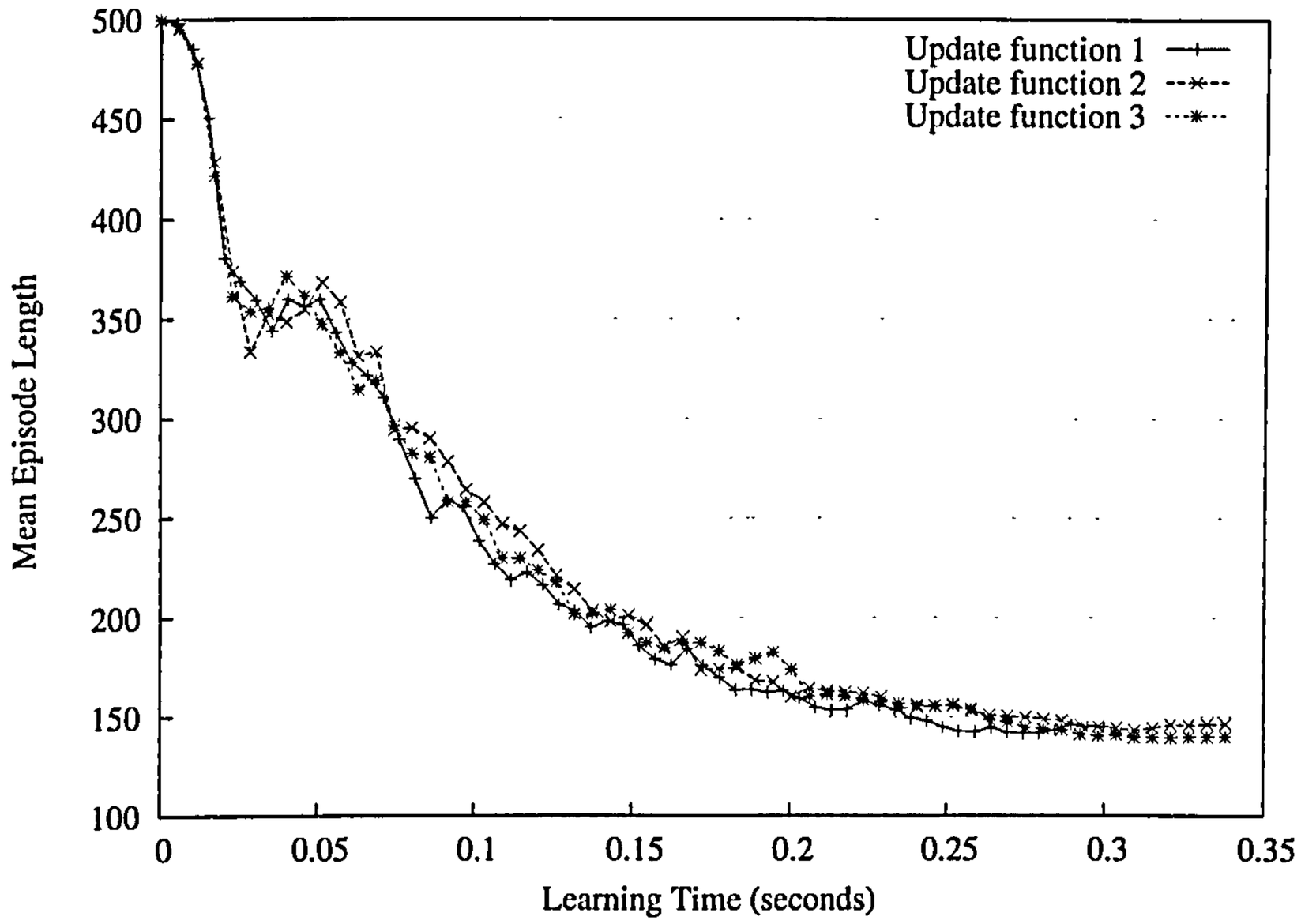


Figure 6.22: Comparing asynchronous update functions with 2 agents in the Mountain-Car task.

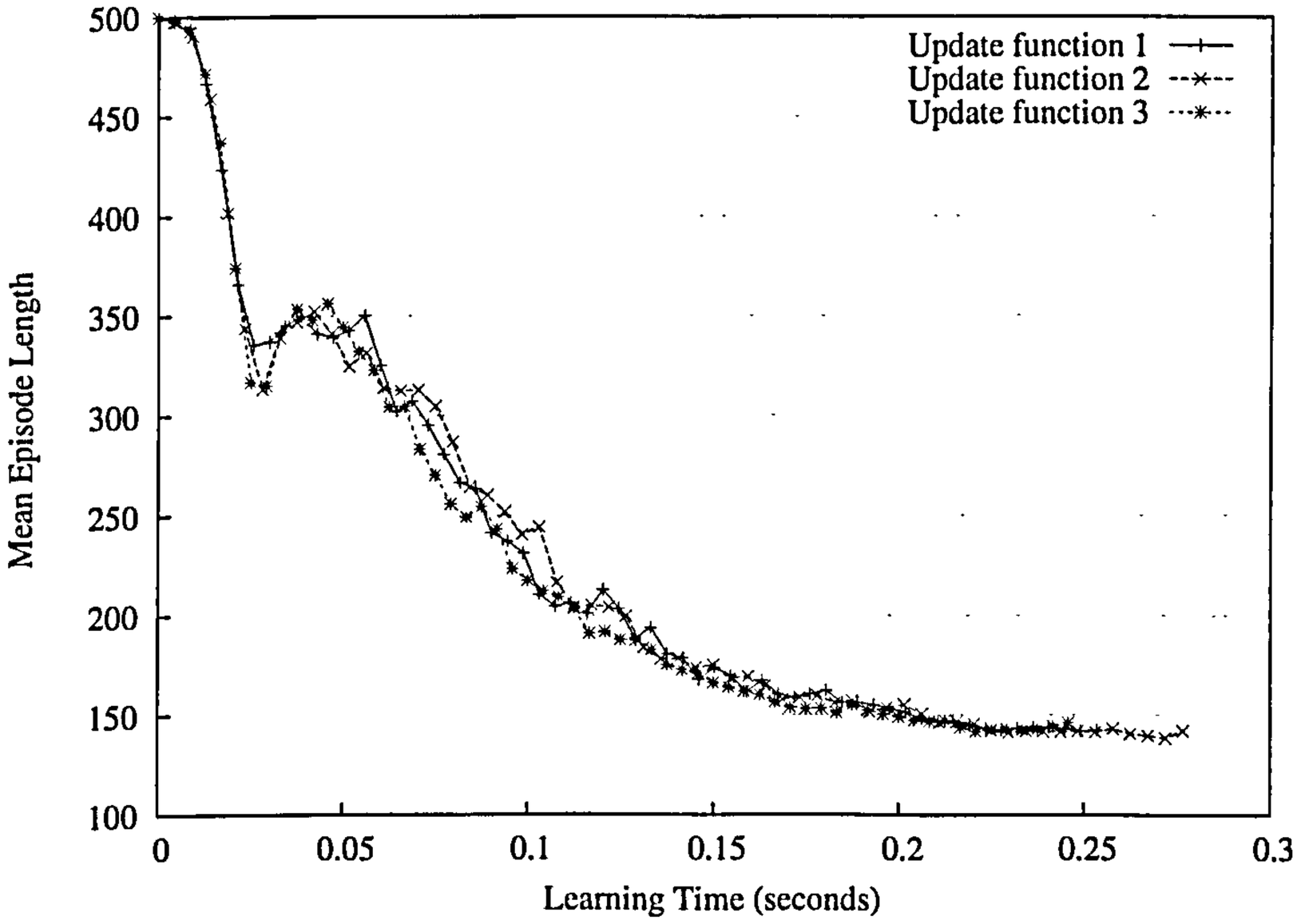


Figure 6.23: Comparing asynchronous update functions with 4 agents in the Mountain-Car task.

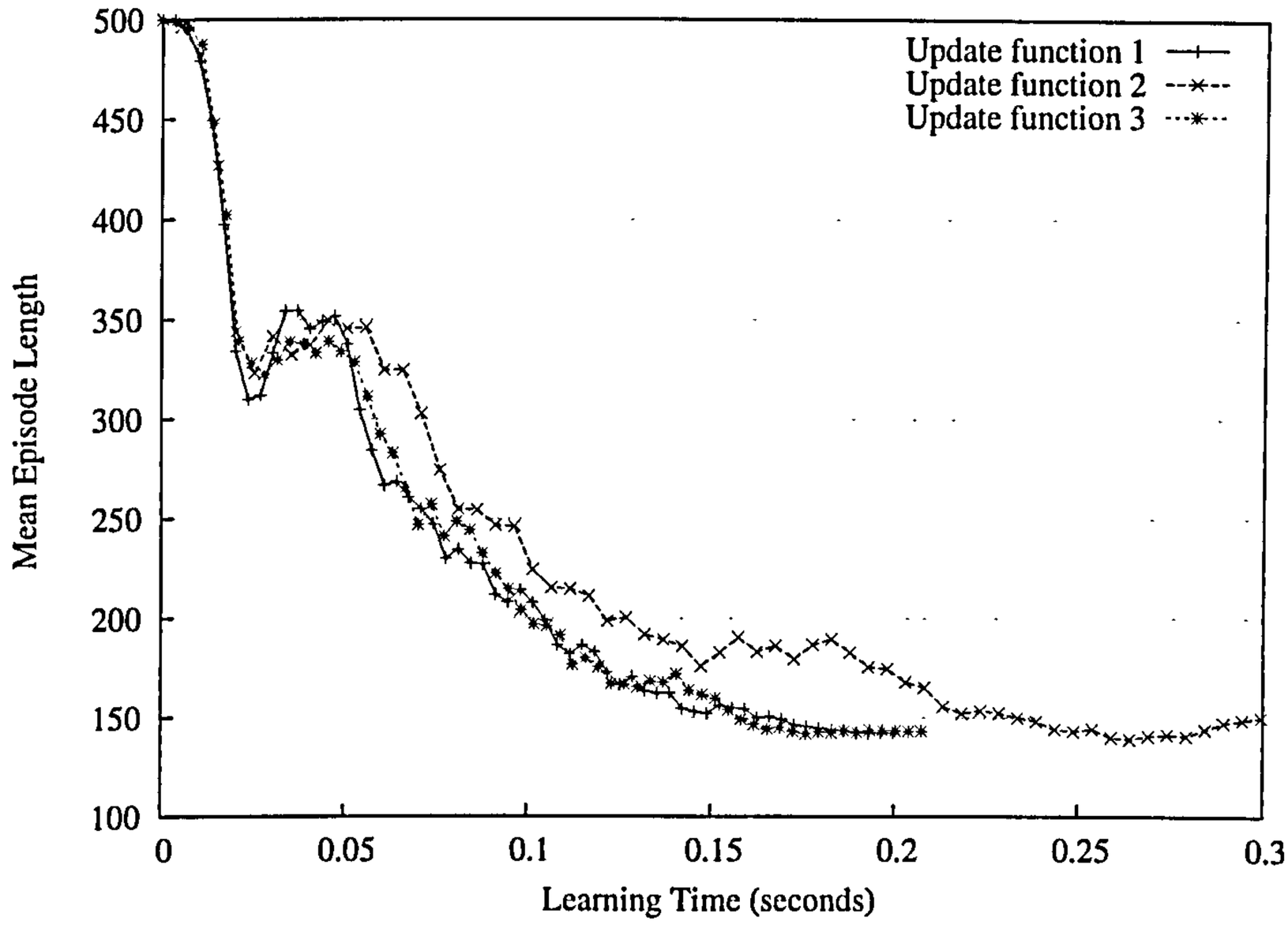


Figure 6.24: Comparing asynchronous update functions with 8 agents in the Mountain-Car task.

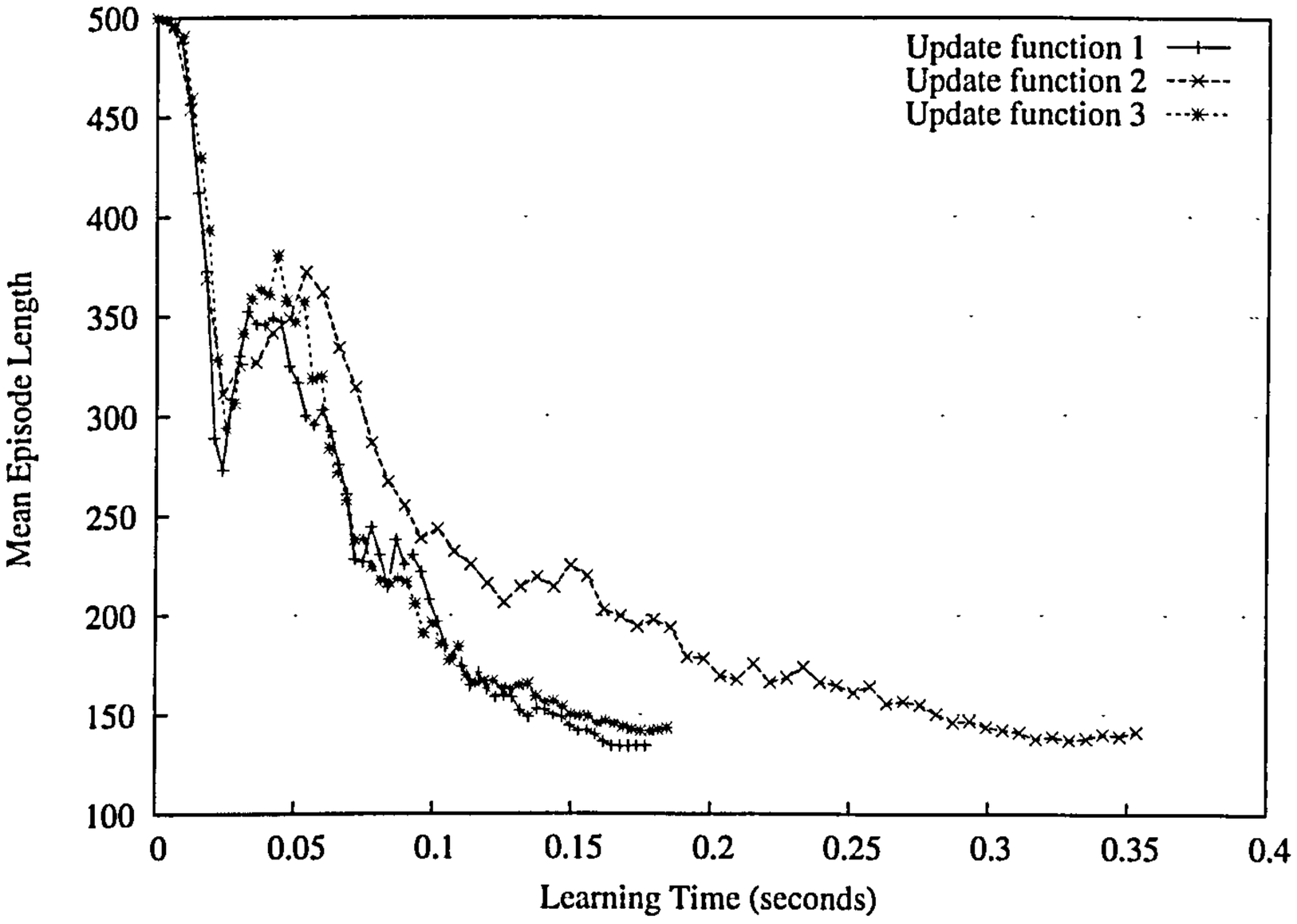


Figure 6.25: Comparing asynchronous update functions with 16 agents in the Mountain-Car task.

numbers of agents. In the 16-agent case, the learning curves are separated the most, with update functions #1 and #3 requiring only slightly more than half the time required by update function #2 to converge.

Acrobot

Graphs for the performance of the different update functions in the Acrobot task are shown in Figures 6.26–6.29. These experiments used reward function #1, with the results being averaged over 100 runs. Episodes were terminated if they reached 600 steps. RL parameters α and ϵ decay linearly during each run, according to the parameters $\alpha_0 = 0.1$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 1.0$, $\lambda = 0.9$ and $\theta_{init} = 0$. The parameters used for the asynchronous merge method were $p = 1000$ and $f_{com} = 128$. Since the policy quality over a given interval is strongly dependent on the exploration parameter ϵ, binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average quality under 140 over the set of 100 runs. These experimental settings are identical to those used in the evaluation of the selective merge method. The staggered schedule was used by the agents to time the asynchronous broadcasts.

The performance of the three evaluation functions is essentially identical when there are only 2 agents. However, as the total number of agents is increased, a clear pattern emerges, and the differences between the update functions become more pronounced. Update function #1 now consistently produces the best performance, with update function #2 producing the worst performance, and update function #3 being somewhere between the two others.

Summary of Evaluation

A qualitative summary of the results presented in this section is shown in Table 6.1, which lists the best performing update function from each of the graphs shown in Figures 6.10–6.29. Broadly speaking, the best performing update function is the one which makes the most rapid progress towards a high quality problem solution over the course of a parallel run.

From Table 6.1 it is fairly clear that the asynchronous merge method using update function #1 has the greatest potential out of the methods proposed in this chapter. However, there are a number of effects that remain unexplained. The most significant unanswered question is "Why does update function #2 produce such good results in the Pole-Balancing domain, but such bad results in all the other domains?" The answer may be related to the fact that Pole-Balancing is the

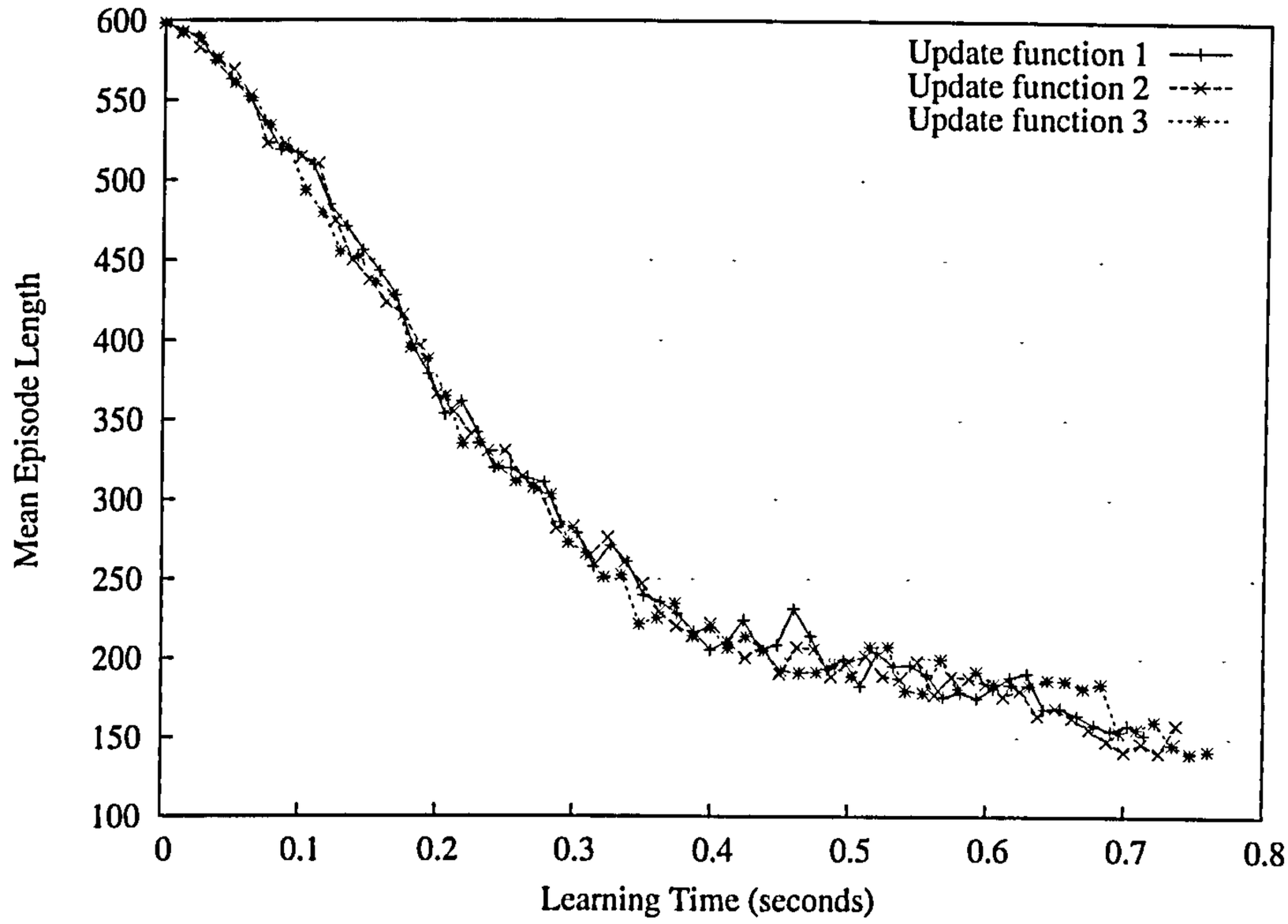


Figure 6.26: Comparing asynchronous update functions with 2 agents in the Acrobot task.

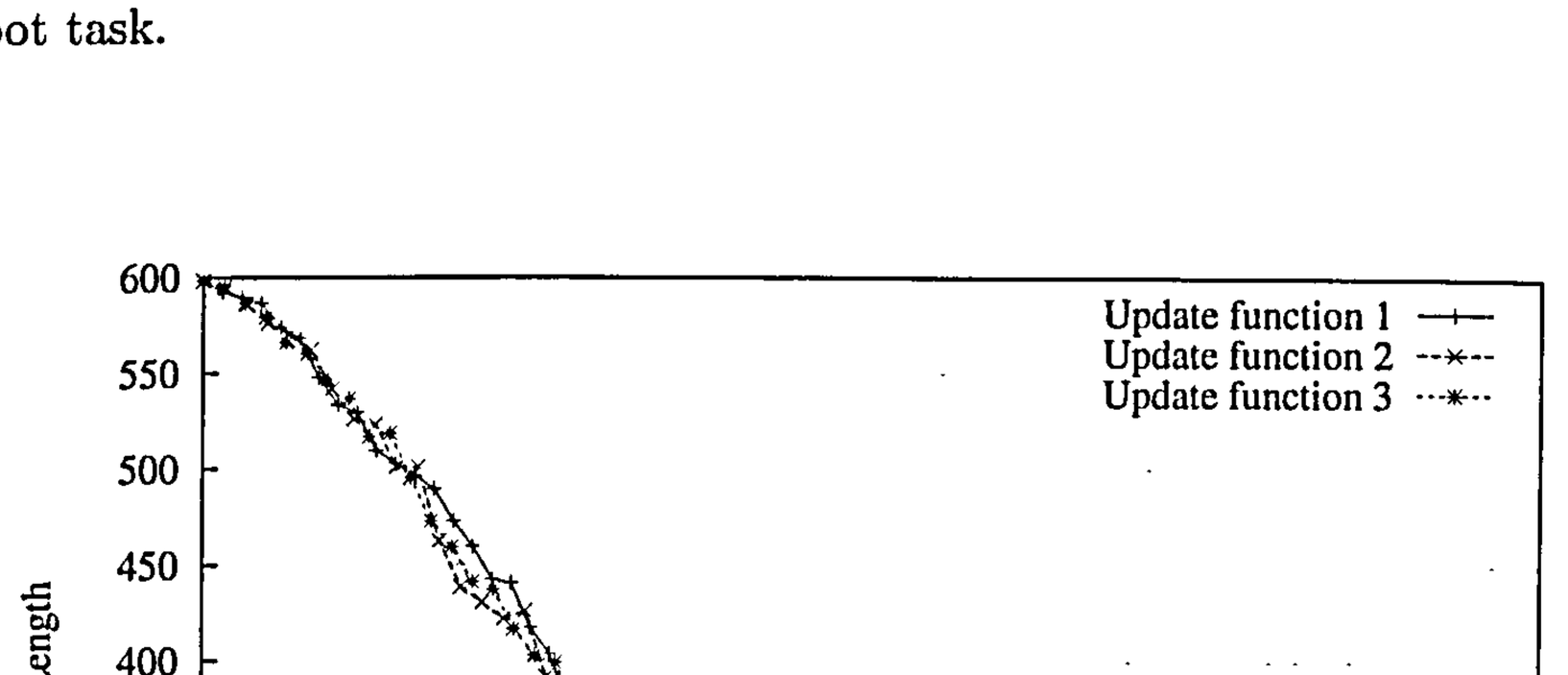


Figure 6.27: Comparing asynchronous update functions with 4 agents in the Acrobot task.

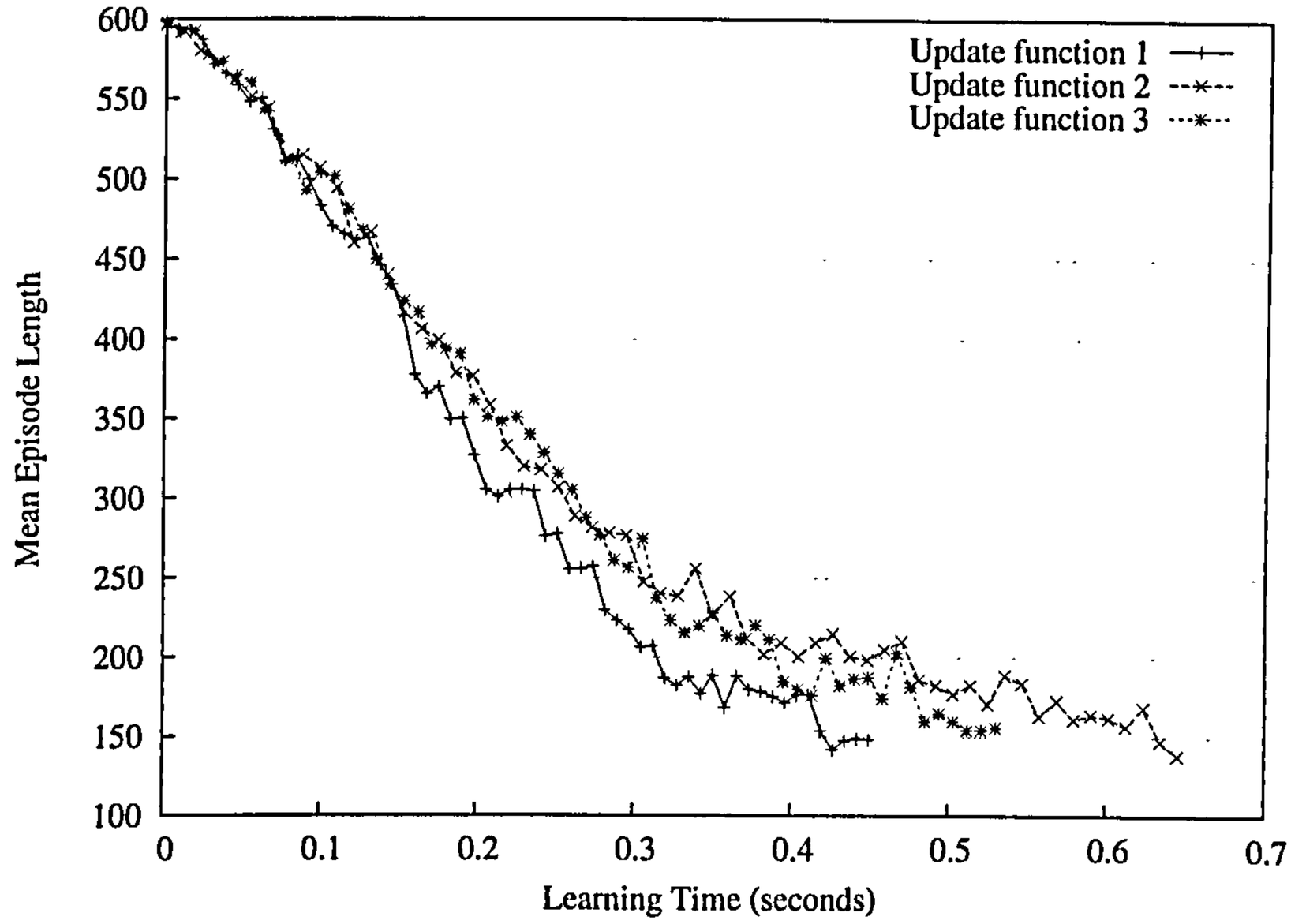


Figure 6.28: Comparing asynchronous update functions with 8 agents in the Acrobot task.

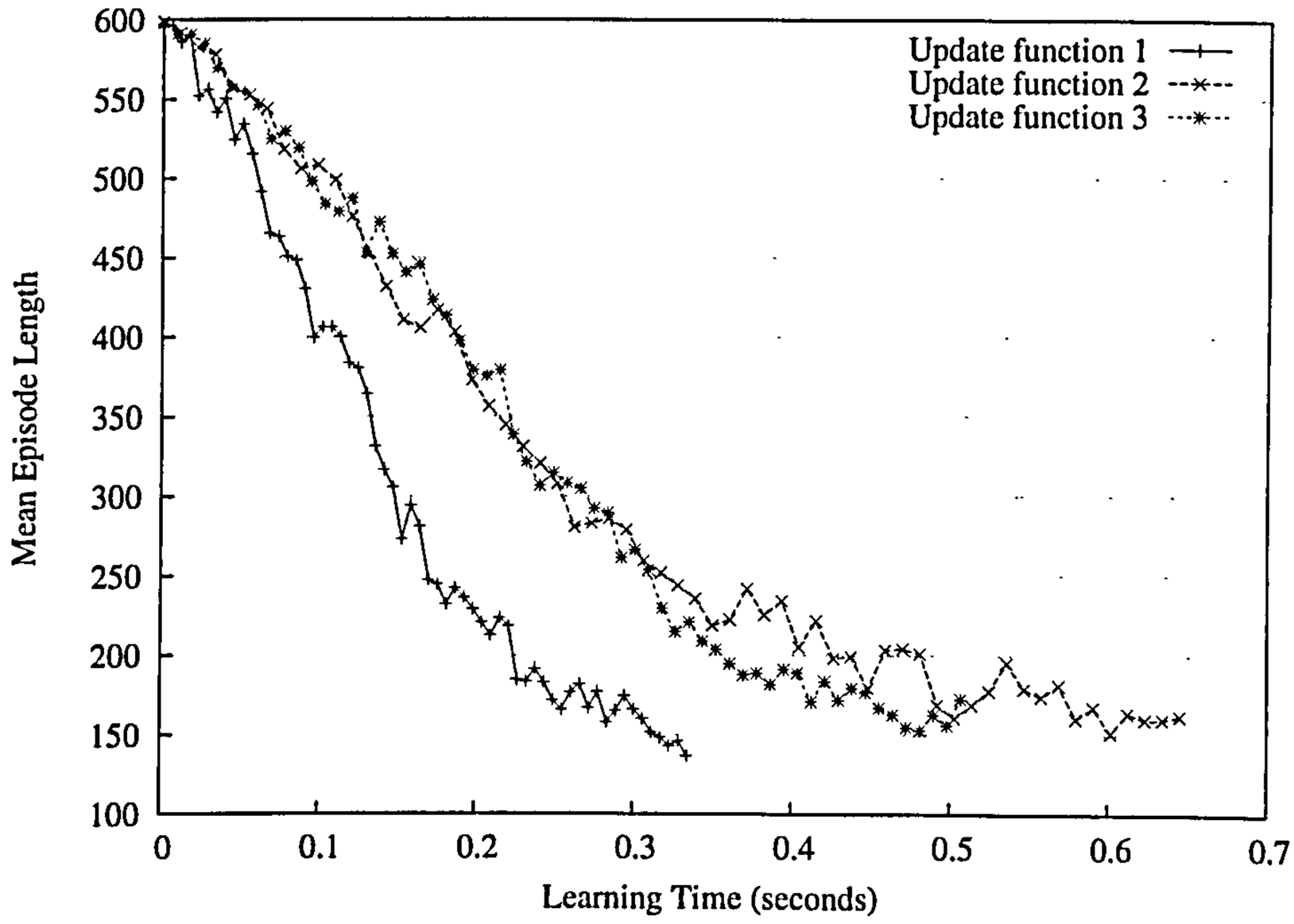


Figure 6.29: Comparing asynchronous update functions with 16 agents in the Acrobot task.

Domain	Number of agents			
	2	4	8	16
Grid world (low-difficulty)	#1/#3	#1	#1	#1/#3
Grid world (high-difficulty)	#1	#1	#1	#1
Pole-Balancing	#1/#2	#1/#2	#2	#1/#2
Mountain-Car	#1	#3	#1	#1
Acrobot	#1	#1	#1	#1

Table 6.1: Lists the best performing update function(s) for each combination of a domain and a number of agents.

only domain considered here that is not *goal-oriented.* In all the other domains, the agents want to reach a terminal state as quickly as possibly. In the Pole-Balancing task, terminal states must be avoided for as long as possible by keeping the pole balanced. This gives the Pole-Balancing task a different character from the other domains, which may produce a different distribution of weight changes in the agents' messages.

Another effect which could be investigated further is the tendency of update function #1 to increase variance in the estimates of feature values. This effect can be observed most clearly in the 16-agent experiments in the Stochastic Grid World (see Figures 6.13 and 6.17). This increase in variance while the ϵ and α parameters are decaying could result in the agents settling in a worse policy on average, but this evaluation has not clearly established that this is the case. What has been established is that in the case of a uniform schedule, where many agents broadcast their changes simultaneously, the asynchronous method based on update function #1 is likely not to converge (see Figure 6.9). This demonstrates the importance of keeping the agents' broadcasts well-distributed over time.

To conclude, it appears that the *cancel* mechanism (used in update functions #1 and #3) is necessary for achieving fast convergence without overshooting in most of the domains evaluated here (Pole-Balancing being the exception). The *filter* mechanism (used in update functions #2 and #3) appears to slow convergence slightly by excluding some of the incoming weight changes, but is essential to ensure convergence when there is a high probability of agents broadcasting messages simultaneously. There is the possibility that more complex update functions not considered in this work could approach the convergence rate of update function #1 in most cases while retaining the safety of a *filter*-like mechanism. This remains a topic for future investigation.

6.4 Comparison with Synchronous Selective Method

In Section 6.3 a comparison was presented of the performance achieved by three candidate mechanisms for updating the VFA in the asynchronous selective method. Now that there exists some empirical evidence as to the suitability of these update mechanisms to particular domains, this section proceeds to give a direct comparison of the asynchronous selective method with the original (synchronous) selective method presented in Chapter 5. The data used to generate the following graphs was drawn from the results already reported in the individual evaluations of the methods, in Sections 5.4 and 6.3. These results are reproduced together on new graphs to facilitate a detailed comparison of the two methods. For the full details of the experimental settings used to generate these results, the reader is referred to the earlier sections.

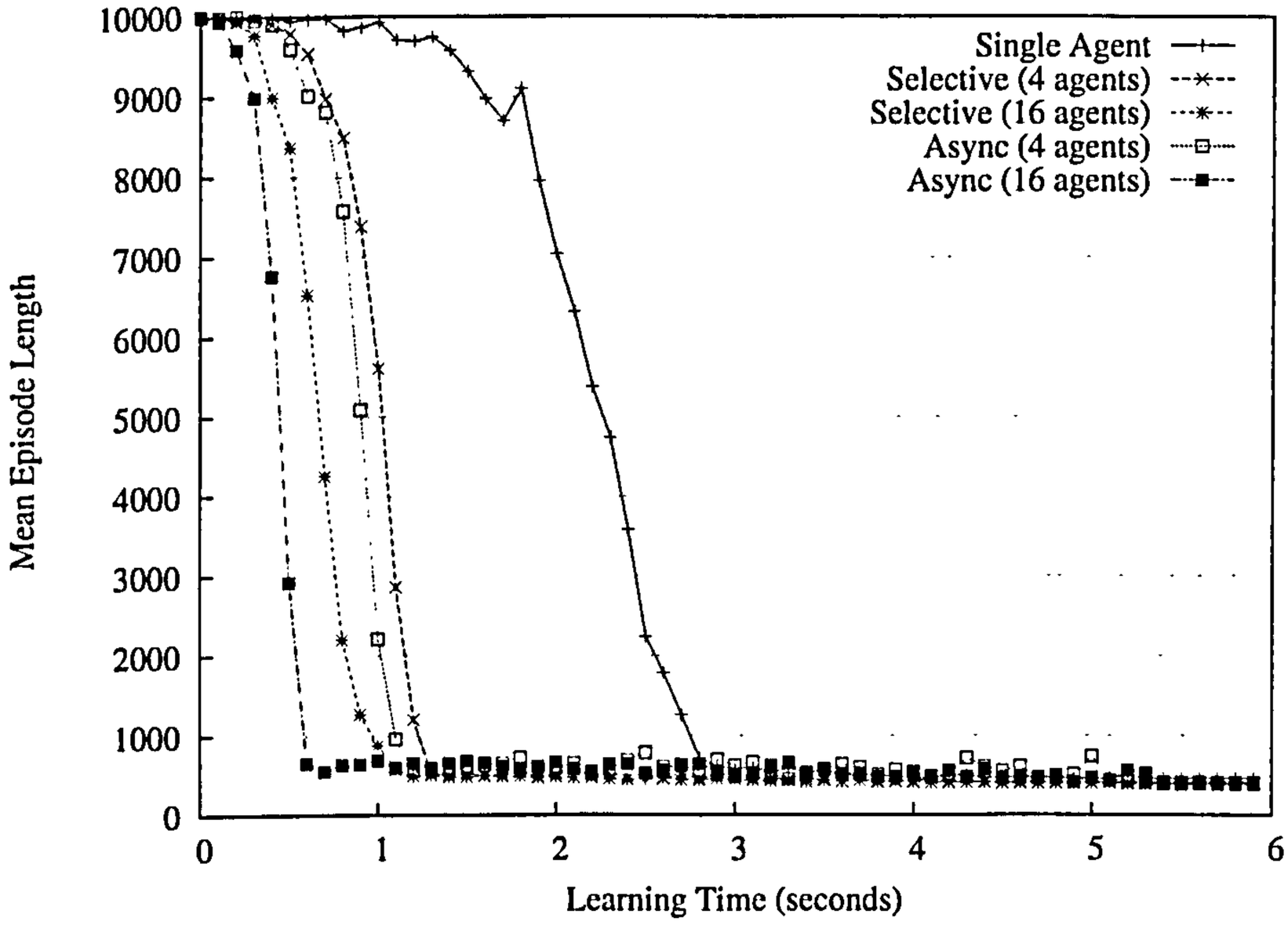


Figure 6.30: Comparing the performance of the synchronous and asynchronous selective methods in the low-difficulty Stochastic Grid World task.

Results showing the performance of a single agent, the *selective* merge method (4 and 16 agent groups), and the *asynchronous selective* merge method (4 and 16 agent groups) in the low-difficulty Stochastic Grid World task are shown in Figure 6.30. To compare the most *stable* results for each approach, the selective method used combination function #3 and the asynchronous selective method used update function #3. With both the 4 and 16 agent groups the asynchronous selective method produces the best performance out of the two parallel approaches. The

improvement in the 4 agent case is fairly modest, although this performance is probably comparable to 8 agent group using the selective method. The improvement in the 16 agent case is much more significant, with the time required to learn a high-quality policy being almost halved. In this particular domain there is a major advantage in moving to an asynchronous approach when large numbers of parallel agents are available.

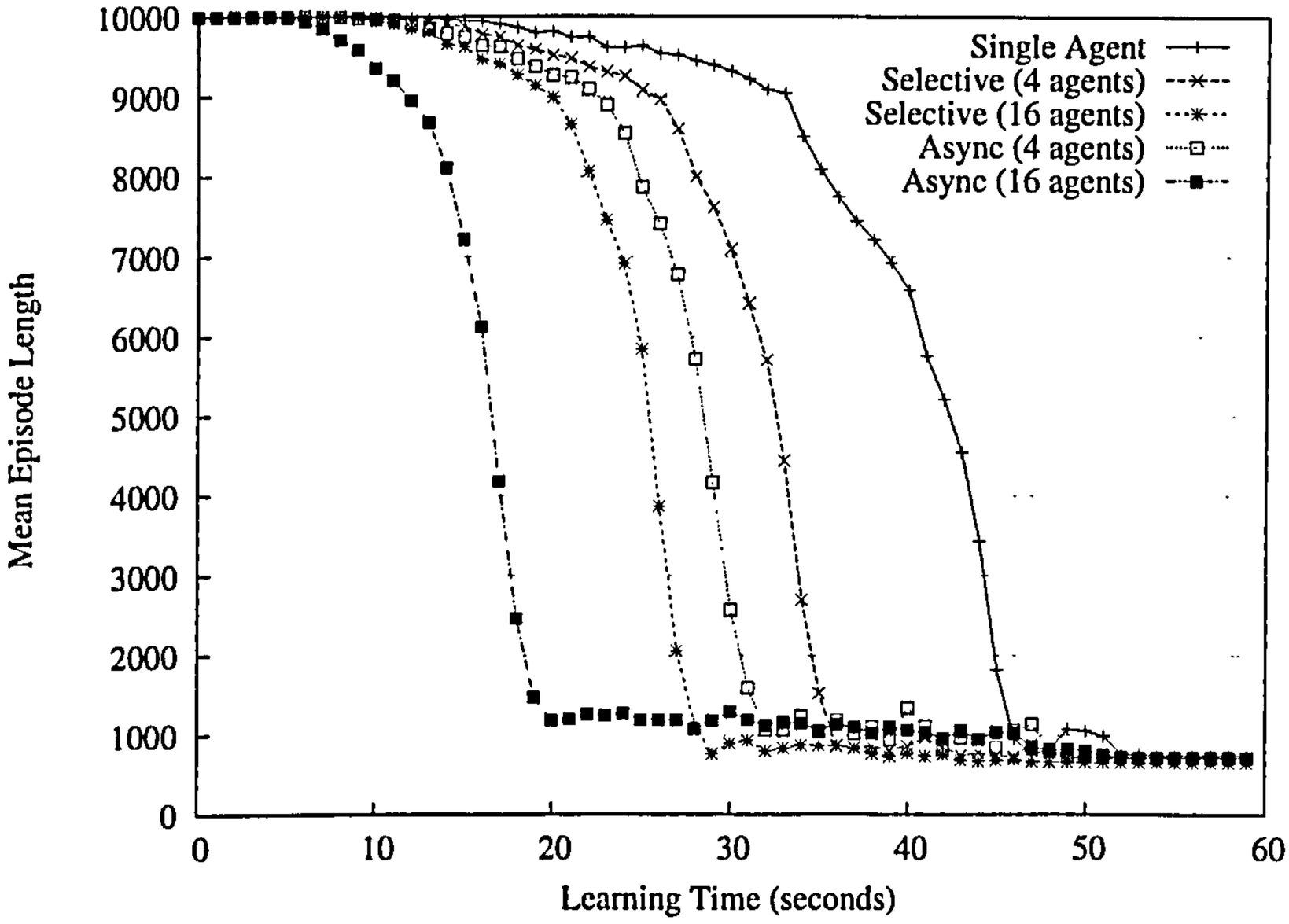


Figure 6.31: Comparing the performance of the synchronous and asynchronous selective methods in the high-difficulty Stochastic Grid World task.

Results for the high-difficulty Stochastic Grid World task (shown in Figure 6.31) exhibit a very similar pattern, although the difference in performance of the 16 agent group is not quite so large. Once again the selective method using combination function #3 was compared with the asynchronous selective method using update function #3. With both sizes of the agent group the asynchronous selective method produces the best performance out of the two parallel approaches. The improvement in performance of the 4 agent group is significant, but the largest improvement is shown by the 16 agent group, where moving to the asynchronous approach shaves off about a third of the time required to find a high-quality policy. There is again a major advantage in following the asynchronous approach.

The results for the Pole-Balancing task (shown in Figure 6.32) were generated with the priority of achieving the best possible performance, since the stability of the update mechanisms seemed to be much less of a factor than in the Stochastic Grid World tasks. To this end, the selective method was used with combination

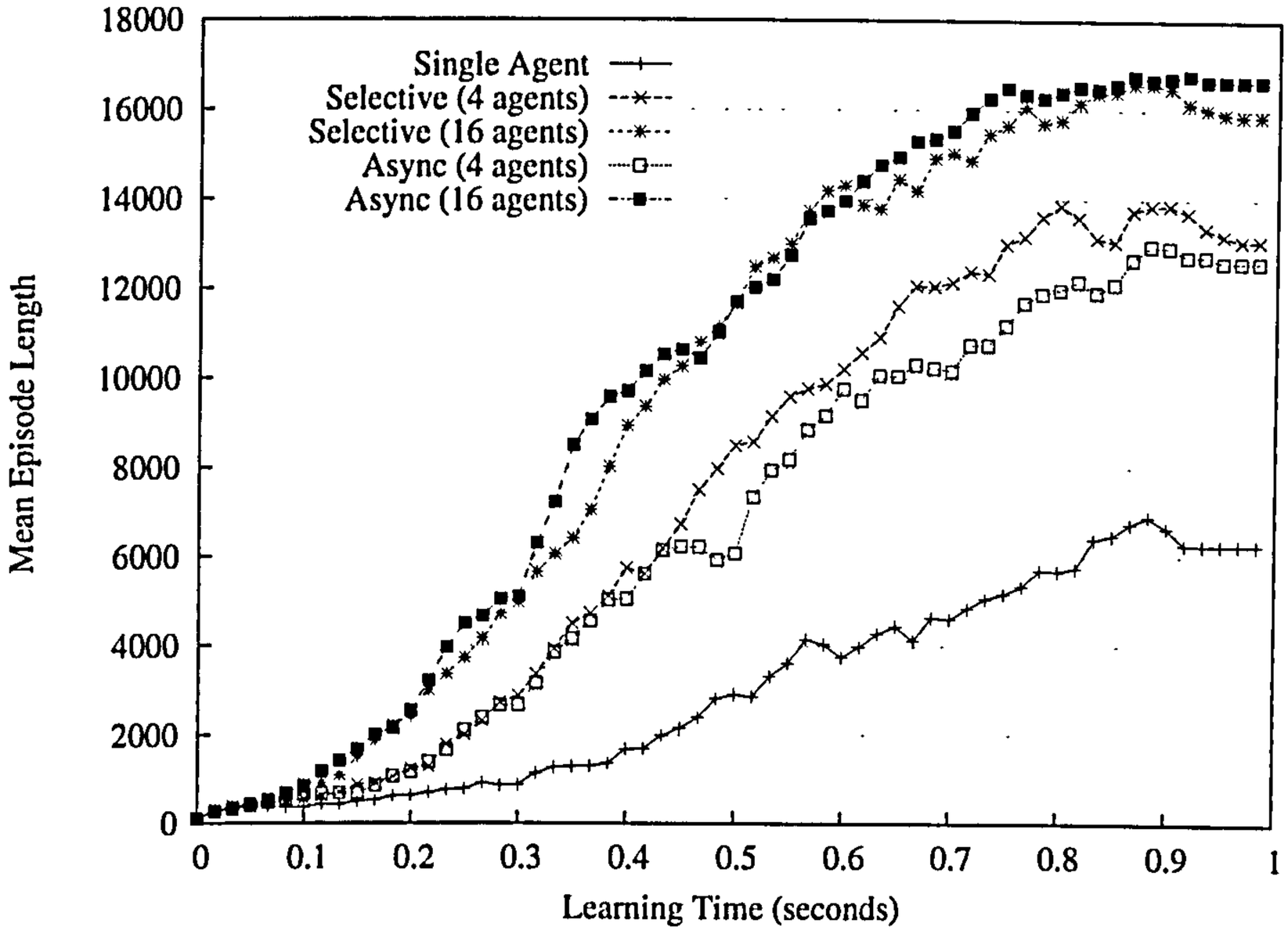


Figure 6.32: Comparing the performance of the synchronous and asynchronous selective methods in the Pole-Balancing task.

function #1 and the asynchronous selective method was used with update function #1. In the Pole-Balancing task the goal of the agents was to learn the highest quality policy in the available time of 1.0s. The use of the asynchronous approach seemed to have much less of an impact in this task. With 4 agents the asynchronous selective approach produced a slightly worse quality, and with 16 agents it produced a slightly better quality. From these results it may be observed that the asynchronous approach does not always produce an improvement in performance, and also that such an improvement is more likely when there is a fairly large number of parallel agents.

The results for the Mountain-Car task (shown in Figure 6.33) were also generated with the selective method using combination function #1 and the asynchronous selective method using update function #1. While both the selective and the asynchronous selective methods achieve good speedups in this domain, there is no significant advantage in using the asynchronous approach over the basic selective method. This is quite a different result than was achieved in domains considered above, where there has been a significant advantage in using the asynchronous approach when there are 16 agents available.

The results for the Acrobot task are shown in Figure 6.34. As with the Pole-Balancing and Mountain-Car tasks, these results were generated with the selective method using combination function #1 and the asynchronous selective method

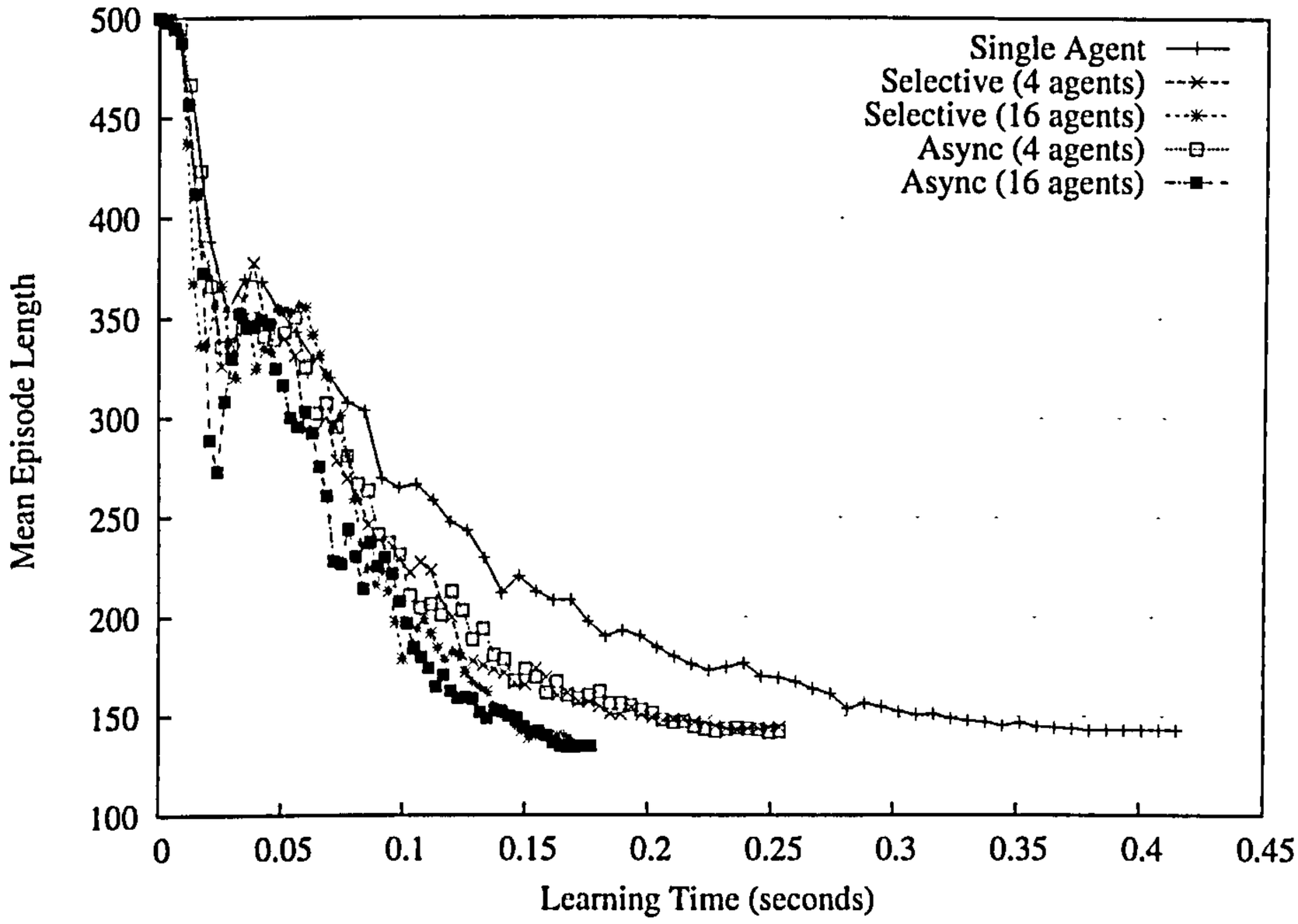


Figure 6.33: Comparing the performance of the synchronous and asynchronous selective methods in the Mountain-Car task.

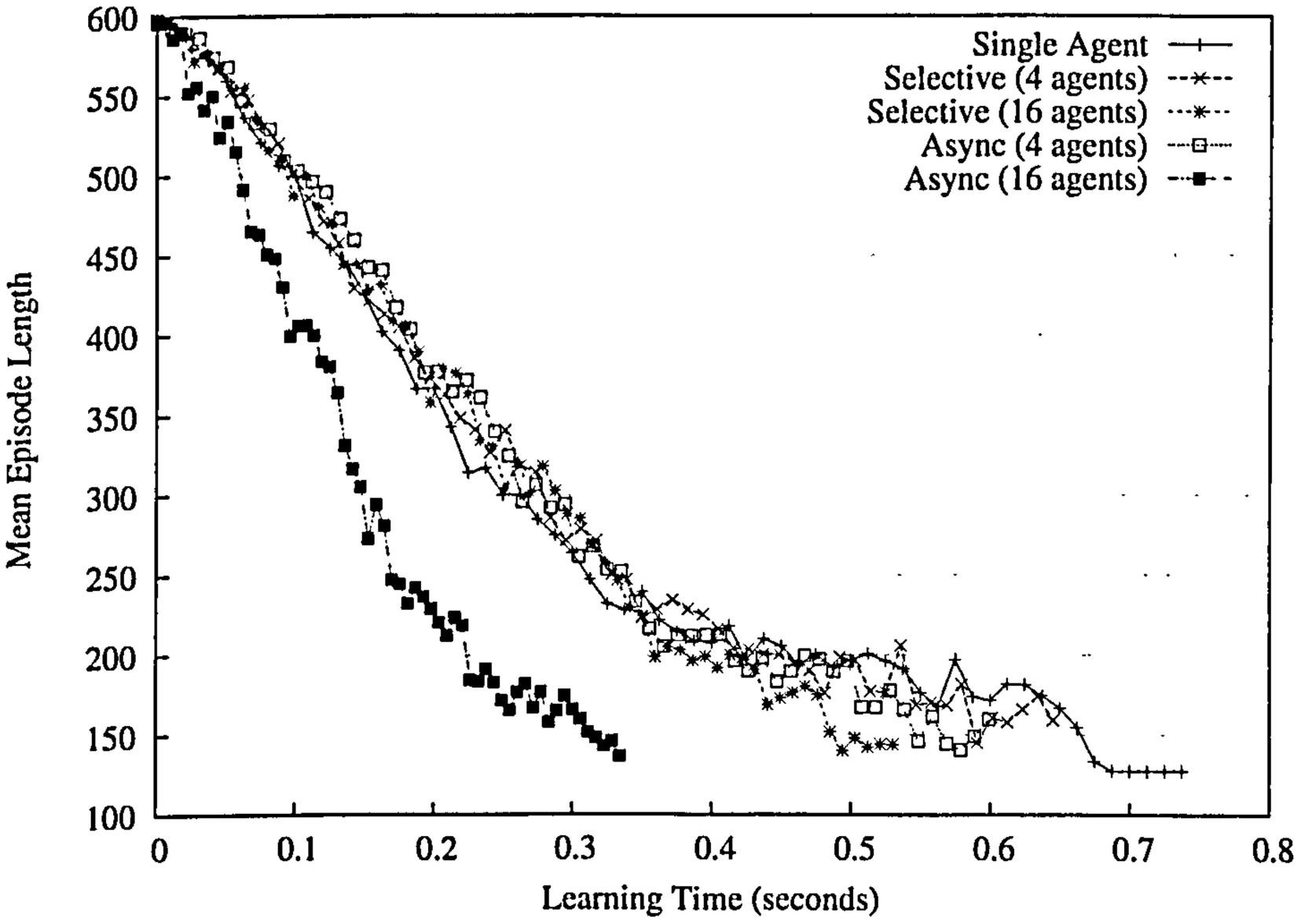


Figure 6.34: Comparing the performance of the synchronous and asynchronous selective methods in the Acrobot task.

using update function #1. These results are similar in character to those achieved in the Stochastic Grid World tasks. When there are 4 agents there is a modest performance advantage to using the asynchronous approach. With 16 agents there is a huge advantage to the asynchronous selective method, reducing by about a third the time required to converge to a high-quality policy.

The overall conclusion that can be drawn from this comparison is that there are significant benefits to adopting the asynchronous selective merge method over the (synchronous) selective merge method. While it does not always produce the best results of the two methods (such as the results from the Mountain-Car described above), its performance has only ever been worse by a small factor, and in some cases with large numbers of agents huge improvements in performance can be obtained (such as in the Acrobot and Stochastic Grid World tasks).

Determining the particular factors which make the asynchronous method so successful in some domains is an important topic for future investigation. One particular line of enquiry would be to investigate the effect of the total number f of features used by the VFA. It may be significant that the asynchronous approach performs poorly in the Mountain-Car task, where $f = 2430$, but performs well in the Acrobot task, where $f = 18432$. In both of these experiments the message size $f_{com} = 128$, although in the Acrobot experiment messages are exchanges twice as often. This suggests that when there are a large number of features, the property of the asynchronous selective method to quickly eliminate identical weight changes has a major positive impact on performance.

A limitation of the comparison given in this section is that in each experiment the asynchronous selective and the selective methods are compared with both methods using the same values for parameters p (the merge period) and f_{com} (the message size). The advantages of this approach are that both methods use up a similar network bandwidth and that comparison experiments are simple and fast to carry out. The disadvantage of this approach is that the optimum parameter choices for p and f_{com} may be *different* depending on which method we are using. Using near-optimal parameters (which are difficult to find) in each case may change the relative performance of the two methods. A comprehensive comparison of the two approaches would therefore be aided by a suitable technique for calculating suitable values for p and f_{com}.

6.5 Asynchronously exchanging absolute weight values

The asynchronous selective method, as described in Section 6.2, uses the values of the weight change vector $\Delta\vec{\theta}$ for two distinct purposes:

1. *Ranking* the weight indices in order of the potential benefit of communicating information about each weight to the group.

2. *Communicating* the weight value changes in the form of messages containing $(i, \Delta\theta_i, \theta_i)$ tuples.

The results presented in Sections 5.4 and 6.3 provide strong evidence that prioritizing weight information according to the size of each $|\Delta\theta_i|$ is an effective way to reduce the bandwidth necessary for parallel RL. However, it is not clear that the changes themselves are necessarily the best information to communicate to the other agents. In particular, adopting the asynchronous approach described in this chapter requires the use of some fairly complex mechanisms (see Section 6.2.2) for incorporating weight changes from remote agents whilst avoiding the *overshooting* problem. It is reasonable to ask at this point whether some of this complexity could be avoided by the agents exchanging only *absolute weight values* $\{\theta_i\}$ in the messages.

The asynchronous communication model makes it difficult to use an averaging approach such as that used in the visit-count average method of Chapter 4. This is because messages from the other agents arrive at different times, and each message must be processed immediately. This means that local data structures must be updated using only *one* remote agent's weight value. While it would be possible to average these values over time by caching recently received values (using a *sliding-window* for example), it is likely that such an approach would slow convergence in the same manner as the mean-merge method (see Section 4.4.3).

Given two estimates of a weight value, one local and one from a remote agent, we could consider taking the mean of these two estimates. However, bear in mind that with the selective approach a remote weight value is only likely to be received if a large change in the weight is observed by the remote agent. To reduce the VFA error quickly in the early stages of a parallel run, it is vital that large weight changes are quickly propagated to all the agents by prioritizing the remote agent's estimate. This leads to the definition of a relatively simple asynchronous algorithm (defined in Algorithm 12) based on messages containing (i, θ_i) tuples (i.e. absolute weight values). When a remote weight value is received, the local weight is overwritten by the remote value. In addition, the local weight change is reset to zero. Note that this means that some locally learned information is lost when updates occur in response to arriving messages. I will refer to this algorithm as *Abs-Async* (an asynchronous method based on exchanging absolute weight values.)

Using absolute weight values instead of weight changes essentially eliminates the problem of overshooting. The main disadvantage of the approach is that if

Algorithm 12 Agent pseudocode for the *Abs-Async* method. Messages sent by the agent contain only the absolute weight values, not the weight changes.

{Initialization}
for all i **do**
$\theta_i \leftarrow \theta_{init}$
$\theta_i^{ref} \leftarrow \theta_{init}$
end for

{Main loop}
while time elapsed $< t_{end}$ **do**

{Learning quantum}
for $step = 1$ to q **do**
Execute a simulation step and update weight vector $\vec{\theta}$.
end for

{Scheduled Broadcasts}
if scheduled broadcast is due **then**
Calculate $\Delta\vec{\theta} = \vec{\theta} - \vec{\theta}^{ref}$.
Rank each index i according to the value of $|\Delta\theta_i|$.
$best \leftarrow$ { the f_{com} highest ranked indices }
$m \leftarrow \{(i, \theta_i) \mid i \in best\}$
Send message m to all other agents.
end if

{Message Receive}
for each new incoming message m **do**
for all $(i, \theta_i') \in m$ **do**
{Assign θ_i' to both θ_i and θ_i^{ref}}
$\theta_i \leftarrow \theta_i'$
$\theta_i^{ref} \leftarrow \theta_i'$
end for
end for

end while

the number of weights per message (f_{com}) is fairly large, the loss of local weight changes during updates could have a negative impact on the rate of convergence.

Experimental Details

To evaluate the potential of the *Abs-Async* method, its performance was compared to that of the best performing method developed so far: the asynchronous selective merge method using update function #1 and a staggered broadcast schedule. This method is labelled *Async* in the graphs which follow.

The evaluation in this section was carried out at a later date than the other experimental work in this thesis. The particular Beowulf cluster used for previous experiments was unavailable at this time, and so the results reported in this section were generated using a second, more powerful cluster (details below). Unfortunately this means that the learning curves in this section cannot be directly compared with those in previous sections, since the underlying system properties are different. On the other hand, collecting data using a second parallel computing system will provide some indication of how the parallel methods developed in this thesis will perform on different parallel hardware.

The cluster used in these experiments consisted of 24 nodes, where each node was a machine based on two AMD Opteron 275 dual core 2.2GHz processors. Since there are 4 cores per node, up to four parallel RL agents can run on each node. Each node contained 8GB of registered DDR memory on a bus with 6.4GB/s throughput. The nodes were connected using a high performance InfiniBand interconnect with 10Gb/s bandwidth and $2us$ latency.

The values used for parameters p and f_{com} in these experiments are different from those used in earlier evaluations. This is so that the best possible performance can be obtained given the particular system properties of this new cluster. In this context, the *relative* performance of the two methods considered here can be assessed as the potential speedup is pushed to the limit.

Stochastic Grid World (low-difficulty)

The graph in Figure 6.35 shows the performance of the two methods in the low-difficulty Stochastic Grid World task. The experimental settings used were as follows: the merge period p was set to 1000 steps, and the message size f_{com} was set to 128. This meant that the overall network bandwidth used was 5 times as much as in the early evaluation on the old cluster. Reward function #2 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. RL parameters α and ϵ decayed linearly during each run, according

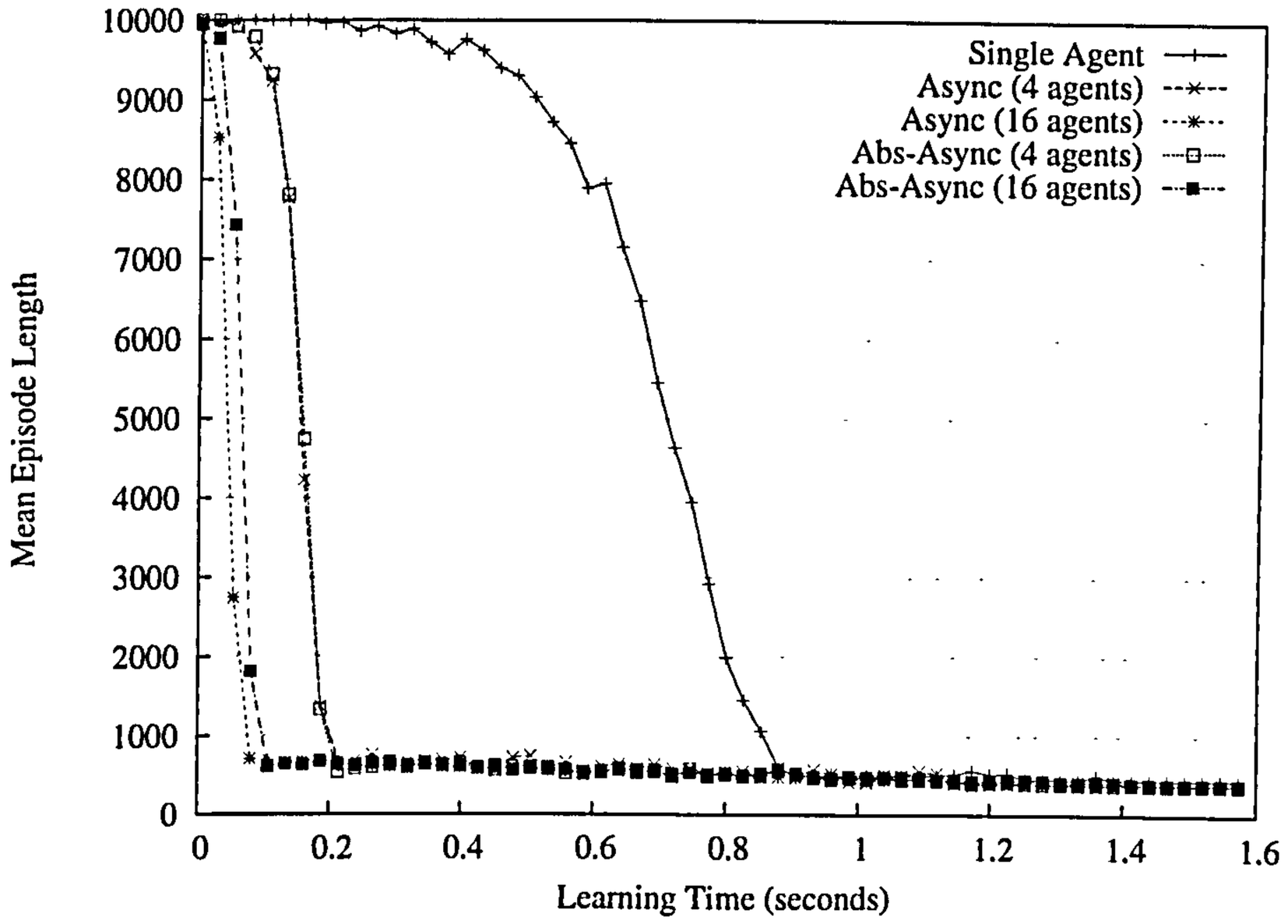


Figure 6.35: Using the low-difficulty Stochastic Grid World task to compare the asynchronous selective method with an alternative method based on absolute weight values.

to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 1 \times 10^{-8}$. Learning curves were plotted for 4 and 16 agents using the *Async* and *Abs-Async* methods. A single agent learning curve was also included for comparison.

The results in Figure 6.35 show that the performance of the two parallel RL methods is identical when 4 agents are used. With 16 agents the *Async* method produces a small improvement in performance compared to the *Abs-Async* method. Note also that using the new (more powerful) cluster means that it is possible to get much closer to achieving a *linear* speedup using either of these methods (compare Figure 6.35 with Figure 6.30 on page 223).

Stochastic Grid World (high-difficulty)

Figure 6.36 shows learning curves for both of the methods in the high-difficulty Stochastic Grid World task. In this experiment, algorithm parameters $p = 2500$ and $f_{com} = 512$, using 20 times the network bandwidth compared to the old cluster. Reward function #1 was used, and results were averaged over 10 runs. Episodes were terminated if they reached 10,000 steps. RL parameters α and ϵ decayed linearly during each run, according to the parameters $\alpha_0 = 0.2$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 1.0$, $\lambda = 0.95$ and $\theta_{init} = 0$.

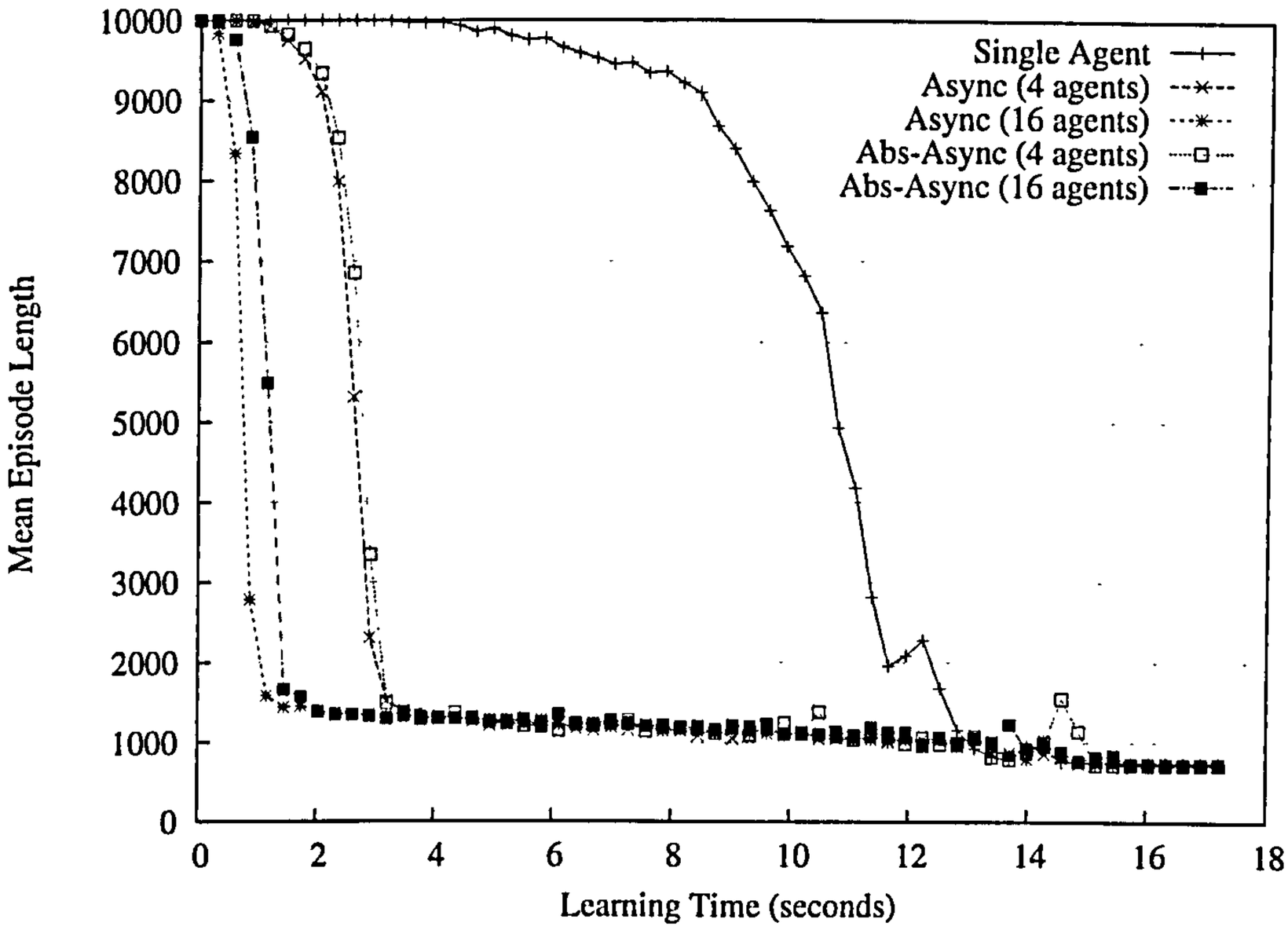


Figure 6.36: Using the high-difficulty Stochastic Grid World task to compare the asynchronous selective method with an alternative method based on absolute weight values.

The results in Figure 6.36 follow a similar pattern to those observed with the low-difficulty grid world in Figure 6.35. With 4 agents the performance of the *Async* and *Abs-Async* methods are practically identical. Increasing the number of agents to 16 means that the *Async* method converges slightly more quickly to an accurate value function approximation.

Pole Balancing

The results for the Pole Balancing task are shown in Figure 6.37. Parameter values $p = 200$ and $f_{com} = 64$ were used, which meant that 5 times the network bandwidth was used compared to experiments on the old cluster. Reward function #1 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 20,000 steps. RL parameters α and ϵ decayed linearly during each run, according to the parameters $\alpha_0 = 0.25$, $\epsilon_0 = 0.2$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$, $\lambda = 0.5$ and $\theta_{init} = 0$.

In contrast to the experiments using the Stochastic Grid World tasks, the 4 agents using the *Async* method clearly outperform the 4 agents using the *Abs-Async* method, producing (on average) a final policy of higher quality in the fixed learning time of $0.25s$. With 16 agents the results are less conclusive. The *Async* and *Abs-Async* methods produce final policies of very similar quality. However,

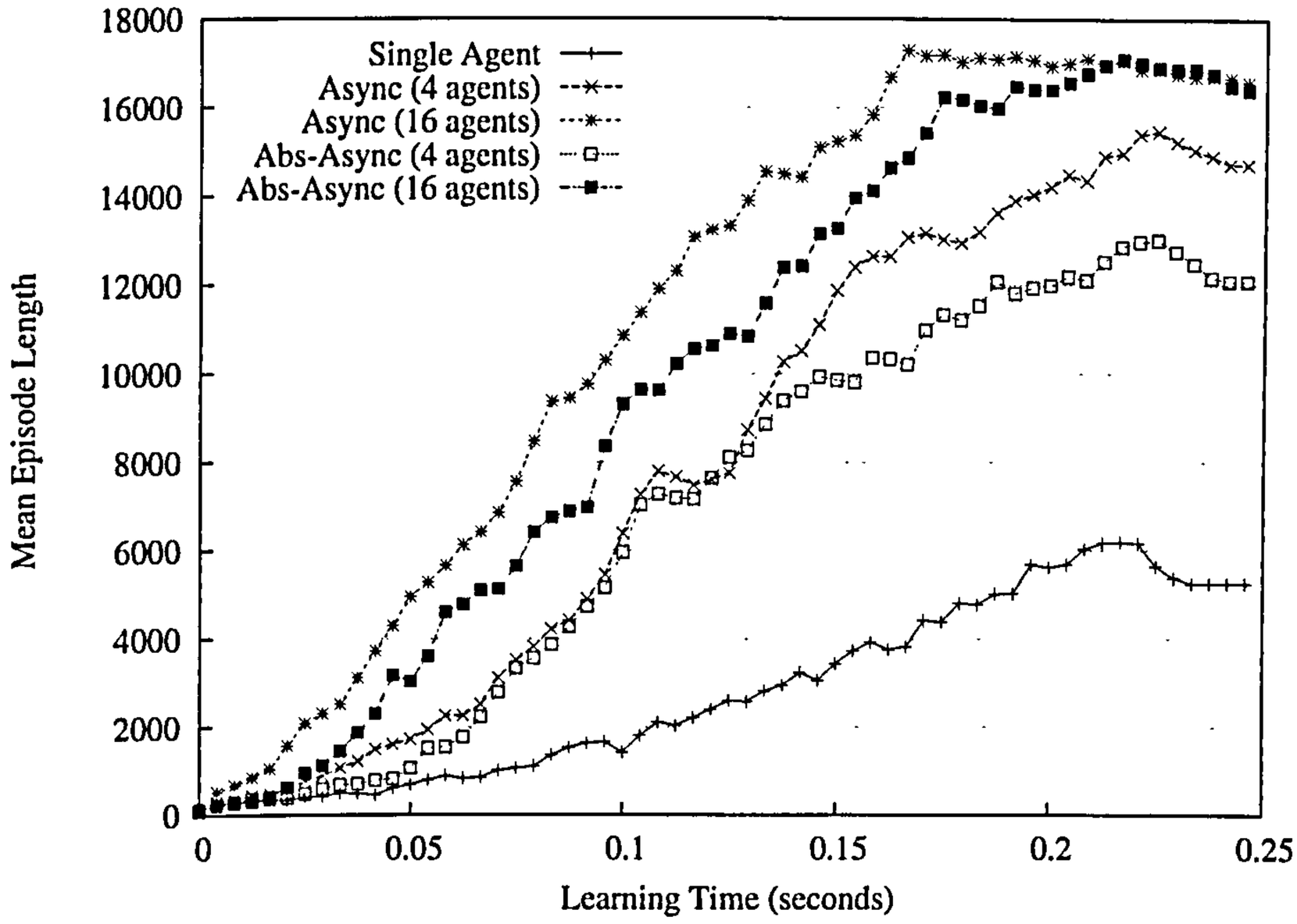


Figure 6.37: Using the Pole Balancing task to compare the asynchronous selective method with an alternative method based on absolute weight values.

during the first 0.15s of the learning time the *Async* method appears to improve policy quality at a faster rate, reaching a plateau at a mean episode length of 17000. There may be a side effect of update function #1 in this setting which degrades policy improvement in the later stages of learning.

Mountain Car

Results for the Mountain Car task are shown in Figure 6.38. Parameter values $p = 100$ and $f_{com} = 128$ were used, which meant that 20 times the network bandwidth was used compared to experiments on the old cluster. Reward function #2 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 500 steps. RL parameters α and ϵ decayed linearly during each run, according to the parameters $\alpha_0 = 0.5$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 0.99$, $\lambda = 0.9$ and $\theta_{init} = 0.0001$. Binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average episode length below 145 over the set of 100 runs.

This set of results provides the clearest demonstration yet that in some circumstances the *Abs-Async* method will perform significantly worse than the asynchronous methods based on exchanging changes in the weights. With 4 agents the *Async* method is a small amount faster. With 16 agents a policy of the same quality can be found by *Async* in about two-thirds of the time required by *Abs-Async*.

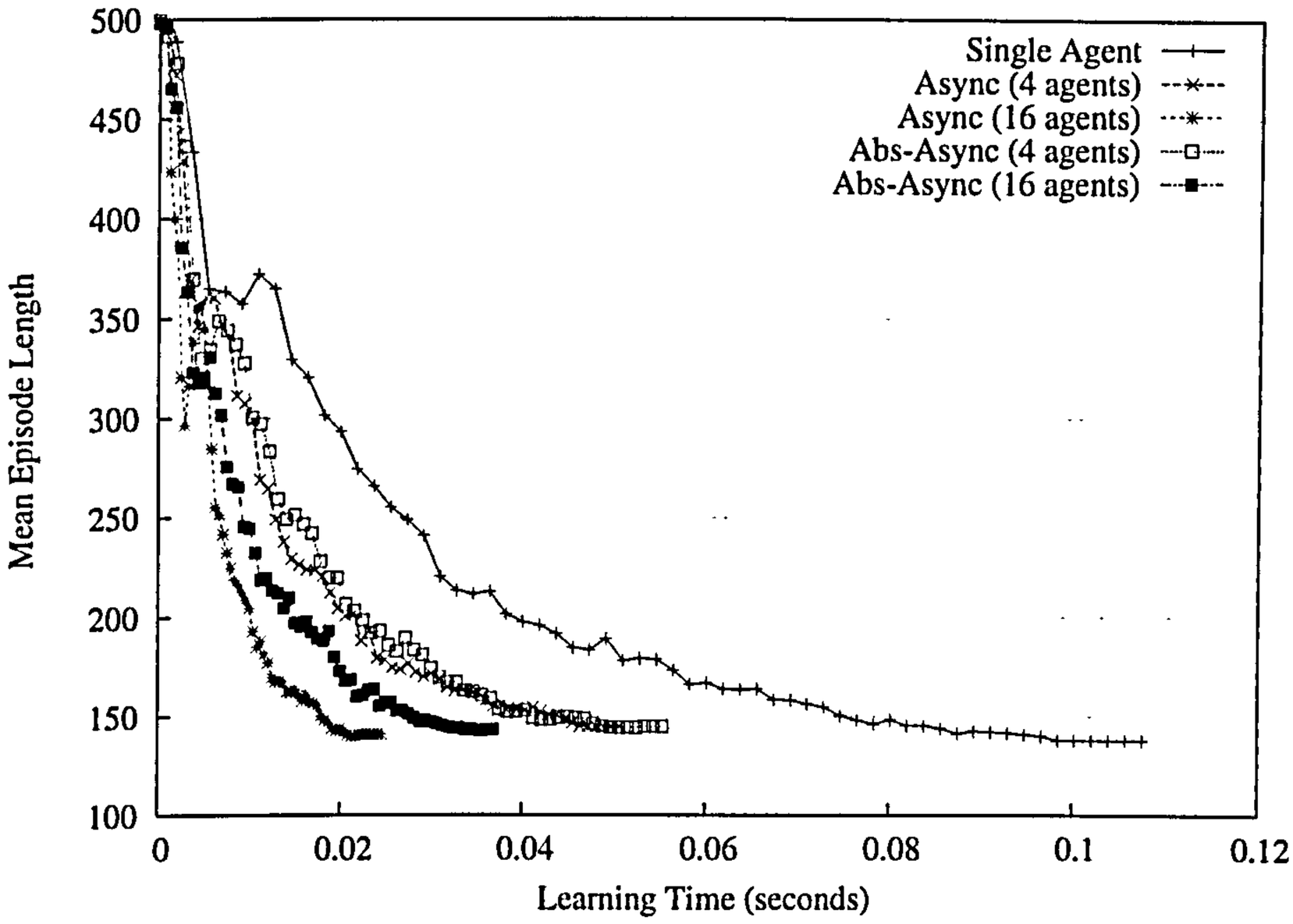


Figure 6.38: Using the Mountain Car task to compare the asynchronous selective method with an alternative method based on absolute weight values.

Acrobot

Results for the Acrobot task are shown in Figure 6.39. Parameter values $p = 100$ and $f_{com} = 256$ were used, which meant that 20 times the network bandwidth was used compared to experiments on the old cluster. Reward function #1 was used, and results were averaged over 100 runs. Episodes were terminated if they reached 600 steps. RL parameters α and ϵ decayed linearly during each run, according to the parameters $\alpha_0 = 0.1$, $\epsilon_0 = 0.1$ and $t_{lim} = 0.9$. The remaining RL parameters were $\gamma = 1.0$, $\lambda = 0.9$ and $\theta_{init} = 0$. Binary search was used to determine for each group of agents the shortest interval of real-time required to achieve an average episode length below 140 over the set of 100 runs.

These results show an even greater difference in performance between *Async* and *Abs-Async*. Using 4 agents the *Async* method converges about 25% more quickly than the *Abs-Async* method. With 16 agents the difference is more pronounced, with the *Async* method converging in under half the time required the *Abs-Async* method. There is something about the character of this particular domain which seems to heavily penalize methods which are not particularly efficient in their use of network bandwidth (consider both Figure 6.39 and Figure 6.34 on page 226). This may be related to the fact that while a large number of features are used (18432), the generalization of the approximator is very good, keeping the

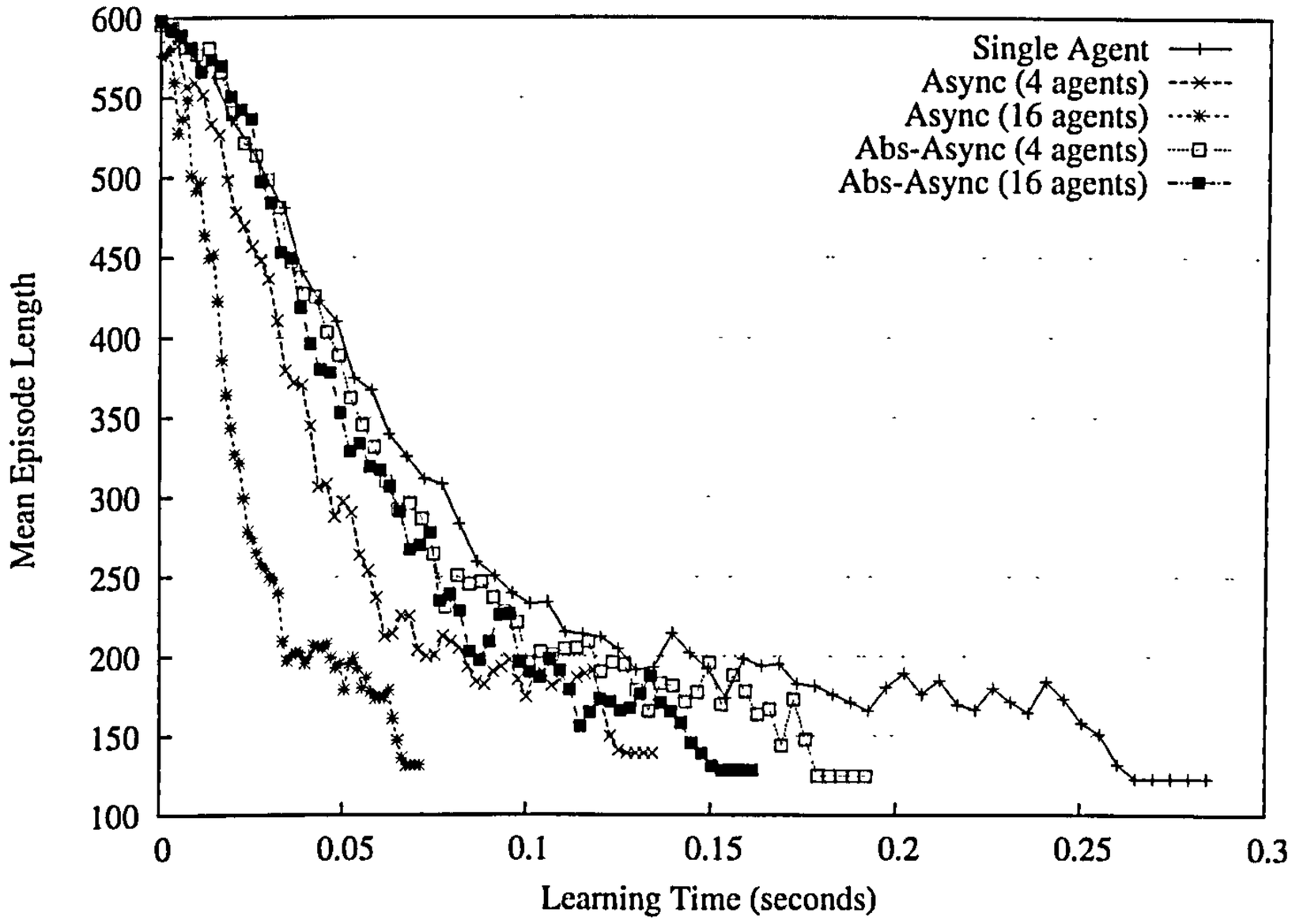


Figure 6.39: Using the Acrobot task to compare the asynchronous selective method with an alternative method based on absolute weight values.

required learning time small and making parallelization particularly difficult.

Summary of Comparison

The empirical results presented in this section examine the performance of a new method, *Abs-Async*, in which agents asynchronously exchange absolute weight values. The results in this section were generated using a different (more powerful) Beowulf cluster, and so the graphs should not be compared directly to other graphs in the thesis. The performance of *Abs-Async* was compared to that of the best performing method developed so far in this chapter, the asynchronous merge method using update function #1 (abbreviated here as the *Async* method).

The comparison showed that in some cases the two methods perform very similarly, but in other cases *Abs-Async* is greatly outperformed by the *Async* method. Two major factors were identified which affect the degree to which *Abs-Async* performs worse than *Async*. The first factor is the number of agents involved. The more agents that are used, the more likely it is that *Async* will produce the better performance. The second factor is the character of the particular learning task at hand. In some learning tasks (e.g. the Stochastic Grid World tasks) the two methods perform very similarly. In other tasks (e.g. the Mountain Car and Acrobot tasks) there is a very large performance advantage in using the *Async* method.

The most likely reason for the poor performance of *Abs-Async* in certain sit-

uations is the way that locally-learned weight changes can be lost when updates occur in response to messages arriving. The fact that messages only contain weights which have undergone large recent changes does mean that most of the time no significant information is lost. However, if a large number of agents are involved, or if there are a large number of approximator weights, the likelihood of losing important information in this way increases, and *Abs-Async* performs poorly.

The poor performance of this method provides some evidence that parallel RL methods based on exchanging *weight changes* (i.e. most of the methods described in Chapters 5 and 6) are more likely to be successful than methods based on exchanging *absolute weight values.* Monitoring recent changes in the VFA makes it easier to combine information from several agents without destroying locally-learned information which has yet to be communicated. Methods based on weight changes do lead to the *overshooting* problem, adding to the complexity of some of the methods. However, based on the evidence presented here, a method based on weight changes which trades off accelerated convergence against the risk of overshooting seems to produce the best performance.

6.6 Summary and Conclusions

The following material has been presented in this chapter:

- Motivation for the use of *asynchronous message passing* to improve the performance of the selective merge method which was described in Chapter 5.
- The general form of the *asynchronous merge method.* This method uses the same mechanism (of ranking weights by the magnitude of the recent accumulated change) for message construction as the selective merge method, but removes the need for any synchronization step.
- Several candidate procedures (known as *update functions*) for updating the local VFA in response to incoming messages from other agents.
- A number of mechanisms to *schedule* the message broadcasts of individual agents without requiring explicit synchronization. The purpose of these mechanisms is to keep the times at which agents send messages distributed fairly evenly over time.
- An evaluation of the proposed update functions, where the performance with each update was tested with different numbers of agents in each of the evaluation domains defined in Section 4.3.1. The reported results were generated using the implementation on the cluster of workstations.

- A comparison of the performances of the *asynchronous merge method* and the (synchronous) *selective merge method* in each of the evaluation domains. This comparison was also based on results collected on the cluster of workstations.

- A comparison of the performances of the asynchronous merge method and an *alternative* asynchronous method which uses messages containing only *absolute weight values*, not the changes in the weights. This comparison was based on results collected at a later date using a different, more powerful cluster of workstations.

From this material we can draw the following conclusions:

- The *asynchronous merge method* can produce parallel speedups that are better than those of the selective merge method of Chapter 5 by allowing the agents to distribute their broadcasts more evenly over time, and to update their local VFAs as and when messages arrive.

- On the cluster of workstations, the asynchronous merge method consistently outperforms both the visit-count merge method of Chapter 4 and the selective merge method of Chapter 5. In particular, the relatively small speedups obtained by the selective merge method in the Acrobot task (which has a large number of approximator weights but a small single-agent learning time) were greatly improved upon. Significant increases in the parallel speedup were obtained in all of the evaluation domains tested.

- The *staggered* schedule for determining when agents broadcast their messages was found to be the most robust mechanism of those proposed, outperforming the uniform and exponential schedules in a set of preliminary experiments.

- Of the three update functions evaluated, update function #1 was shown to produce the greatest speedups in almost all situations. However using update function #1 when there is a high probability of two or more agents transmitting messages simultaneously may have an impact on performance, or prevent convergence in extreme cases.

- The asynchronous merge method inherits the parameters p and f_{com} from the selective merge method. As it was demonstrated in Chapter 5, selecting appropriate values for these parameters is vital for achieving good performance using either of these methods. Since there is not currently an analytic or heuristic method for determining the optimum values of p and f_{com}, it is necessary to use some degree of trial and error to select these parameters.

- Parallel RL methods where agents communicate changes to the VFA weights are very effective in practice, allowing a trade-off to be made between an increased convergence rate and the risk of *overshooting*. While alternative methods based on the exchange of *absolute weight values* effectively eliminate the overshooting problem, the empirical evaluation of Section 6.5 suggests that such methods will result in degraded performance in many situations.

Chapter 7

Combining RL with Symbolic Planning

This chapter moves on from the topic of parallelization investigated in the preceding chapters, and proceeds to examine how *symbolic planning* can be used to constrain and accelerate learning in an RL problem. A hybrid method called PLANQ-learning is presented which integrates a planner based on the STRIPS representation with the well-known Q-learning algorithm. A high-level plan that achieves the goal of the Q-learner is computed and is then used to guide the learning process. This is achieved by shaping the reward function based on the preconditions and effects of the abstract plan operators. Using this approach allows a high quality policy to be learned more quickly.

This chapter begins with a brief overview of planning using the STRIPS representation, followed by a description of the PLANQ learning method itself. A problem domain is then defined which will be used to evaluate the performance of PLANQ in problems of increasing difficulty. A comparison in this domain of the performance of a PLANQ-learner and a Q-learner produces encouraging results, but a much greater improvement in performance is achieved by incorporating a *state-abstraction* mechanism. The performance of this extended PLANQ-learner is compared with that of an agent using HSMQ-learning (Dietterich, 2000a), a hierarchical RL algorithm that is able to exploit the same state abstraction. The results show that PLANQ is superior in its scaling-up properties both in terms of environment time steps and in terms of computation time. It is also shown that PLANQ exhibits high variance in the computation time expended per time step.

7.1 The STRIPS Planning Representation

The STRIPS *representation* (Fikes and Nilsson, 1971) and its descendants form the basis of many symbolic planning systems. It is based on first-order predicate logic, but has a number of restrictions which make it possible to search for plans without requiring a full theorem proving system. Despite these restrictions, STRIPS is sufficiently expressive to represent many interesting and difficult planning problems. Each individual state is represented by a set of *positive ground literals.* The set of goal states is described by a *conjunction of positive literals.* Each STRIPS operator is represented by three components:

Preconditions The literals which must be true in a state for the operator to be applicable in that state.

Add List The literals which become true in the state which results from applying the operator.

Delete List The literals which become false in the state which results from applying the operator.

Two different planners based on the STRIPS representation were used during the course of the work reported in this chapter. Both of these planners were based on the influential GRAPHPLAN algorithm. GRAPHPLAN itself is based on a data structure called a *planning graph*, a graph structure annotated with STRIPS literals and operators. The planning graph encodes which literals can be made true after n operators have been applied, and which literals are *mutually exclusive* at that time step. The algorithm works by constructing the planning graph *forwards* from the initial state (time step 0) until a time step is reached where all the goal literals are true and not mutually exclusive. Then the planning graph is searched *backwards* for a valid plan. If no plan can be found the planning graph is extended by one time step and the backward search is repeated. GRAPHPLAN constructs plans very quickly in domains which do not have a large number of objects.

The initial experiments reported in this chapter used the FASTFORWARD or FF planner (Hoffmann, 2000), which uses the GRAPHPLAN algorithm on a relaxed version of the planning problem. The planning graph then forms the basis of a heuristic for forward search. In later experiments a custom implementation of the GRAPHPLAN algorithm was used, which eliminated expensive parsing and file access operations in order to minimize the time required to generate a plan.

7.2 The PlanQ Learning Method

In this section a novel method for hierarchical learning in large-scale problems is presented. This hybrid method uses symbolic plans to explicitly represent prior knowledge of the internal structure of a (goal-oriented) MDP. By exploiting this knowledge the number of steps in the environment required to learn an adequate policy can be greatly reduced. I will call this approach PLANQ-learning.

The definition of an *adequate* policy will vary according to the application domain. The use of the word "adequate" emphasizes that the primary goal here is not to find a truly optimal policy, but to find an acceptable policy in a reasonable amount of time. The truly optimal policy may not be obtainable for a number of reasons:

- The available knowledge of the problem structure may be *incomplete*, and learning can only be accelerated by this partial knowledge.
- The available knowledge of the problem structure may be *inaccurate*, arising either from an error in the design of the knowledge base, or as a result of sacrificing some accuracy to obtain a simpler abstract model of the problem.
- Limited availability of experience in the environment and/or limited computational resources.

The approach explored in this chapter uses a STRIPS knowledge base and planner to define the desired high-level behaviour of an agent, and reinforcement learning to learn the unspecified low-level behaviour. This can be viewed as an instance of a *layered architecture* (Gat, 1997; Stone, 1998), with high-level planning and low-level learning. One low-level behaviour must be learned for each STRIPS operator in the knowledge base. There is no need to separately specify a reward function for each of these operators - instead a reward function is derived *directly* from the logical *preconditions and effects* of each STRIPS operator.

As well as a knowledge base describing the high-level operators to be learned, the agent has access to an interface which, given a low-level reinforcement learning state (representing low-level percepts), can construct a high-level set of STRIPS literals which describe the state. The STRIPS output of the interface must include the *current goal* of the agent. This limits the learning agent to domains where the only reward received is associated with reaching one of a set of goal states.

Initially the agent has no plan, so it uses the above interface to turn the initial state into a STRIPS problem description. The STRIPS planner takes this problem description and returns a sequence of operators which solves the problem. The

agent has a subordinate Q-learning agent to learn each operator, so the Q-learner corresponding to the first operator in the plan is activated.

The activated Q-learner takes responsibility for choosing actions, while the primary agent monitors the high-level descriptions of subsequent states. When the high level description changes the primary agent performs one or more of the following operations:

Goal Changed If the overall goal of the agent is detected to have changed a new plan is needed, so the agent must run the STRIPS planner again.

Effects Satisfied If the changes specified by the Add and Delete Lists of the operator have taken place then the Q-learner has been successful and receives a reward of +1. The Q-learner for the next operator in the plan is then activated.

Preconditions Violated If a precondition becomes false while the effects are still unsatisfied then the operator is assumed to have failed. The Q-learner receives a reward of -1 and the STRIPS planner is activated for re-planning.

Operator In Progress If the effects are unsatisfied and the preconditions inviolate, either the effects are partially complete, or a irrelevant literal has changed truth value. The current Q-learner receives reward 0 and continues.

The assumption that violating the preconditions indicates operator failure has some important consequences. It means that any operator which deletes some of its own preconditions must perform all these deletions together in the *final action* of the low-level behaviour. To relax this restriction the STRIPS representation could be extended by specifying an *invariant* for each operator. The invariant is a logical formula which must remain true during the lifetime of an operator. If the invariant ever becomes false an operator failure is deemed to have occurred. The experiments reported in this chapter did not require invariants, but they may be needed for other application domains.

7.3 Evaluation Domain

The evaluation domain used here is a grid world which consists of both smaller *grid squares* and larger *region squares* which contain groups of grid squares. The region squares represent target areas to which the mobile robot must navigate. Using region squares for the agent's goal rather than individual grid squares means that it will be possible to reason about goals at an abstract level which only considers the region squares, ignoring the detail provided by individual grid squares. There

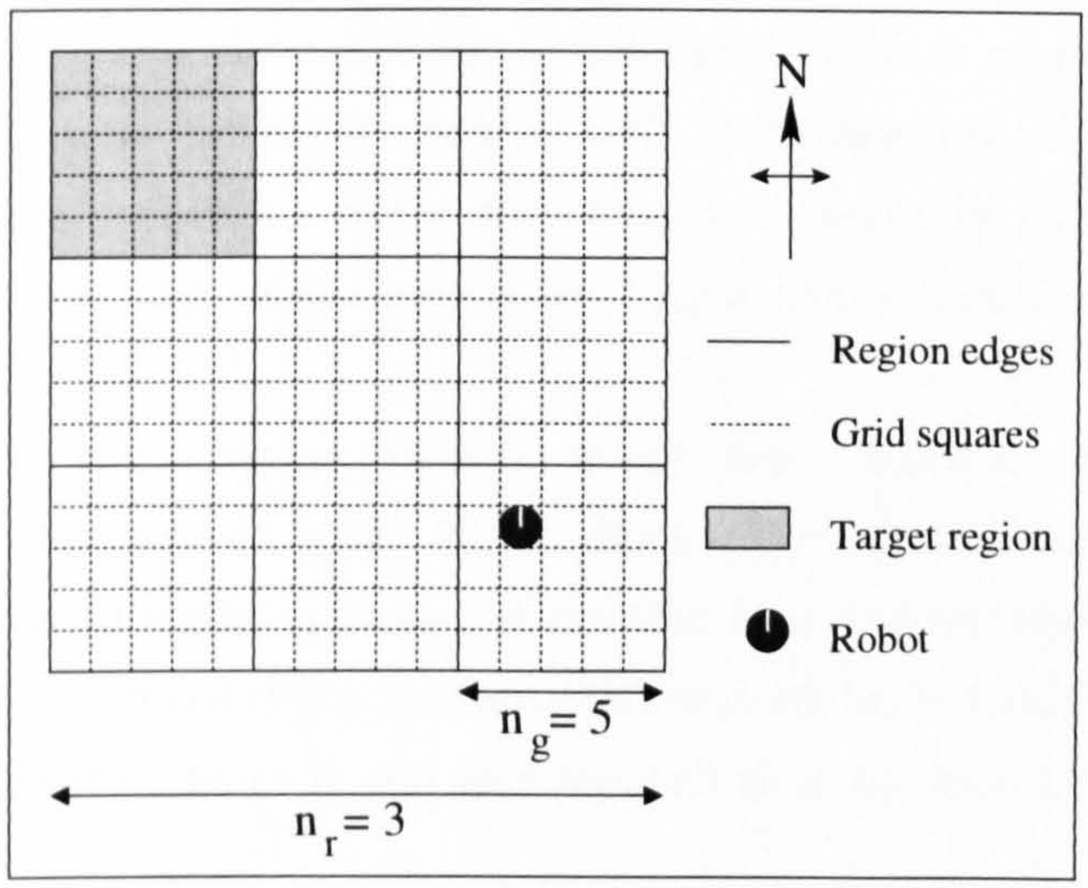


Figure 7.1: An instance of the evaluation domain.

is only one *active* (i.e. goal) region square at any time. Whenever the robot enters the active region it receives a reward and a new goal region is chosen at random. The robot is situated in one of the grid squares, and faces in one of the four compass directions: *north*, *east*, *south* and *west*.

To evaluate performance as the state space is scaled-up, a class of these problems was defined, where the regions are arranged in a square of side n_r (see Figure 7.1). Each region contains a square set of grid squares, of side n_g. There are a total of $n_g^2 n_r^2$ grid squares which the robot can occupy. Since the destination of the robot must be a region square, there are n_r^2 possible destinations. Hence the size of the state space S, which encodes the position and direction of the robot, as well as the location of the current destination region, is:

$$|S| = 4n_g^2 n_r^4$$

This domain is intended to be representative of other goal-oriented domains which have a significant degree of *high level structure*. In addition, the simplicity of this domain makes it an ideal choice for illustration purposes. It is easy to partition the overall problem into parts where low-level learning is to be used (within individual region squares) and parts where high level planning is to be used (navigating between the region squares using high level movement operators). Allowing n_g and n_r to be varied independently means that not only can we scale up to larger problems in a quantitative fashion, but we can also control the relative difficulty of the parts of the problem assigned to the planner and the low-level learners.

There are only three actions available to the robot: *turn left*, *turn right* and *forward*. *Turn left* turns the robot 90° anticlockwise to face a new compass direction. *Turn right* causes the robot to make a 90° clockwise turn. *Forward* will move the robot one square forward in the direction it is currently facing. If the robot tries to move off the edge of the map of grid squares the *forward* action will have no effect.

The robot receives a reward of 0 on every step, except on a step where the robot moves into the active region. When this happens the robot receives a reward of 1 and a new active region is picked at random from the remaining regions. This introduces a small element of stochasticity to the domain, but this is not significant for the PLANQ-learner, since it will re-plan each time the goal (the active region) changes.

The high level STRIPS representation of the evaluation domain abstracts away the state variables corresponding to the orientation of the robot and the position of the grid square it occupies in the current region. Reasoning with this representation is limited to the level of regions. It allows a path to be planned between the current and target regions using a knowledge base which encodes an adjacency relation over the set of regions.

Each region at a position (x, y) is represented as a constant `r_x_y`. The predicate `adj(r1,r2,dir)` encodes the fact that region `r2` can be reached from region `r1` by travelling in the direction `dir`, which can be one of the compass points `N`,`S`,`E` or `W`. The `at(r)` predicate is used to encode the current location of the robot, and to define the goal region to be reached.

The operators available are `NORTH`, `SOUTH`, `EAST` and `WEST`, which correspond to low-level behaviours to be learned for moving in each of the four compass directions.

PDDL, the *Planning Domain Definition Language* (Ghallab et al., 1998), is used to pass problem descriptions and solutions between the agent and the FF planner. It forms the basis of a weakly-coupled interface between the agent and the planner, which allows any other external planner to be used if it can manipulate PDDL data. Examples of an operator definition and a problem description (from the evaluation domain) in PDDL format are shown in Figures 7.2 and 7.3.

7.4 Experiment 1: Results

In the first experiment the PLANQ-learner was evaluated using a variety of values for n_r and n_g. For the purpose of comparison a standard Q-learning agent and an agent using a hand-coded version of the optimal policy were also evaluated in the domain.

```
(:action NORTH :parameters (?from ?to)

 :precondition (and (at ?from)
                    (adj ?from ?to N))

 :effect (and  (at ?to)
               (not (at ?from)))
)
```

Figure 7.2: The operator NORTH from the evaluation domain.

```
(define (problem regiongrid1)
(:domain regiongriddomain)

(:objects  r_0_0
           r_0_1
           r_1_0
           r_1_1
)
(:init  (adj r_0_0 r_1_0 E)
        (adj r_0_0 r_0_1 S)
        (adj r_0_1 r_1_1 E)
        (adj r_0_1 r_0_0 N)
        (adj r_1_0 r_0_0 W)
        (adj r_1_0 r_1_1 S)
        (adj r_1_1 r_0_1 W)
        (adj r_1_1 r_1_0 N)
        (at r_0_1)
)
(:goal
        (at r_0_0)
))
```

Figure 7.3: A PDDL problem description for $n_r = 2$.

The standard Q-learner uses the full state space S as defined above, and chooses between the three low-level actions: *turn left*, *turn right* and *forward*. Like the PLANQ-learner, it receives a reward of 1 on a step where it enters a goal region, and a reward of 0 everywhere else. In all of these experiments the learning rate α is 0.1 and the discount factor γ is 0.9. An ϵ-*greedy* exploration strategy is used (see Section 3.3), with the ϵ parameter decaying linearly from 1.0 to 0.0 over the course of the experiment. The rate of decay for ϵ was chosen so that the decay was as fast as possible without impacting on the final quality of the solution.

Examples of the performance of the agents over time are shown in Figures 7.4 and 7.5. The graphs in these Figures demonstrate that as larger values of n_r are considered the performance advantage of PLANQ-learning over Q-learning becomes progressively smaller. The Q-learning agent consistently learns the true optimal policy. The PLANQ-learner learns a good policy, but not quite the optimum. This is because the planning model of the grid world does not model the cost of making turns - the plans {`NORTH, EAST, NORTH, EAST`} and {`NORTH, NORTH, EAST, EAST`} are considered equally suitable by the planner, but in reality the latter plan has a better reward rate. This results in slightly sub-optimal performance.

In both of the experiments the PLANQ-learner finds a good policy several times more quickly than the Q-learner. This is to be expected: the Q-learner must learn both high and low-level behaviours, whereas the PLANQ-learner need only learn the low-level behaviour. However it can be observed that the advantage of the PLANQ-learner over the Q-learner is less in the $n_r = 5$ experiment than in the $n_r = 3$ experiment. The general trend for the PLANQ-learner to lose advantage as n_r increases is discussed in the next section.

7.5 Problems with Experiment 1

The learning speed-up achieved by the PLANQ-learner over the Q-learner can be attributed to the *temporal abstraction* inherent in the STRIPS formulation of the problem domain. The temporal abstraction allows us to express the overall problem as a number of *sequential sub-problems*, each of which is easier to learn than the overall task. Because the PLANQ-learner can learn the sub-tasks separately, it can finish learning more quickly than the Q-learner, which must tackle the problem as a whole.

However, the advantage offered by temporal abstraction grows smaller as larger domains are considered since there is no *state-abstraction* available to the PLANQ-learner. A state abstraction allows state variables to be excluded from the learner's state space if they are not relevant to learning a particular task (or subtask). For

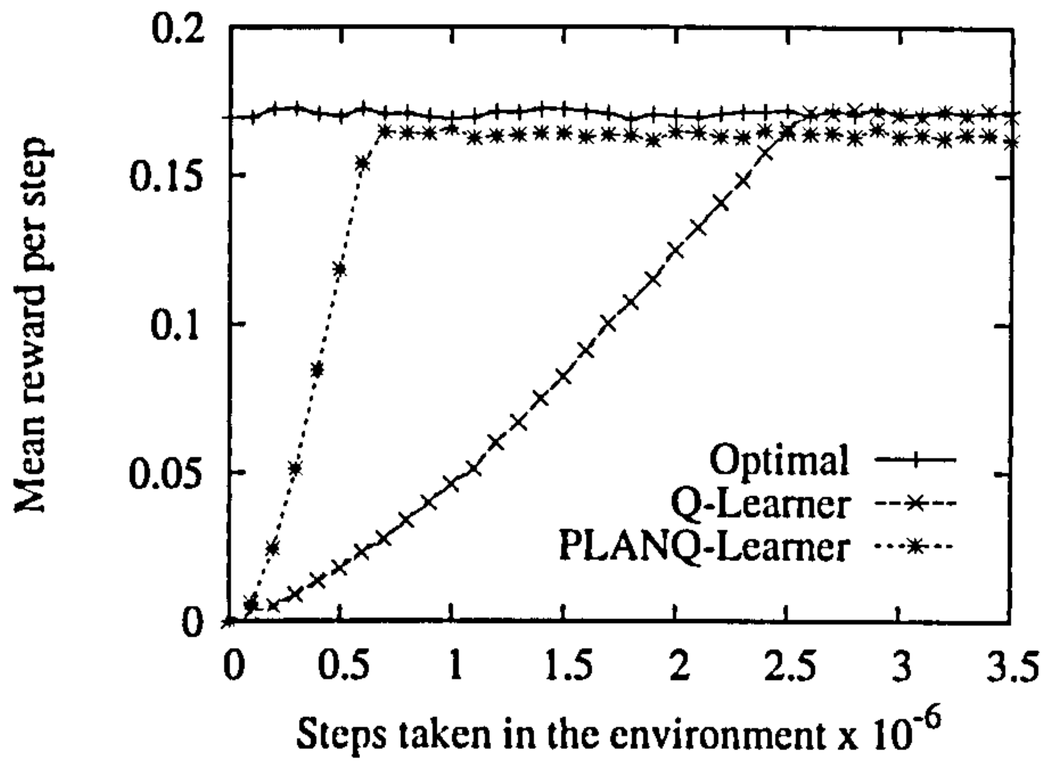


Figure 7.4: Results for experiment 1, where $n_r = 3$ and $n_g = 5$.

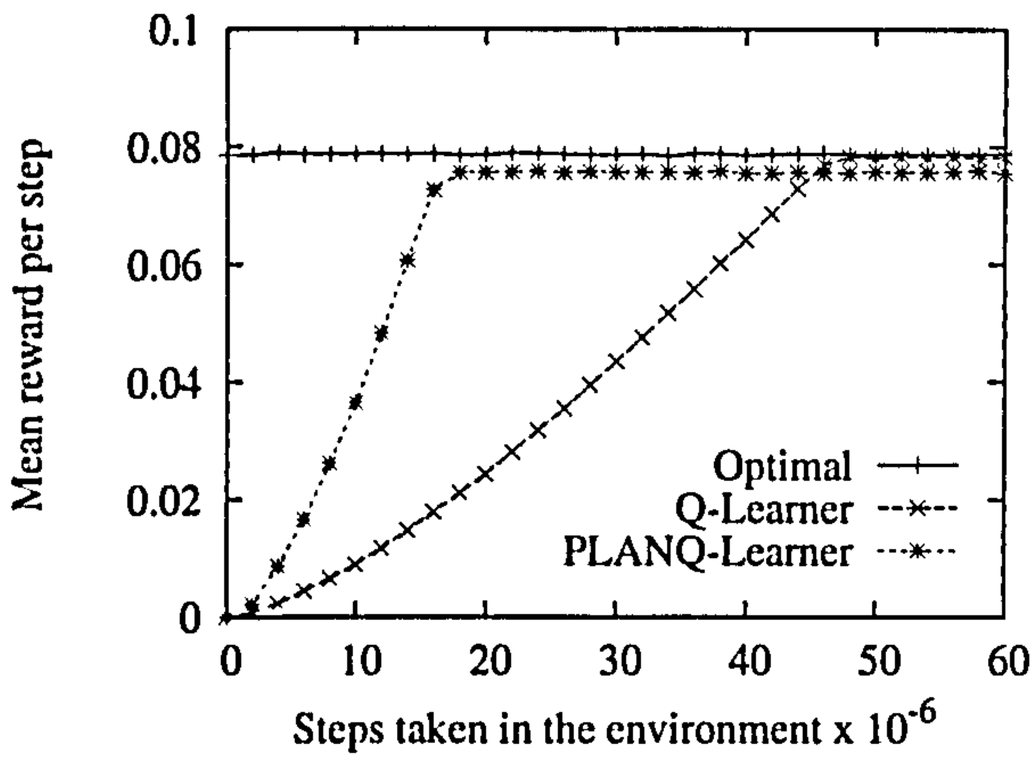


Figure 7.5: Results for experiment 1, where $n_r = 5$ and $n_g = 5$.

instance, to learn the behaviour for the NORTH operator, only the direction and the position of the agent *within* the current region are relevant. The identities of the current and destination regions are irrelevant.

Without the state abstraction, the PLANQ-learner has no way of knowing that the experience learned for moving NORTH from $r_{0,1}$ to $r_{0,0}$ can be exploited when moving from $r_{2,1}$ to $r_{2,0}$ (where $r_{x,y}$ is written to indicate the region at position (x, y) in the region grid). This leads to situations like the one in Figure 7.6, where the quality of a partially-learned operator can vary considerably in different regions of the grid world.

The PLANQ-learner needs to perform enough exploration in the state space to learn the operator separately in *all* of the regions in which it is applicable. As n_r is increased the time taken to perform this exploration approaches the time taken by the Q-learner to learn the entire problem from scratch.

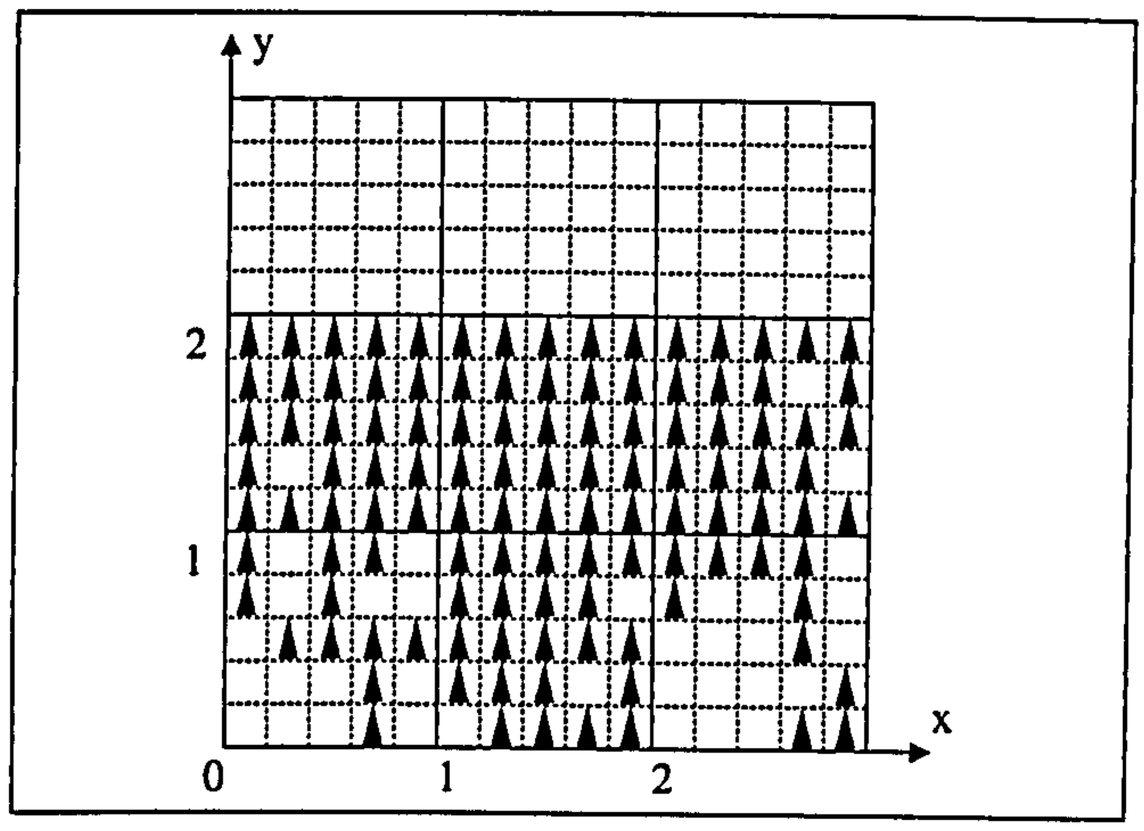


Figure 7.6: Learning the NORTH operator without state abstraction. Blank squares indicate states which have not yet converged to the optimal action. Note that NORTH is never applied in a region $r_{x,y}$ when $y = n_r - 1$.

7.6 Adding State Abstraction

To exploit the STRIPS representation of PLANQ effectively a state abstraction mechanism was added to the system. Each of the STRIPS operators was annotated with the names of the state variables which were relevant to the learning of that operator (see Figure 7.7). The Q-learner for that operator learns with a state space consisting only of these relevant variables. This speeds up learning by *generalising* the experience from one region to improve performance in another region.

However, supplying this extra information to the PLANQ-learner gives it a significant advantage over the Q-learner, and comparing their learning times is unlikely to be useful. A more revealing comparison would be with a *hierarchical* Q-learner (Barto and Mahadevan, 2003) which can take advantage of the temporal and state abstractions already exploited by PLANQ.

For the purposes of this comparison, the *Hierarchical Semi-Markov Q-Learning (HSMQ)* algorithm (Dietterich, 2000a) was selected. The HSMQ learning algorithm is a simplified version of the MAXQ-Q learning algorithm, which was reviewed in Section 3.5.4. Like the MAXQ-Q algorithm the HSMQ algorithm can learn at multiple levels of a hierarchy *simultaneously* while using a different *state abstraction* at each node of the hierarchy. It also shares the property of MAXQ-Q that it can be theoretically guaranteed to converge to a *recursively optimal*[1] policy. The key difference between the two algorithms is that HSMQ does not use the MAXQ value function decomposition. This means that the more powerful state abstraction techniques made possible by the decomposition cannot be used

[1]The definition of recursive optimality is given on page 63 of Section 3.5.

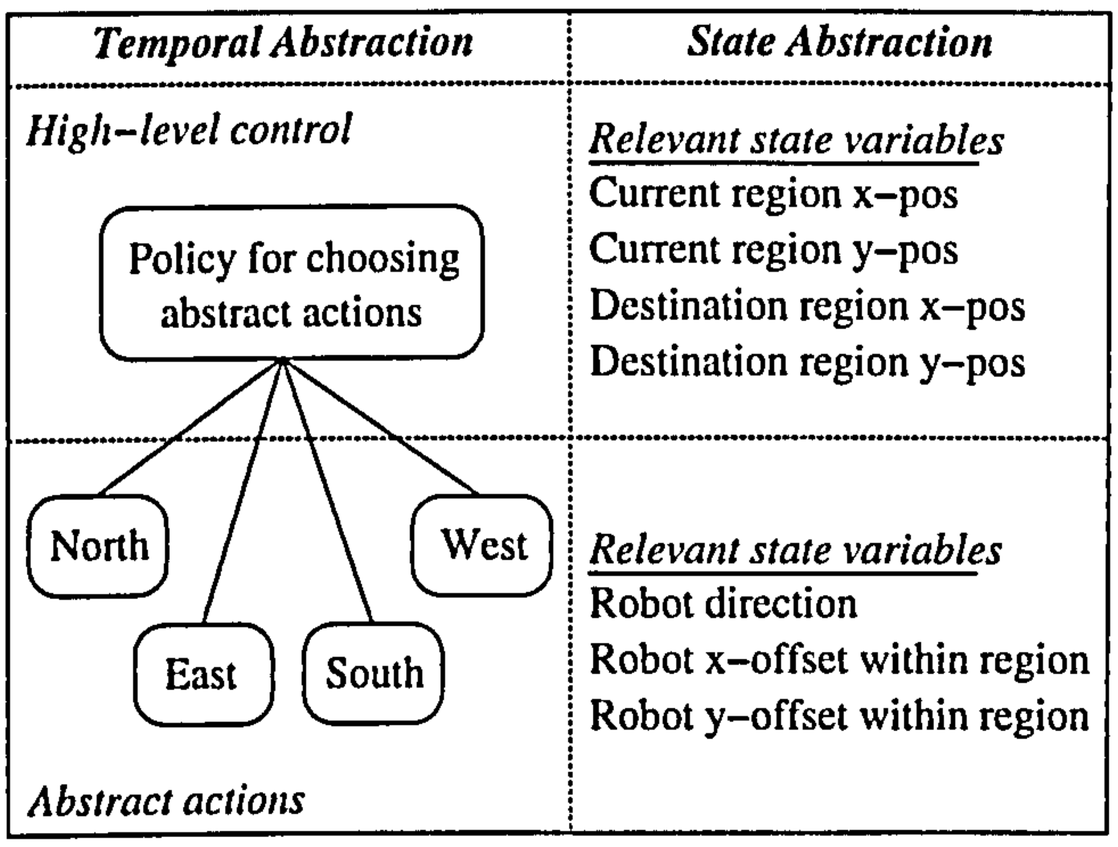


Figure 7.7: Temporal and state abstractions used by the PLANQ-learner and the HSMQ-learner. Note that of these two only the HSMQ-learner learns at the high-level as well as (simultaneously) at the low-level.

with HSMQ. For the purposes of the comparison in this chapter a relatively simple state abstraction will be perfectly adequate. Pseudocode for the HSMQ-learning algorithm is given in Algorithm 13.

The hierarchy used by the HSMQ learner (see Figure 7.7) is based on four abstract actions corresponding to the STRIPS operators of PLANQ. The desired behaviour of each abstract action is determined by an *internal* reward function supplied as part of the hierarchy. The high-level task in the hierarchy is to find a policy for executing the abstract actions which maximizes the reward accumulated in the environment. The hierarchy also encodes those state variables which are relevant to the learning of each operator, and those state variables which are relevant to the learning of the high-level policy for choosing abstract actions.

7.7 Experiment 2: Results

Figures 7.8–7.10 show the results obtained by the augmented PLANQ-learner and the HSMQ-learner for two instances of the evaluation domain. The HSMQ-learner requires an increasing number of time steps to learn a recursively optimal policy as n_r is increased. In contrast the PLANQ-learner consistently achieves a policy of a similar quality within a constant number of steps (around 100,000). Once it has learned a good policy for achieving each of the operators in an arbitrary 5x5 region (thanks to the state abstraction) the PLANQ-learner has enough information to achieve a good rate of return in a region square of arbitrary size. In other words, the number of steps needed for the PLANQ-learner to achieve a good rate of return is dependent only on n_g, not on n_r.

Algorithm 13 The Hierarchical Semi-Markov Q-learning (HSMQ) algorithm.

{$\forall n, s, a$ initialise $Q(n, s, a)$ to 0.}

{Call HSMQ(ROOTNODE(),STARTINGSTATE()) at the start of each episode.}

function HSMQ(node n, state s)

$\mathcal{A} \leftarrow$ CHILDREN(n) {actions at the next hierarchy level}

$r_{\text{total}} \leftarrow 0$ {accumulate discounted reward}

$k_{\text{total}} \leftarrow 0$ {time steps elapsed at this node}

while not TERMINATIONCONDITION(n, s) **do**

Choose action a from $\mathcal{A}$ according to the exploration strategy.

if a is a primitive action **then**

Execute a, and observe new state s' and reward r.

$k \leftarrow 1$ {k is the time taken by the action}

else

$(s', r, k) \leftarrow$ HSMQ(a, s) {recurse down the hierarchy}

end if

$S \leftarrow$ ABSTRACTSTATE(n, s)

$S' \leftarrow$ ABSTRACTSTATE(n, s')

$R \leftarrow$ LOCALREWARD(n, a, S, S')

$$Q(n, S, a) \leftarrow (1 - \alpha)Q(n, S, a) + \alpha \left[(r + R) + \gamma^k \max_{a' \in \mathcal{A}} Q(n, S', a')\right]$$

$r_{\text{total}} \leftarrow r_{\text{total}} + r\gamma^{k_{\text{total}}}$

$k_{\text{total}} \leftarrow k_{\text{total}} + k$

$s \leftarrow s'$

end while

return $(s, r_{\text{total}}, k_{\text{total}})$

end function

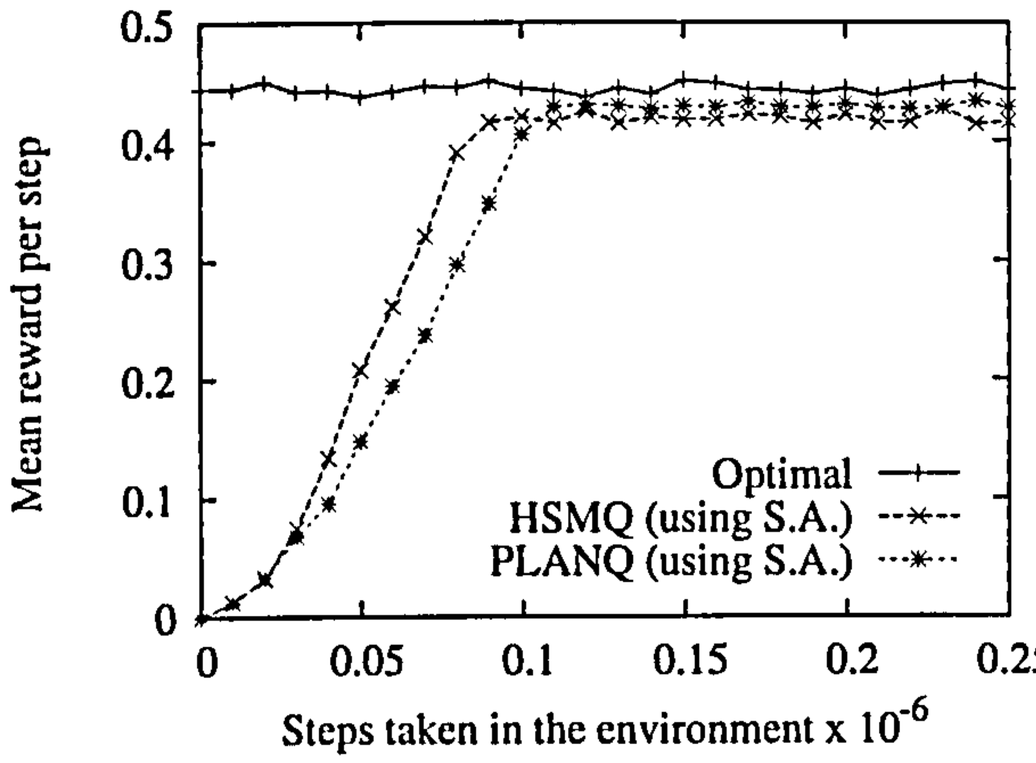


Figure 7.8: Results for experiment 2, where $n_r = 2$ and $n_g = 5$.

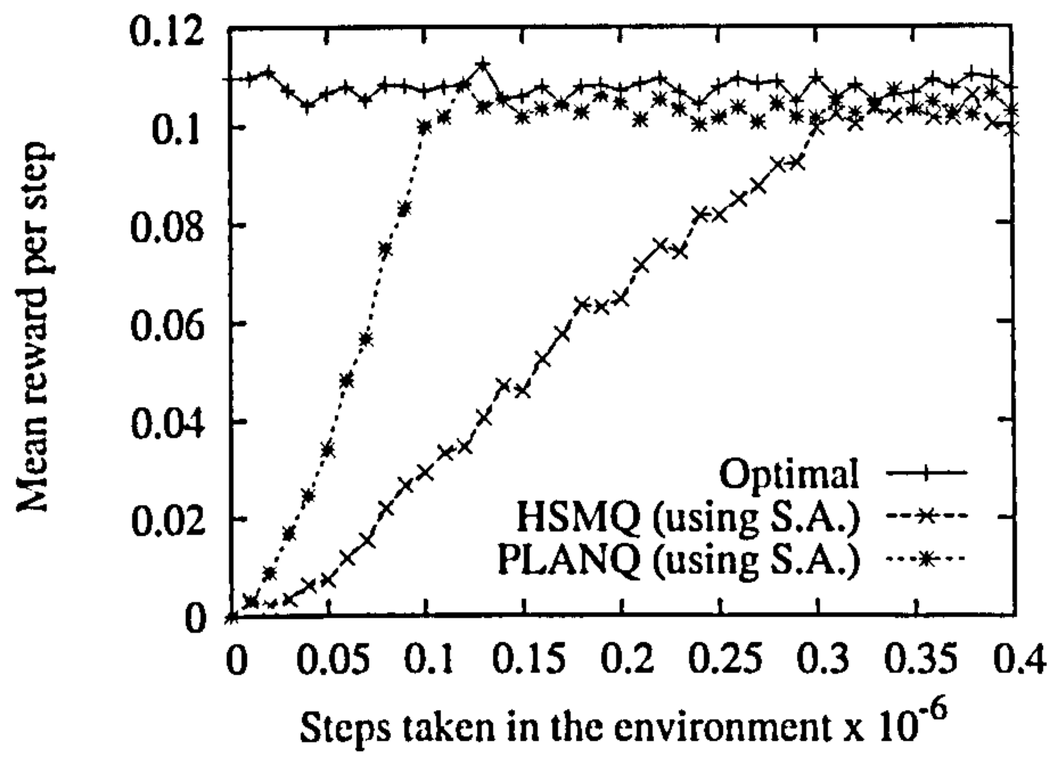


Figure 7.9: Results for experiment 2, where $n_r = 4$ and $n_g = 5$.

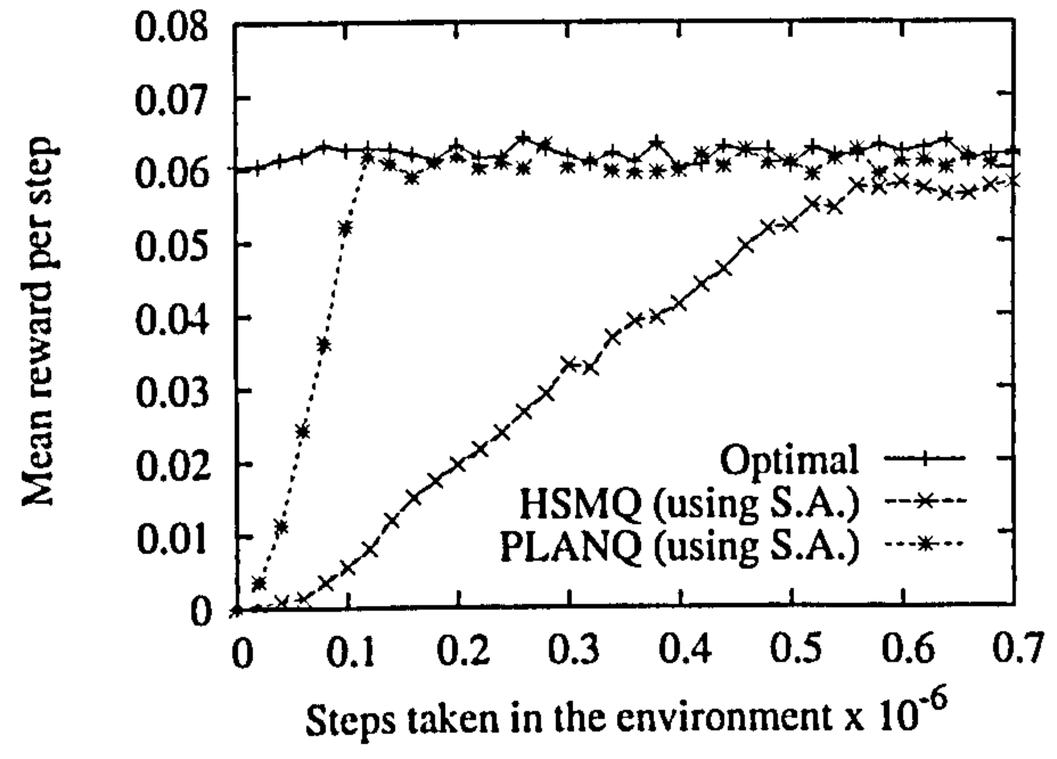


Figure 7.10: Results for experiment 2, where $n_r = 6$ and $n_g = 5$.

The HSMQ-learner on the other hand needs to learn both the low-level abstract actions *and* the high-level policy for choosing abstract actions. By exploiting both this temporal abstraction and the state abstraction information supplied with the hierarchy, the HSMQ-learner can achieve a *recursively-optimal* policy in orders of magnitude less time than the original Q-learner takes to achieve a good rate of return. However, the number of steps the HSMQ-learner needs to achieve this policy does increase with n_r, since the high-level policy becomes more difficult to learn. So as the value of n_r is increased, the PLANQ-learner outperforms the HSMQ-learner to a greater degree.

7.8 Computational Requirements

Although PLANQ achieved a good policy after fewer actions in the environment than the other agents, it is important to consider the CPU time required to calculate each action choice. The original implementation used the FF planner and the STRIPS encoding shown in Figure 7.3. This scaled very poorly in terms of CPU time. Results could only be obtained in a feasible time for values of $n_r \leq 6$.

To improve the scaling properties of PLANQ a custom GRAPHPLAN planner was written, which eliminated costly operations such as parsing and file-access, but still provided a fully functional domain-independent planner. This reduced the time required to generate each new plan, but overall the scaling performance remained poor (see Figure 7.11).

An alternative STRIPS encoding of the evaluation domain was also adopted, as shown in Figures 7.12 and 7.13. This involves encoding a subset of the natural numbers with the successor relation `s(a,b)`, and representing the x and y coordinates independently as `x(n)` and `y(n)`. Replacing the adjacency relation with a successor relation means that the number of formulae in the initial conditions is $O(n_r)$ instead of $O(n_r^2)$, which makes a great improvement to the performance of PLANQ.

Figure 7.14 shows the amount of CPU time taken for PLANQ to learn a policy with 95% optimal performance. While the HSMQ learning method is infeasible for $n_r > 20$, PLANQ can learn to make near-optimal action choices in under a minute if $n_r < 50$. However, as n_r approaches 70 PLANQ also starts to become infeasible. At this point the time required to learn a 95% optimal policy becomes completely dominated by the time required for symbolic planning in the abstract model of the domain, as shown by the measurements in Figure 7.15.

A key limitation of the PLANQ algorithm in its current form is the large variance in CPU time required per time step. On most steps an action choice can be

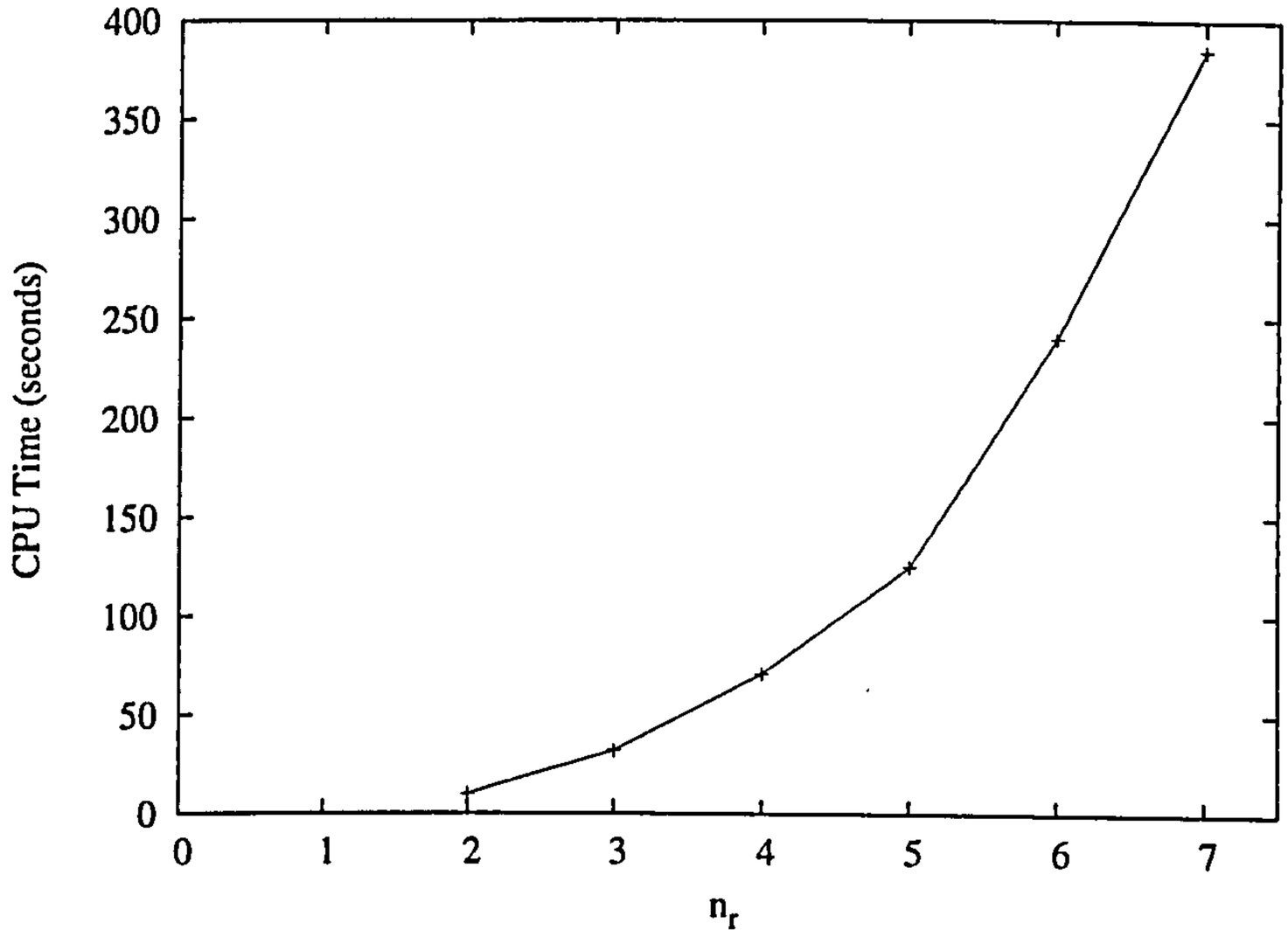


Figure 7.11: CPU time required to achieve 95% optimal performance when $n_g = 5$. This experiment used the new GRAPHPLAN implementation for planning, and the first PDDL domain encoding (given in Figure 7.3).

made in a few microseconds, but if the planner needs to be invoked the choice may be delayed for 50 or 100 *milliseconds*. For systems with real-time constraints this is clearly unacceptable.

7.9 Discussion

In this series of experiments it has been shown that a symbolic planning algorithm based on the STRIPS representation can be combined successfully with reinforcement learning techniques. This results in an agent which uses an explicit symbolic description of its prior knowledge of a learning problem to constrain the number of action steps required to learn a policy with a good (but not necessarily optimal) rate of return.

PLANQ-learning has some similarities to methods used in the RACHEL system (Ryan, 2002a), which was surveyed in Section 3.6.1. The mechanism described in this chapter for generating a reward function from a symbolic operator is also used in the RACHEL system. The two systems differ in their use of planning techniques. RACHEL uses *semi-universal* plans, which require more memory than linear plans but result in less replanning when individual operators fail. In addition, Ryan (2002a) focuses primarily on the benefits that teleo-reactive behaviour can bring to an RL-based system. In contrast, this chapter focuses on performance issues as

```
(define (problem regiongrid2)
(:domain regiongriddomain)

(:objects  n0 n1 n2)

(:init  (s n0 n1)
        (s n1 n2)
        (x n0)
        (y n0)
)

(:goal  (x n1)
        (y n2)
))
```

Figure 7.12: Alternative encoding of a PDDL problem description for $n_r = 3$.

```
(:action NORTH :parameters (?from ?to)

 :precondition (and (y ?from)
                    (s ?to ?from))

 :effect (and  (y ?to)
               (not (y ?from)))
)
```

Figure 7.13: Alternative encoding in PDDL of operator NORTH.

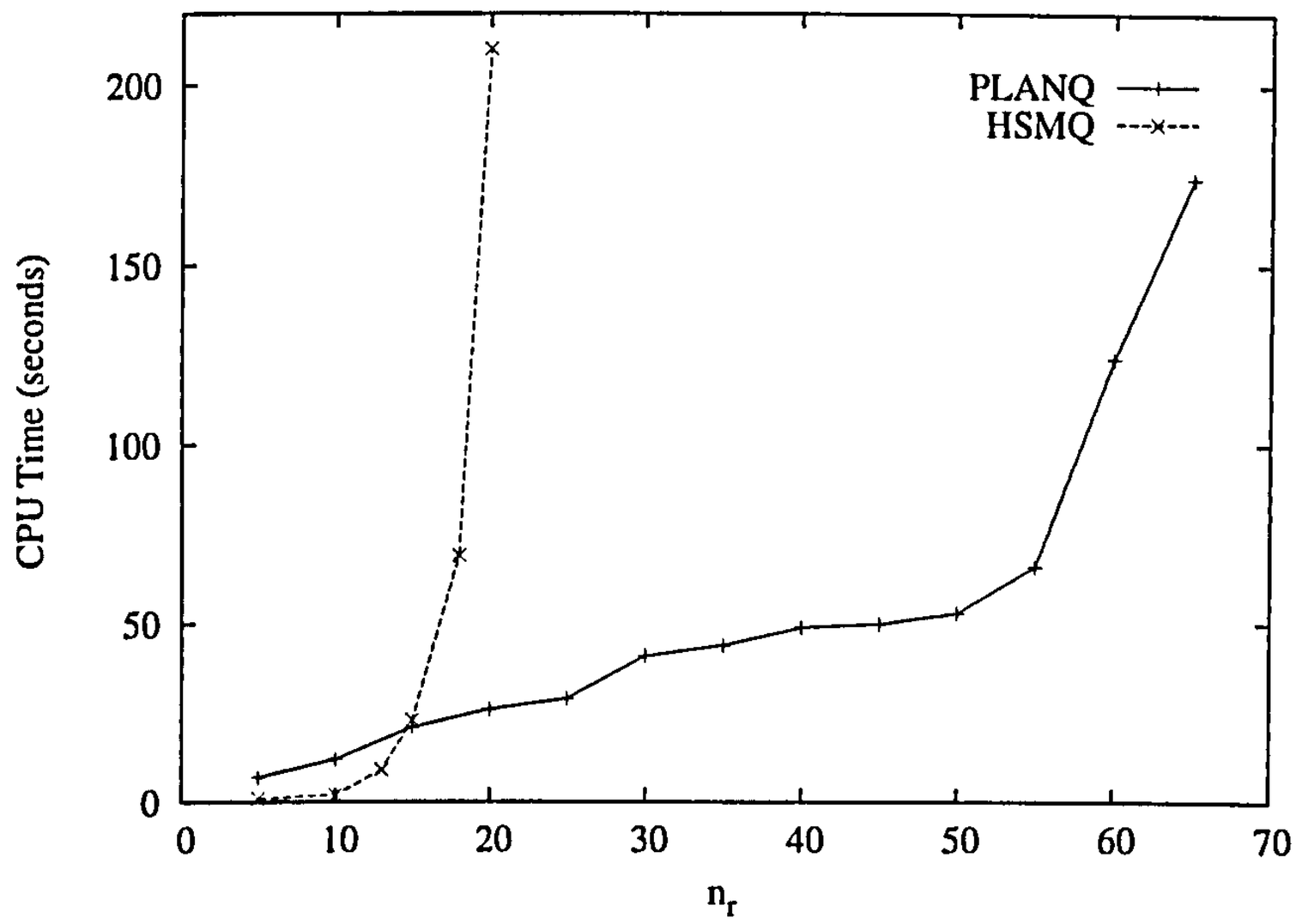


Figure 7.14: CPU time required to achieve 95% optimal performance when $n_g = 5$. This experiment used the new GRAPHPLAN implementation for planning, and the alternative PDDL domain encoding (given in Figure 7.12).

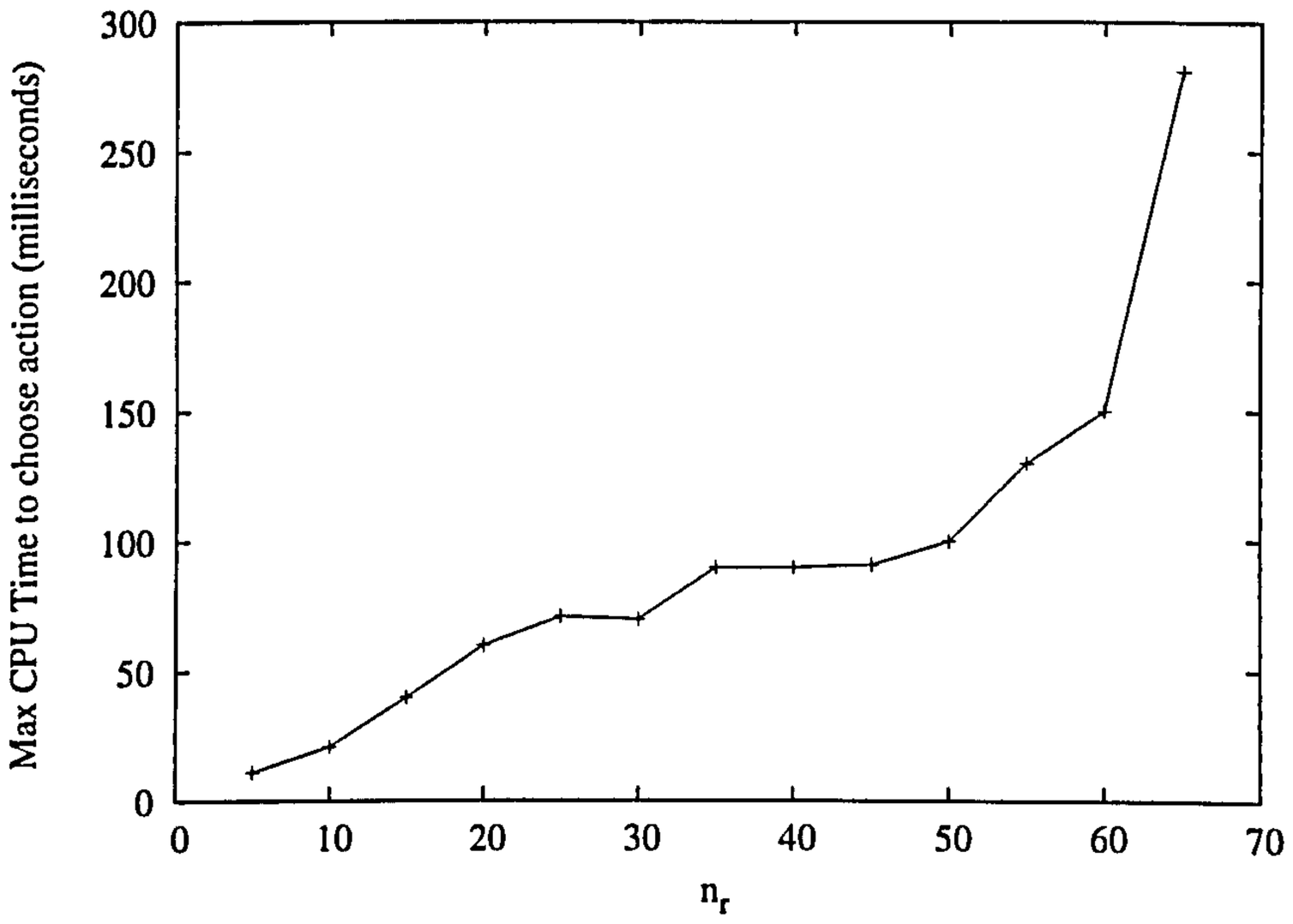


Figure 7.15: For each value of n_r this graph shows the maximum time required for the construction of a *single* plan recorded during the time required to learn a 95% optimal policy.

the size of the state space is scaled-up exponentially, with particular attention paid to the benefits of using state abstraction in combination with symbolic planning.

The STRIPS representation used in this work is limited to describing problems which are *deterministic*, *fully observable*, and *goal oriented*. To overcome some of these limitations, the PLANQ method could be adapted to use a more complex planner which can reason about stochastic action effects and plan quality. However, it is also possible that prior knowledge encoded in the limited STRIPS representation will still be useful for speeding up the learning of many problems, even if some aspects of those problems are inexpressible in this representation.

One limitation of PLANQ-Learning is the rigid separation of planning and learning in the layered architecture. The high level structure must be known in full before learning begins, and learning has the quite limited scope of implementing the low-level actions. While some domains exhibit this kind of structure, it is more often the case that solving high level aspects of a problem by planning alone is infeasible, and that some learning at this level is also necessary.

In an ideal system, a closer integration of planning and learning would enable a greater synergy to emerge from their combination. If new symbolic facts about the environment can be discovered from experience, it should be possible to add them to the knowledge base. Alternatively, the knowledge base could be used to generate likely hypotheses for the learner to evaluate in the environment.

7.10 Summary and Conclusions

The following material has been presented in this chapter:

- A novel hybrid method called PLANQ-learning, which combines high-level symbolic planning with low level reinforcement learning to implement behaviours for abstract planning operators.
- A grid-world based evaluation domain which can be quantitatively scaled up to produce a series of more difficult problems by modifying the parameters n_r and n_g.
- An evaluation in this domain comparing the performance of PLANQ-learning with that of a standard Q-learning agent.
- An analysis of the use of *state abstraction* in this domain.
- An evaluation in this domain comparing the performance of PLANQ-learning with state abstraction to a hierarchical reinforcement learner which can also exploit the specified state abstraction.

- A study of the computational resource requirements of PLANQ-learning.

From this material we can draw the following conclusions:

- Without the use of state abstraction PLANQ can only accelerate learning for the simplest of problems. As progressively more difficult problems are considered PLANQ-learning offers very little benefit over standard Q-learning.

- With the addition of state abstraction PLANQ-learning requires less time to learn a high-quality policy than either a standard Q-learning agent or a hierarchical RL agent using state abstraction.

- Using the well known FASTFORWARD planner with a naive STRIPS encoding of the domain, the computation time required to achieve a high-quality policy became rapidly infeasible as n_r was increased.

- Using a GRAPHPLAN planner with a more streamlined interface, and an improved STRIPS encoding of the domain, a high quality policy could be feasibly achieved for much larger values of n_r.

- Overall, these experiments show the potential of symbolic planning as a technique for providing high level structure for RL agents, making complex learning problems feasible by constraining learning to those areas where symbolic knowledge is inadequate or unavailable.

Chapter 8

Conclusions

This thesis has been concerned with developing methods for *scaling-up* reinforcement learning, so that high-quality policies can be learned for problems which are infeasible using standard RL algorithms. In this concluding chapter, a summary is presented of the material contained in this thesis on the two key topics of *parallelism* and *symbolic planning* in RL. In addition, the limitations of the methods presented in this thesis are examined. Finally, the benefits for the wider RL field resulting from this research are assessed and several directions for future research are presented.

8.1 Parallel Reinforcement Learning

The first hypothesis of this work, given on page 21, stated that:

> It is possible to exploit parallel hardware in reinforcement learning to achieve a speedup without sacrificing policy quality.

To demonstrate the hypothesis, a series of novel methods for parallel RL were presented in this work. These methods are based on the assumption that a *simulated* version of the target learning environment exists and that this simulation can be *replicated* on each node of a parallel computer. This means that an agent can be situated on each node, and that the group of agents can learn *in parallel*, each interacting with a *local* copy of the simulation.

Across all of the proposed methods, each individual agent uses the SARSA(λ) algorithm (Rummery and Niranjan, 1994) in combination with a *linear* value function approximator. The features for the approximator are *binary* and are generated using *tile coding* (Sutton, 1996). All of the proposed methods are based on the periodic exchange of information about the weights of each agent's value function approximator.

The three methods proposed are:

The Visit-Count Merge Method Each agent keeps a count of how many times each binary feature is *active* in the local simulation. This is known as a *visit-count.* Periodically the group of agents performs a *distributed summation* to calculate a *weighted average* of the agents' weight vectors. The weighted average gives greater credence to feature-value estimates from agents with a large visit-count for a particular feature. After each periodic merging operation, each agent replaces its local weight vector with the weighted average.

The Selective Merge Method Agents in this method exchange *recent changes* to the approximator weights instead of exchanging the absolute weight values themselves. In addition, the agents are *selective* about which weight changes are communicated. An agent sends a fixed number of weight changes with the *largest magnitudes* to the other agents during each merge operation. A merge operation consists of a *simultaneous broadcast* by all of the agents of a message containing the selection of weight change values.

The Asynchronous Selective Merge Method An extension of the selective merge method described above. *Asynchronous message passing* is used to eliminate the requirement for the group of agents to synchronize during every merge operation. This means that the broadcasts of weight changes need not be simultaneous and can be distributed more evenly over time. In addition, agents no longer need to wait for a message from every member of the group before their local weights can be updated. Individual messages can be processed and incorporated into the local value function as and when they arrive.

The *selective* and *asynchronous selective* merge methods each have a number of variants, depending on the particular mechanism which is chosen for updating the local value function approximator in response to incoming messages. The relative performance of these variants was examined during the evaluation, in addition to a comparison of the best performance achieved by each of the three methods proposed above.

8.1.1 Summary of Experimental Results

The experimental evaluation of the parallel RL methods used two different settings, both based on a *distributed memory* model of parallel computation. The first of these settings was a simulation of parallel agents, which could be run on a uniprocessor computer. This simulation did not model the time delay incurred

when messages are transmitted between the agents, so the simulation was used primarily to show that the methods converged to good policies, ignoring whether the utilised communication bandwidth was realistic. The second setting was a cluster of workstations based on commodity hardware using a 100Mbs Ethernet interconnect. The limited bandwidth of this communication network means that the large parallel speedups that can be achieved in simulation cannot necessarily be achieved in the more realistic setting.

Five example single-agent RL problems were chosen for the evaluation. Three of these are well known RL benchmark problems: the Mountain-Car task, the Pole-Balancing task and the Acrobot task. The other two problems are low- and high-difficulty instances of a stochastic grid world domain defined in Section 4.3.1. These five problems were chosen to exhibit a range of different characteristics, such as variations in the level of stochasticity, and whether the problem is goal-oriented or not. Successful performance over this whole set of problems should be a good indication that a given parallel RL method will perform well for many other problems than just the ones considered here.

The evaluation showed that the *visit-count* merge method produced large speedups in *all* the evaluation domains using the simulation of parallel agents. The speedups fell short of the perfect case of a *linear* speedup, predominantly due to the fact that there is not a perfect division of labour between the agents (i.e. there is some duplication of effort.) The success of the method using the simulation of parallel agents shows that combining approximator weights from several parallel agents *is* a valid mechanism for the agents to share intermediate results and progress more quickly towards a high-quality solution.

However, such positive results for the visit-count merge method are not repeated when the method is evaluated using the cluster of workstations. The limited bandwidth of the cluster interconnect proves to be a significant bottleneck. Even using an efficient distributed summation algorithm, each agent must transmit a weight and a visit count over the network for *every* feature in this method. All of our example problems use thousands of features, so the time required to complete the distributed summation is significant. Since the agents wait for the complete summation before learning is resumed, they waste a lot of time waiting for messages to arrive. Under these conditions, modest speedups can only be achieved in the two grid world problems. No speedup is possible in the other three problems.

The *selective* merge method was motivated by the need to reduce the communication burden of the visit-count merge method. Changes to the weights are broadcast instead of absolute values, and only a fixed number of the *largest* changes are sent in each agent broadcast. The evaluation shows that this approach is suc-

cessful, allowing speedups to be obtained in *all* the domains using the cluster of workstations. However, some of the speedups are very small, especially in the Acrobot task. This appears to be because the Acrobot task requires a large number of features for the approximator, but does not require many simulation steps to converge to a good policy. This means that the communication overhead is large in comparison to the relatively small amount of time required for a single agent to converge.

The *asynchronous selective* merge method eliminates the requirement for each agent to wait for a message from each of its peers before a new period of learning can begin. This eliminates the *synchronization penalty* inherent in the two previous methods. The evaluation shows that the time saved by making this modification is significant, allowing much better speedups to be obtained in *all* the evaluation domains. The greatest improvement is shown in the Acrobat task, where previously only a very small speedup could be obtained using the selective merge method. This asynchronous method is the most effective algorithm for parallel RL developed in this work.

With regards to the variants of the latter two methods, the asynchronous selective merge method had one variant which produced the best performance in a wide variety of situations, so this is the one to prefer. The selective merge method on the other hand had no clear winner out of the proposed mechanisms for updating the approximator weights in response to messages received from the group. The best performing mechanism varied according to which particular RL problem was being considered. In spite of this, the performances of *all* the mechanisms were fairly similar, so there is not a large penalty for selecting one of these mechanisms and using it for all the problems.

8.1.2 Research Benefits

The immediate benefit of this research to the RL community is that the time required to prepare and run RL experiments can be reduced, either by using dedicated parallel hardware or by utilising groups of idle workstations in a laboratory. Obviously empirical results for single-agent algorithms must still be generated for research purposes, so parallelism should not be used in every experiment. However, even setting up a single-agent experiment requires a number of steps beforehand such as determining a good set of approximator features, selecting good values for RL parameters such as the learning rate and discount factor, and determining the number (and length) of runs required to achieve a good policy with high probability. Using a parallel approach for these preliminary stages could dramatically reduce the turnaround time for a given experiment.

Another way to view the immediate benefit of this research is that if a *fixed time* is available for learning then a policy of *higher quality* can be learned in this time. This is particularly relevant for the case when offline learning is used to find a high quality stationary policy for deployment in some domain. In such cases it is not unreasonable to set aside hours or days of computation time to generate this policy. If parallel hardware can be made available, it is likely that the quality of the deployed policy will be increased if the same time is available for learning.

In the longer term, this research is also potentially relevant to the problem of *multi-agent learning.* In this thesis I have focused on single-agent learning problems that can be *simulated.* This is a relatively simple way of ensuring that the parallel agents learn a value function corresponding to *the same problem.* In multi-agent learning, several agents are situated in the same environment. This means that their actions can potentially affect each other, introducing new challenges for machine learning researchers. However, suppose that two agents are in different locations of the environment, so that their actions do not directly affect each other. Suppose also that the agents are working on two similar subtasks of the overall problem. By using *abstraction* to ignore state variables which are irrelevant to the subtask, it may be possible to make the subtasks look like two *(almost) identical* but *separate* learning problems. If the agents also have a communication channel, the methods described in this thesis become directly applicable. Thus the use of parallel RL in a multi-agent context is bound up with the problem of detecting and/or defining abstract hierarchies for planning, acting and learning in multi-agent domains.

Parallel RL methods do not obviate the need for exploration, generalization, abstraction and relational representation in reinforcement learning. However, neither are parallel methods incompatible with these other techniques. While the work in this thesis has focused on SARSA(λ) and linear approximation, there is nothing which ties the approach of exchanging value function weights to either of these specific techniques. As long as there is some representation of a value function, a variation of the methods in this work is likely to be applicable. Parallelism is best viewed as another technique in the RL "toolbox", to be deployed when appropriate in combination with one or more of these other techniques for scaling-up RL.

8.1.3 Research Limitations

The main limitation of the methods reported in this thesis is that values for parameters p and f_{com}, which are important for achieving good parallel performance, must currently be found by trial and error. Parameter p defines the (average) num-

ber of simulation time steps which elapses between successive message broadcasts. For the selective methods, parameter f_{com} controls the number of weight changes which can be included in a single broadcast. Together, these parameters are used to strike a balance between communicating often enough to converge quickly to a good policy, but not communicating so often that most of each agent's time is spent sending, receiving and processing messages. Although the overall performance is insensitive to small variations in these parameters, it is a prerequisite for good performance that the parameter values are not many times smaller or larger than the optimum values. There are already several RL parameters (such as α, λ and ϵ) which require a trial and error approach to tailor an algorithm to a specific domain, so it is unfortunate that these parallel methods introduce additional parameter choices. However, there is still the possibility that a heuristic method for selecting p and f_{com} could be defined, based on system- and domain-specific properties that can be measured very quickly: the time required for an agent to send a message containing the entire set of value function weights to another agent, and the time required for an agent to execute a single simulation step and update the local value function in response to this step.

In this work I have focused on a *distributed-memory* model of parallel computation, which maps well to clusters of workstations. An advantage of focusing on this model is that the methods presented in this work (which are based on message passing) should also work well on a *shared-memory* computer such as a *symmetric multiprocessor (SMP)* computer. This is because agents can easily exchange messages by copying data into shared memory. These methods will therefore perform well on a wide variety of parallel architectures. However, alternative approaches to parallel RL which *specifically* target a shared memory model may perform even better on SMP computers. Determining the form of these alternative approaches and the size of the performance improvement that can be achieved is a possible direction for future research in this area.

A further limitation of this work is that the empirical analysis of the parallel RL methods and the resulting conclusions about which of the methods perform best are, to a certain degree, tied to the specific properties of the cluster of workstations used in the evaluation. The ratio of the bandwidth of the cluster interconnect to the processing speed of the CPU at each node is an important indicator of the overall performance that can be achieved. If we took the cluster used in this work and replaced the interconnect with a faster *1Gbs Ethernet* interconnect, larger parallel speedups would be reported than those in this thesis. In *all* cluster systems however, the efficient use of whatever bandwidth is available remains extremely important, and with appropriate choices for the p and f_{com} parameters, the meth-

ods reported in this thesis can be adapted to get good performance out of any cluster of workstations.

8.1.4 Future Research Directions

There are a number of directions in which this work on *parallel RL could be* extended in the future.

Selection and Update Mechanisms

In this thesis, only one mechanism was used to select which weight changes were most important to communicate, namely those weight changes with the *largest magnitudes.* This approach is quite effective, prioritizing those weights in the local value function with the most pronounced differences from the group's existing knowledge of the environment. However, there remains the potential for improvement in this area. One flaw of the current approach is that information about the f_{com} largest weight changes is always sent in every broadcast, no matter how small these changes are. Towards the end of the experiment, when the VFA has all but converged, many of these changes could be zero, or very close to zero. Communicating these tiny changes is unlikely to have any benefit, so an additional mechanism to exclude insignificant changes may improve performance. One alternative selection mechanism is to track the *range* of values observed for a particular weight and communicate a weight change only when the value of the weight moves outside the previously observed range. Another alternative is to track the *mean* and *standard deviation* of each weight value over time, prioritizing communication of weights which undergo large changes relative to the standard deviation. This may improve performance in situations a number of state-action pairs have a very similar mean reward but very different levels of variance.

Other Approximation Architectures

The evaluation of the parallel RL methods in this thesis used *tile-coded linear approximators* based on *binary* features to represent the value functions of the agents. The *visit-count* merge method described in Chapter 4 makes use of a count of the number of times each binary feature is active (i.e. has value = 1), so this method is strongly tied to the use of binary features. On the other hand, the *selective* and *asynchronous selective* merge methods do not use visit counts, and are therefore applicable to *any* linear function approximation architecture, such as using *radial basis functions* to generate learning features. A further empirical study using a variety of approximation architectures would provide stronger evidence of

the potential of these methods for accelerating RL with linear approximation.

The parallel RL methods described here could also be applied to *non-linear* neural network approximation architectures, although the combination of neural networks and RL has proved unreliable in previous research (see Section 3.4.6).

A more interesting question is whether these methods can be applied to *memory based* (also known as *instance based*) approximators, which have been shown to be stable and successful approximators for RL (see Section 3.4.4). An assumption of this thesis has been that all agents use an *identical set* of learning features which are *ordered*. It is important that the features are ordered, since it means that each feature (and hence each linear approximator weight) can be identified by an *index*. To specify a *sparse* set of changes to the set of weights, a message can be constructed efficiently from a set of pairs, where each pair consists of *an index* and *a weight change*. With a memory-based approximator, there is no longer a fixed set of features. Each agent simply stores in memory all the different states that it has visited. The agents therefore no longer have a common frame of reference for value estimates, which means that some number of *exemplar* states must be exchanged over the communication network to establish a basis for communication. In domains which require many state variables to describe the environment, the exchange of such exemplars will be expensive in terms of network bandwidth. The development of parallel RL methods which use memory-based approximation represents a major challenge for future research.

Theoretical Analysis

In this thesis, I have presented strong empirical evidence (using a wide range of evaluation domains) that parallel RL based on merging approximator weights can speed up learning without compromising the final quality of the learned policy. While theoretical proofs of convergence have not been provided for the methods described in this thesis, previous research in RL has shown that methods without a proof may still be of great practical importance (Sutton, 1996). However, to increase our confidence in these methods, it would be good to prove that they converge to policies which are at least as good as those learned by (single-agent) SARSA(λ). A major difficulty here is the fact that there is not currently a proof of convergence for the single-agent SARSA(λ) algorithm if *both* linear approximation *and* a GLIE policy[1] are used, as they are in this thesis.

In the short term, until a convergence proof is found (or shown not to exist) for the single-agent algorithm, the best that we could do is to show that convergence

[1]GLIE stands for "Greedy in the Limit with Infinite Exploration." See Singh et al. (2000) for further details.

proofs exist for two specific situations:

1. Each of the parallel agents *does not explore*, but follows a *fixed*, *stationary* policy for its entire lifetime.

2. The parallel agents do explore, but $\lambda = 0$ and each agent's value function is represented with a table (i.e. no function approximation).

In each of these cases a convergence proof exists for the restricted single-agent algorithm. For case 1, a modified version of the proof of Tsitsiklis and van Roy (1997) can be used to guarantee convergence. For case 2, the proof of Singh et al. (2000) applies. Extending these results to the parallel case would provide a degree of confidence that the mechanisms of parallel merging do not interfere with the long term convergence of SARSA(λ), even if we cannot show that the result still holds in the most general case.

Shared-Memory Parallel Systems

This thesis has focused on a *distributed-memory* model of parallel computing, under which the parallel RL agents must use *message passing* to communicate intermediate results between themselves. This model maps well to an implementation on a cluster of workstations using commodity hardware. An alternative approach would be to use a *shared-memory* model of parallel computing, with an associated implementation on a *symmetric multiprocessor (SMP)* computer. One advantage of the methods described in this thesis is that they can easily be deployed on an SMP computer. This would be accomplished with a message-passing implementation in which agents write message data into shared memory rather than using an interconnection network.

In addition, a *shared-memory* model allows the possibility of other parallel RL methods where the parallel agents use and update a *shared* value function representation. This idea was investigated in the context of multi-agent learning by Tan (1993), though to my knowledge there has not been an implementation using this concept on real parallel computing hardware rather than in simulation. This approach raises the question of how the integrity of the shared value function data structure can be maintained while still allowing fast efficient read/write access for the agents. The cost of communication in the shared-memory model arises from *locking* areas of shared memory rather than delays in message transmission. A study which compared the effectiveness of both the shared-memory and distributed-memory approaches to parallel RL would also provide new insights.

8.2 Symbolic Planning and RL

The second hypothesis of this work, given on page 23, stated that:

> A hybrid planning-learning system based on a high-level STRIPS-based planner and low-level reinforcement learning will exhibit better scaling properties than both standard and hierarchical RL algorithms for goal-oriented learning problems.

To demonstrate this hypothesis the novel PLANQ-learning algorithm was presented in this work. PLANQ-learning is applicable to RL problems where the only non-zero reward is received when the agent reaches one of a set of goal states (i.e. goal-oriented problems.) A high-level STRIPS plan is used to guide the agent towards the goal states, providing the high-level structure for the agent's policy. The agent has access to an interface which, given a low-level state, will return a high-level symbolic description of that state and the current goal. The agent also has symbolic descriptions of a number of abstract operators, which initially have no implementation in terms of the available low-level actions.

An efficient STRIPS planner based on the GRAPHPLAN algorithm (Blum and Furst, 1997) constructs a high-level plan to achieve the current goal using the abstract operators. The implementation of each operator is provided by learning a policy at the low-level using RL. The reward function used to learn each low-level policy is derived *directly* from the preconditions and effects of each abstract operator. The symbolic description of the current state is monitored as the agent interacts with the environment. The agent only receives a positive reward if the postcondition of the current operator is achieved without violating the preconditions.

8.2.1 Summary of Experimental Results

To evaluate the scaling properties of the PLANQ-learning algorithm, a deterministic grid-world domain was selected. Instances of this domain could be scaled up in quantitative steps to create more difficult problems by modifying the defining parameters of a problem instance.

The PLANQ-learning algorithm was compared with the standard Q-learning algorithm (Watkins, 1989) in a series of increasingly difficult instances. In all of these instances, PLANQ-learning produced the best performance out of the two algorithms. However, the trend was observed that the difference in performance between the two algorithms became smaller as the problem instances became larger. This trend arose because the PLANQ learner was not able to effectively generalize

experience gained in different areas of the state space. This result suggests that for larger problems, high-level planning is essentially useless without state-abstraction.

Introducing a state-abstraction mechanism meant that the PLANQ-learner was able to vastly outperform the Q-learning agent. However, this comparison was unfair since the Q-learning agent was not able to exploit the state abstraction mechanism available to the PLANQ-learner. A more balanced comparison was possible with an agent using the HSMQ-algorithm (Dietterich, 2000a), since this algorithm is hierarchical and can make use of the state abstraction. This comparison showed that as the number of regions was increased, the PLANQ-learner required a constant number of environment time steps to converge to a near-optimal policy, whereas the number of time steps required by the HSMQ-learner continued to increase. This is because the PLANQ-learner only needs to learn low-level policies, whereas the HSMQ-learner needs to learn both high- and low-level policies simultaneously.

In terms of computation time rather than environmental time steps, learning a high-quality policy using the PLANQ-learner remained feasible for much larger numbers of regions than either the Q-learning or HSMQ-learning algorithms. However, this property comes with the disadvantage of an extremely high variance in the computation time expended per time step, since on time steps where replanning is required the time delay to generate a new STRIPS plan can be significant. Where no replanning is required, the agent's responses are almost instantaneous by comparison.

8.2.2 Research Benefits

The PLANQ-learning algorithm demonstrates how concepts of hierarchy (see Section 3.5) and symbolic planning (see Section 3.6) can be combined with reinforcement learning to create a hybrid planning-learning method. This type of hybrid approach is most appropriate for learning problems which can be described at a high level as a number of distinct sequential stages. If a large-scale learning problem has this kind of structure, the results presented in this thesis have shown that PLANQ-learning is likely to perform much better than standard flat or hierarchical RL algorithms.

The results of the experiments reported in Chapter 7 indicate a number of important guidelines for researchers wishing to implement a hybrid learning method of this kind. The use of *state-abstraction* was shown to be vital for generalization in large, regular domains. Using a symbolic plan to structure a problem solution without state-abstraction is not going to scale well in such domains. Optimization of the interface between the agent and the planner was also shown to be impor-

tant for controlling the computational requirements of the hybrid method. This is especially important in simulated learning environments, where it is very cheap to generate new experiences. Finally, encoding the STRIPS domain theory to facilitate efficient planning is also vital for controlling computational complexity. As is often the case with AI methods, selecting the wrong problem representation will significantly degrade the effectiveness of the method.

8.2.3 Research Limitations

The main limitation on the applicability of PLANQ-learning to RL problems is that an appropriate STRIPS domain theory must be found which describes a high-level solution for a given problem. This requirement for a domain theory places a number of restrictions on the type of problems which can be solved. The problems must be *goal-oriented* (i.e. the only non-zero reward is received when a *goal state* is reached) and low-level stochastic effects must be hidden by the policies learned for the STRIPS operators. In addition, each problem must have a suitable internal structure such that high-level planning is advantageous for achieving a good return in the domain.

PLANQ-learning also has limitations which are exhibited by hierarchical RL methods in general, such as the fact that a hierarchical policy will be sub-optimal if the high level abstract operators used cannot express the true optimal policy. It would be extremely useful if the hierarchical policy could be used as an intermediate step towards learning the optimal policy. However, this is generally not possible without giving up the use of state-abstraction and the major improvements in performance which are made possible by it.

The requirement for a STRIPS domain theory and goal to describe an RL problem can be viewed as a kind of *prior problem knowledge*. The requirement for this knowledge can be seen as a limitation, since the knowledge base must generally be created by hand before learning begins, but it can also be seen as an advantage of the approach, since if prior knowledge of the problem *does* already exist it can be exploited effectively by the PLANQ-learning algorithm.

8.2.4 Future Research Directions

There are a number of directions in which the work on PLANQ-learning could be extended in the future.

Advanced Symbolic Planning

In this thesis a planner based on the GRAPHPLAN algorithm was used, which is based on the STRIPS representation of states, goals and actions. One restriction this places on the problems we can solve with PLANQ-learning is that only a *deterministic* set of effects can be specified for each high-level operator. If the RL policies which implement the operators have any stochastic side-effects, the agent will expend a great deal of effort on *replanning*. A more advanced planner based on probabilistic STRIPS operators (Kushmerick et al., 1995) would be able to avoid some (or all) of the replanning, although typically algorithms for probabilistic planning are more computationally expensive than deterministic planning.

A second restriction the STRIPS representation imposes on the problems we can solve is the fact that operators have no *cost*. This will produce good results only in domains which are *purely* goal-oriented, i.e. where the only objective is to reach a goal state in as short a sequence of actions as possible. A more general representation would allow for the fact that the various *STRIPS operators* accumulate different levels of reward while executing in the environment. A planning algorithm using such a representation could favour where appropriate those operators with a better return. However, in combination with probabilistic actions, this level of generality quickly approaches the complexity of planning in a Relational MDP (van Otterlo, 2005). Adapting the PLANQ-learning algorithm to such a general case is likely to be extremely expensive in terms of computation time. There is definitely an argument in favour of using as simple a planning representation as possible to keep computation costs feasible, even if this representation is not a perfect match to the domain in some circumstances.

Multi-Level Hierarchies

The PLANQ method described in this thesis uses a two-level hierarchy, where symbolic planning is used for the high-level component and reinforcement learning is used for the low-level component. While a symbolic plan can often be used to model the high level structure of a policy, this rigidly defined hierarchy does not have much flexibility, and cannot be applied to problems with a more complex solution structure. A more general method would allow hierarchies with multiple levels, where each node of the hierarchy at any of the levels can be either a *symbolic planning* or a *reinforcement learning* node. Which type of node is used would depend on which method was most appropriate for that part of the problem and on which domain knowledge was already available or easy to add. It is also worth recognising that, in some situations, a *hand-coded program* for selecting actions

would be more appropriate than either planning or RL. Allowing nodes to contain programs as well would further broaden the number of domains for which policies can be feasibly learned, although obviously it is impossible to guarantee that the quality of such a policy would be near that of an (infeasible to generate) optimal policy. Note that this approach would be quite similar to the *layered learning* method of Stone (1998).

8.3 Concluding Remarks

Reinforcement learning in large-scale domains remains an extremely challenging area of research. Auspicious theoretical guarantees of convergence (already shown for most of the standard algorithms) belie a host of practical problems in the application of these techniques. These include managing the scope and depth of *exploration* in a domain, *generalizing* between states in a stable manner and using *relational* representations of state to combat the curse of dimensionality. Active research continues in each of these areas, gradually broadening the range of problems to which RL can be applied effectively.

This thesis provides two main contributions to the overall goal of large-scale reinforcement learning. The algorithms for *parallel RL* presented here allow parallel computing hardware to be used for RL. This means that high-quality policies can be found more quickly for domains which can be simulated. The significance of this for the broader field is that problems of *borderline feasibility* can potentially now be solved in minutes rather than hours, or hours rather than days. In the area of relational representations, the hybrid PLANQ-learning algorithm presented in this thesis shows how a symbolic plan can be used as the overall structure for an RL policy. It has also been demonstrated that the combination of symbolic reasoning, state-abstraction and low-level RL can feasibly generate high-quality policies for much larger learning problems than hierarchical RL alone.

Despite the large body of existing work on scaling-up RL, fundamental problems such as the *curse of dimensionality* and the *exploration/exploitation* dilemma continue to challenge researchers in this area. It may also be observed that as RL researchers continue to enlarge the set of feasible application domains, a range of techniques from other areas of Artificial Intelligence research are increasingly incorporated. As surveyed in Chapter 3, this includes work in *Bayesian statistics*, *heuristic search*, *regression*, *classification*, *classical planning*, *decision-theoretic planning*, *hierarchical task networks* and *inductive-logic programming*. In its broadest sense therefore, RL encapsulates several of the most important goals in AI research, namely planning effective action in the world whilst learning from ex-

perience in an interactive, agent-oriented setting. While the scope of these goals makes a completely general RL approach seem pretty distant, the overall relevance of RL to the "AI problem" suggests that future progress in RL may allow us to make significant progress in the design of intelligent agents.

Increasing the number of successful practical applications of RL seems to require progress in two separate directions. The first direction involves "cleaning up" various aspects of low-level RL. The number of ad-hoc parameter choices that are required for many algorithms needs to be reduced. The theoretical basis for RL in the presence of generalization requires further work. There is also a need for standard mechanisms to address fundamental trade-offs such as exploration vs. exploitation, or sample-complexity vs. computational complexity. These low-level problems currently receive a great deal of attention in the research community.

The second direction is addressing the problem of scale, which may involve the use of hierarchy, abstraction, relational representations, symbolic reasoning, or perhaps other mechanisms for learning, organizing and applying an agent's knowledge. While research is progressing well along a number of fronts in this direction, there remains little understanding about how RL should be integrated with symbolic planning, hand-coded policies, or even high-level cognitive approaches to experimentation and discovery. A unifying framework is obviously unlikely, given the overall diversity of the techniques available for scaling-up RL, but there remains the opportunity to develop a new family of methods for reinforcement learning *in the large*. By maintaining the goal of an interactive agent which receives rewards and improves in performance with experience, but relaxing various aspects of low-level RL which are less appropriate at a high level, a new front of RL research could be established. This could enable the development of a whole new generation of agents which learn from direct interaction and rewards.

List of Mathematical Symbols

Symbol	Description
a	An action.
A	A set of actions.
c	A *visit-count*, which indicates the number of times a binary feature is active during a given period.
f	The total number of features used by a particular function approximator.
f_{com}	The number of feature weights communicated by a *selective* parallel RL agent in a single merge period.
o	An observation in a partially observable environment.
O	An observation function, which defines the probability of observing o after taking action a in state s.
p	The *merge period*, which is the period between consecutive communications for parallel RL methods *in this thesis*.
Q	A value function which maps each state-action pair to a value.
Q^π	A state-action value function corresponding to the expected return of policy π.
Q^*	A state-action value function corresponding to the expected return of the *optimal* policy π^*.
r	A reward.
R	A reward function, which defines the expected reward for a transition (s, a, s').
s	A state.
S	A set of states.
t	A discrete time step.

Symbol	Description
t_{lim}	The *proportion* of the time for a single parallel run which is used to decay the α and ϵ parameters linearly towards zero.
T	A transition function, which defines the probability of each transition (s, a, s').
V	A value function which maps each state to a value.
V^π	A value function corresponding to the expected return of policy π.
V^*	A value function corresponding to the expected return of the *optimal* policy π^*.
α_0	The initial value of α before a period of linear decay begins.
α	The *learning rate*, which determines how quickly a value function is adjusted towards new estimates of state values.
γ	The *discount factor*, which determines the relative worth of short-term rewards and long-term rewards.
ϵ	The probability of choosing a random action at each time step when using the ϵ-*greedy* exploration strategy.
ϵ_0	The initial value of ϵ before a period of linear decay begins.
θ	An adjustable weight of a function approximator.
θ_{init}	The initial value given to all the weights of the function approximator.
λ	The *eligibility trace* parameter, determining the extent to which new value estimates affect the values of states visited in the past.
π	A *policy* which maps each state-action pair to the probability of choosing that action in that state.
π^*	The optimal policy.
ϕ	A learning feature (or alternatively, a basis function) for a function approximator.
Ω	Set of possible observations in a partially observable environment.

List of Abbreviations

Abbreviation	Description	Further Details
AHC	Adaptive Heuristic Critic	Section 3.4.6
AI	Artificial Intelligence	
BSP	Bulk Synchronous Parallel	Section 3.7.1
CPU	Central Processing Unit	
FF	"Fast Forward" (planning algorithm)	Section 7.1
HSMQ	Hierarchical Semi-Markov Q-learning	Section 7.6
MDP	Markov Decision Process	Section 2.2
MIMD	Multiple Instruction Multiple Data	Section 3.7.1
MPI	Message Passing Interface	Section 3.7.1
MPICH	Message Passing Interface CHameleon (open source implementation of MPI)	Section 4.3
MSE	Mean Squared Error	Section 3.4.2
PDDL	Planning Domain Definition Language	Section 7.1
POMDP	Partially Observable Markov Decision Process	Section 2.6
PRAM	Parallel Random Access Machine	Section 3.7.1
RL	Reinforcement Learning	Section 2.1
SARSA	"State Action Reward State Action" (the name of an RL algorithm)	Section 2.4
SMDP	Semi-Markov Decision Process	Section 3.5.3
SMP	Symmetric Multi-Processor	Section 3.7.1
SPI	Structured Policy Iteration	Section 3.6.2
SPUDD	Stochastic Planning Using Decision Diagrams	Section 3.6.2
STRIPS	STanford Research Institute Problem Solver	Section 3.6.1
TD	Temporal Difference learning	Section 2.4
VFA	Value Function Approximation	Section 3.4

List of References

M. N. Ahmadabadi and M. Asadpour. Expertness based cooperative q-learning. *IEEE Transactions on Systems, Man and Cybernetics*, 32(1):66–76, 2002.

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

E. Alonso, D. Kudenko, and D. Kazakov, editors. *Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning*, volume 2636 of *Lecture Notes in Computer Science*, 2003. Springer.

C. W. Anderson. Strategy learning with multilayer connectionist representations. In *Proceedings of the 4th International Workshop on Machine Learning*, pages 103–114, 1987.

C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems.* PhD thesis, University of Massachusetts, Amherst, MA, 1986.

D. Andre. *Programmable Reinforcement Learning Agents.* PhD thesis, University of California, Berkeley, 2003.

T. Archibald. Parallel dynamic programming. In L. Kronsjö and D. Shumsheruddin, editors, *Advances in Parallel Algorithms.* Blackwell Scientific, 1992.

T. W. Archibald, K. I. M. McKinnon, and L. C. Thomas. Serial and parallel value iteration algorithms for discounted markov decision processes. *European Journal of Operational Research*, 67(2):188–203, 1993.

L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.

M. Barreno and D. Liccardo. Reinforcement learning for RARS. Technical report, EECS Department, University of California, Berkeley, May 2003.

A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Systems*, 13:41–77, 2003.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume 13, pages 835–846, 1983.

J. Baxter and P. L. Bartlett. Direct gradient-based reinforcement learning. In *IEEE International Symposium on Circuits and Systems*, 2000.

R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

R. E. Bellman and S. E. Dreyfus. Functional approximations and dynamic programming. *Math Tables and Other Aides to Computation*, 13:247–251, 1959.

G. D. Benson, C.-W. Chu, Q. Huang, and S. G. Caglar. A comparison of MPICH Allgather algorithms on switched networks. In *10th European PVM/MPI Users' Group Conference (EuroPVM/MPI'03)*, 2003.

D. C. Bentivegna, C. G. Atkeson, and G. Cheng. Learning tasks from observation and practice. *Robotics and Autonomous Systems*, 47(2-3):163–169, 2004.

D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments.* Chapman and Hall, London, UK, 1985.

D. P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27:610–616, 1982.

D. P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, second edition, 2001.

D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall International, 1989.

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101:285–297, 1998.

A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

C. Boutilier, R. I. Brafman, and C. Geib. Prioritized goal decomposition of Markov decision processes: Towards a synthesis of classical and decision theoretic planning. In *International Joint Conference on Artificial Intelligence*, 1997.

C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1–2):49–107, 2000.

C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *International Joint Conference on Artificial Intelligence*, pages 690–700, 2001.

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

S. J. Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems*, volume 5, pages 295–302, 1993.

T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

A. Cassandra, M. L. Littman, and N. L. Zhang. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pages 54–61, 1997.

D. Chapman and L. P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

H.-T. Cheng. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, Vancouver, 1988.

L. Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 746–752, 1998.

D. Cliff and S. Ross. Adding temporary memory to zcs. *Adaptive Behavior*, 3: 101–150, 1994.

R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the pram model. In *Proceedings of the 1st annual ACM symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

J. W. Daniel. Splines and efficiency in dynamic programming. *Journal of Mathematical Analysis and Applications*, 54:402–407, 1976.

P. D. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 5, pages 271–278. Morgan Kaufmann, 1993.

T. Dean and R. Givan. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111, 1997.

T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

R. Dearden and C. Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89(1–2):219–283, 1997.

R. Dearden, N. Friedman, and S. J. Russell. Bayesian Q-learning. In *AAAI/IAAI*, pages 761–768, 1998.

R. Dearden, N. Friedman, and D. Andre. Model based bayesian exploration. In *Proc. of Fifteenth Conf. on Uncertainty in Artificial Intelligence*, pages 150–159. Morgan Kaufmann, 1999.

T. G. Dietterich. An overview of MAXQ hierarchical reinforcement learning. In *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA '2000)*, pages 26–44, 2000a.

T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000b.

T. G. Dietterich and N. S. Flann. Explanation-based learning and reinforcement learning: a unified view. *Machine Learning*, 28:169–210, 1997.

T. G. Dietterich and X. Wang. Batch value function approximation via support vectors. In *Advances in Neural Information Processing Systems*, volume 14, pages 1491–1498, Cambridge, MA, 2002. MIT Press.

K. Driessens and S. Džeroski. Integrating experimentation and guidance in relational reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning*, pages 115–122, 2002.

K. Driessens and J. Ramon. Relational instance based regression for relational reinforcement learning. In *Proceedings of the 20th International Conference on Machine Learning*, 2003.

S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1):7–52, 2001.

R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of ACM Symposium on Theory of Computing*, pages 114–118, 1978.

T. Gärtner, K. Driessens, and J. Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In *Proceedings of 13th International Conference on Inductive Logic Programming*, pages 146–163, 2003.

E. Gat. Three-layer architectures. In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 8. MIT/AAAI Press, 1997.

S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wikins. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

J. C. Gittins. *Multi-armed Bandit Allocation Indices.* Wiley, Chichester, NY, 1989.

G. J. Gordon. Reinforcement learning with function approximation converges to a region. In *Advances in Neural Information Processing Systems 13*, pages 1040–1046, 2001.

G. J. Gordon. Stable function approximation in dynamic programming. Technical Report CMU-CS-95-103, Carnegie Mellon University, 1995.

C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *International Joint Conference on Artificial Intelligence*, 2003.

P. Haddawy and M. Suwandi. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 266–271. AAAI Press, 1994.

G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. Distributed representations. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* MIT Press, 1986.

J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*, pages 216–227, 2000.

R. A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes.* Wiley, New York, 1971.

K. Hwang and Z. Xu. *Scalable Parallel Computing.* WCB/McGraw-Hill, 1998.

T. Jaakkola, M. I. Jordan, and S. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.

L. P. Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the 10th International Conference on Machine Learning*, pages 167–173, 1993a.

L. P. Kaelbling. *Learning in Embedded Systems.* MIT Press, Cambridge MA, 1993b.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

S. Kapetanakis and D. Kudenko. Reinforcement learning of coordination in heterogeneous cooperative multi-agent systems. In D. Kudenko, D. Kazakov, and E. Alonso, editors, *Adaptive Agents and Multi-Agent Systems II*. Springer, 2005.

M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49:209–232, 2002.

K. Kersting, M. Van Otterlo, and L. De Raedt. Bellman goes relational. In *Proceedings of the 21st International Conference on Machine Learning*, 2004.

F. Kirchner. Q-learning of complex behaviours on a six-legged walking machine. *Journal of Robotics and Autonomous Systems*, 25:256–263, 1998.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

H. Kitano, editor. *RoboCup-97: Robot Soccer World Cup I*. Springer Verlag, Berlin, 1998.

R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

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

N. Kushmerick, S. Hanks, and D. S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.

C.-S. Lin and H. Kim. CMAC-based adaptive critic self-learning control. *IEEE Transactions on Neural Networks*, 2:530–533, 1991.

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

L.-J. Lin and T. M. Mitchell. Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 271–280, 1992.

M. Littman. Memoryless policies: Theoretical limitations and practical results. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pages 297–305, 1994.

M. L. Littman. Friend-or-foe q-learning in general-sum games. In *Proceedings of the 18th International Conference on Machine Learning*, pages 322–328, 2001.

J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable markov decision processes. In *Proceedings of the 15th International Conference on Machine Learning*, pages 323–331, 1998.

D. Luce. *Individual Choice Behavior.* Wiley, New York, 1959.

O. Madani. *Complexity Results for Infinite-Horizon Markov Decision Processes.* PhD thesis, University of Washington, 2000.

O.-A. Maillard, R. Coulom, and P. Preux. Parallelization of the TD(λ) algorithm. In *European Workshop on Reinforcement Learning*, 2005.

S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the 21st International Conference on Machine Learning*, 2004.

A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, Rochester, NY, 1996.

J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proc. 18th International Conf. on Machine Learning*, pages 361–368, 2001.

N. Meuleau and P. Bourgine. Exploration of multi-state environments: Local measures and back-propagation of uncertainty. *Machine Learning*, 35(2):117–154, 1999.

N. Meuleau, M. Hauskrecht, K.-E. Kim, L. Peshkin, L. P. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov decision processes. In *AAAI/IAAI*, pages 165–172, 1998.

A. W. Moore. *Efficient Memory-based Learning for Robot Control.* PhD thesis, University of Cambridge, 1990.

A. W. Moore and C. G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. *Machine Learning*, 21 (3):199–233, 1995.

A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

A. W. Moore, L. C. Baird, and L. P. Kaelbling. Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs. In *International Joint Conference on Artificial Intelligence*, pages 1316–1323, 1999.

R. Munos and A. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49(2–3):291–323, 2002.

N. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

L. Nunes and E. Oliveira. Cooperative learning using advice exchange. In *Adaptive Agents and Multi-Agent Systems, LNCS vol. 2636*, 2003.

D. Ormoneit and S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49(2-3):161–178, 2002.

P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, CA, 1997.

R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1997.

E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

J. Peng and R. J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22:283–290, 1996.

T. J. Perkins and D. Precup. A convergent form of approximate policy iteration. In *Advances in Neural Information Processing Systems 15*, pages 1595–1602, 2002.

L. Peshkin. *Reinforcement Learning by Policy Search.* PhD thesis, Massachusetts Institute of Technology, 2001.

J. Pineau, G. J. Gordon, and S. B. Thrun. Point-based value iteration: An any-time algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence*, 2003.

L. D. Pyeatt and A. E. Howe. Decision tree function approximation in reinforcement learning. Technical Report TR-CS-98-112, Colorado State University, 1998.

D. V. Pynadath and M. Tambe. The communicative multiagent decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 1989.

C. E. Rasmussen and M. Kuss. Gaussian processes in reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 16, Cambridge, MA, 2004. MIT Press.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.

M. T. Rosenstein and A. G. Barto. Robot weightlifting by direct policy search. In *International Joint Conference on Artificial Intelligence*, pages 839–846, 2001.

N. Roy and G. J. Gordon. Exponential family PCA for belief compression in POMDPs. In *Advances in Neural Information Processing Systems*, volume 15, 2002.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* MIT Press, Cambridge, MA, 1986.

G. A. Rummery. *Problem Solving with Reinforcement Learning.* PhD thesis, University of Cambridge, Engineering Dept., 1995.

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. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

M. R. K. Ryan. *Hierarchical Reinforcement Learning: A Hybrid Approach.* PhD thesis, University of New South Wales, School of Computer Science and Engineering, Sydney, Australia, 2002a.

M. R. K. Ryan. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *Proceedings of the 19th International Conference on Machine Learning*, 2002b.

E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

C. Sammut. Automatic construction of reactive control systems using symbolic machine learning. *Knowledge Engineering Review*, 11:27–42, 1996.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

M. Sato, K. Abe, and H. Takeda. Learning control of finite Markov chains with an explicit trade-off between estimation and control. *IEEE Transactions on Systems, Man and Cybernetics*, 18:677–684, 1988.

C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), 1950.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

J. F. Shepanski and S. A. Macy. Manual training techniques of autonomous systems based on artificial neural networks. In *IEEE 1st International Conference on Neural Networks*, pages 697–704, 1987.

J. W. Sheppard and S. L. Salzberg. A teaching strategy for memory-based control. *Artificial Intelligence Review*, 11(1–5):343–370, 1997.

R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1080–1087, 1995.

Özgür. Şimşek and A. G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, 2004.

Özgür. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*, volume 9, pages 974–980. The MIT Press, 1996.

S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, 38 (3):287–308, 2000.

S. P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the AAAI*, pages 202–207, 1992.

W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the 17th International Conference on Machine Learning*, pages 903–910, 2000.

P. Stone. *Layered Learning in Multi-Agent Systems.* PhD thesis, Carnegie Mellon University, December 1998.

A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman. PAC model-free reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

M. Strens. A bayesian framework for reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

R. Sun and C. Simmons. Self segmentation of sequences. In *Proc of International Joint Conf. on Neural Networks.* IEEE Press, 1999.

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*, pages 216–224, 1990.

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.

R. S. Sutton, D. Precup, and S. P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211, 1999.

M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning (ICML-1993)*, pages 330–337, 1993.

B. Tanner and R. S. Sutton. TD(λ) Networks: Temporal-difference networks with eligibility traces. In *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–67, 1995.

S. B. Thrun. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches.* Van Nostrand Reinhold, 1992.

S. B. Thrun and K. Möller. Active exploration in dynamic environments. In *Advances in Neural Information Processing Systems*, pages 531–538, 1991.

S. B. Thrun and A. Schwartz. Finding structure in reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 7, pages 385–392. The MIT Press, 1995.

S. B. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, 1993.

J. N. Tsitsiklis and B. van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

J. N. Tsitsiklis and B. van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42:674–690, 1997.

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11): 1134–1142, 1984.

M. van Otterlo. A survey of reinforcement learning in relational domains. Technical Report TR-CTIT-05-31, University of Twente, The Netherlands, 2005.

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

C. J. C. H. Watkins and P. D. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

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.

M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

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

B. Wolfe, M. R. James, and S. Singh. Learning predictive state representations in dynamical systems without reset. In *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

J. Wyatt. *Exploration and Inference in Learning from Reinforcement.* PhD thesis, University of Edinburgh, 1997.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *International Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.