OPTIMAL CONTROL OF DYNAMIC SYSTEMS THROUGH THE REINFORCEMENT LEARNING OF TRANSITION POINTS

By

Kenneth M. Buckland

M.A.Sc. (Electrical Engineering) University of British Columbia

B.Sc. (Electrical Engineering) University of Calgary

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1994

© Kenneth M. Buckland, 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date Apr. 28/94

DE-6 (2/88)

Abstract

This work describes the theoretical development and practical application of transition point dynamic programming (TPDP). TPDP is a memory-based, reinforcement learning, direct dynamic programming approach to adaptive optimal control that can reduce the learning time and memory usage required for the control of continuous stochastic dynamic systems. TPDP does so by determining an ideal set of transition points (TPs) which specify, at various system states, only the control action *changes* necessary for optimal control. TPDP converges to an ideal TP set by using a variation of Q-learning to assess the merits of adding, swapping and removing TPs from states throughout the state space.

This work first presents how optimal control is achieved using dynamic programming, in particular Q-learning. It then presents the basic TPDP concept and proof that TPDP converges to an ideal set of TPs. After the formal presentation of TPDP, a Practical TPDP Algorithm will be described which facilitates the application of TPDP to practical problems. The compromises made to achieve good performance with the Practical TPDP Algorithm invalidate the TPDP convergence proofs, but near optimal control policies were nevertheless learned in the practical problems considered. These policies were learned very quickly compared to conventional Q-learning, and less memory was required during the learning process.

A neural network implementation of TPDP is also described, and the possibility of this neural network being a plausible model of biological movement control is speculated upon. Finally, the incorporation of TPDP into a complete hierarchical controller is discussed, and potential enhancements of TPDP are presented.

Table of Contents

List of Figures

Acknowledgements

I would like to thank the following people:

John Ip, my very good friend and kindred spirit, for his extensive help with the proofs in this thesis – and for our discussions in general.

My supervisor, Peter Lawrence, for providing many insights into reality.

The members of my thesis committee, David Lowe, Mabo Ito and Chris Ma, for their suggestions, and for taking the time to be involved in this work.

The taxpayers of Canada for their financial support.

All of those whose ideas and work have been of fundamental importance to my own and to this thesis, especially Marvin Minsky, Andrew Barto, James Albus and, always, Ayn Rand.

Dedication

This thesis, and all the best of my effort behind it,
is dedicated to my wonderful wife, Heather.

All in one, the best of women.
Full instinct of woman, self refined.
Lovely, healthy, firm, with thick hair and broad smile.
Unique air and look, deeply warming and exciting.
That quirky something.
A grand passion for adventure, worldwide and domestic.
New possessions she adores, choosing clothes and things for home.

The best of mothers, caring, careful, informed, and always a full heart.
Scientific in mind, practical, but a wide streak of creativity.
Art comes easily in boldness, flow and color.
And inventions that amuse, yet always have good purpose.
Trusting, reliable, loyal, relentlessly supporting.
A strong, natural sense of the decent and the right.
I love her fully.

Chapter 1

Introduction

1.1 Dynamic Programming Control

Dynamic programming (DP) methods can be used to optimally and adaptively control linear and non-linear systems (Barto *et al.*, 1991). In such applications, DP methods are used to determine the expected cost for each action that may be taken in each state. The action that has the lowest expected cost is then selected as the optimal action. Specifically, the determination of the optimal action for each state is formed as a Markov decision process to which DP methods can be applied (Watkins, 1989).

DP methods are effective in determining optimal actions in this manner. The problem with DP control is that, in a state space of any size, the learning time required to determine the optimal actions can be extensive (Barto *et al.*, 1991). Another problem is that the amount of memory required to store the DP computational elements can be enormous (Barto *et al.*, 1991).

Many different approaches have been taken to solve these two problems. A common approach is to use functional approximation to facilitate generalization of the expected costs being learned (Barto *et al.*, 1989, 1990*c*, Anderson, 1989*a*, 1989*b*, 1993, Werbos, 1990, Chinchuan *et al.*, 1990, Yee, 1992, White *et al.*, 1992, Thrun *et al.*, 1993). Such generalization reduces the learning time required to determine the optimal actions. Approaches have also been developed that partition the state space in ways which reduce the amount of memory required for representation (Moore, 1991, Chapman *et al.*, 1991).

Generally such partitioning approaches also reduce the learning time required because they group states together, facilitating the simultaneous (as opposed to individual) learning of optimal policies for all of the grouped states. Other approaches to reducing the learning time include ones where learning is focused on the regions of the state space where it will produce the best results (Moore *et al.*, 1993, Yee, 1992), and approaches where previous experiences are replayed (Lin, 1991*a*). Sections 2.4.3 through 2.4.6 will describe these various approaches in more detail.

1.2 The Transition Point Dynamic Programming Approach

This work describes another approach to reducing the learning time and memory usage required for DP control. Transition point dynamic programming (TPDP) operates by placing DP computational elements only at locations in the state space where *changes* to the action being specified are required for optimal control. The resulting reduction in memory usage decreases learning time because learning time is normally related to the number of DP computational elements (see Section 3.1.3). TPDP is thus a memory-based control approach (see Section 2.3.2) that can control a system using a minimal amount of memory[1].

TPDP employs reinforcement learning (see Section 2.3.3) to determine the proper placement of the DP computational elements that specify the control action changes. As a reinforcement learning approach TPDP can learn optimal control policies when the final outcome of a control sequence can be judged, but the desired intermediate actions are unknown. This capability typically requires longer learning times relative to other approaches (see Section 2.3.3). As a result, TPDP may not be the best control approach to take if the desired intermediate actions are known.

[1]Assuming a fixed, finely resolved state space (see Section 3.1.3).

TPDP operates as a direct DP controller that does not employ an explicit system model (see Section 2.2.1). TPDP is thus a form of Q-learning (Watkins, 1989).

TPDP is best suited to learning control policies for continuous stochastic dynamic systems – systems that have inertia. Some examples of systems to which TPDP control could be successfully applied are robot manipulators, chemical process control, hydraulic system control and flight control. The conditions under which TPDP operates best are described in Section 3.1.4.

1.3 The Format of this Work

TPDP will be described and demonstrated in this work in the following manner:

Chapter 2	DP will be described in detail, particularly direct DP in the form of "Q-Learning". The problems of DP will be discussed, as well as solutions to them that have been investigated by others.
Chapter 3	TPDP will be described in detail. The inspiration behind TPDP will be explained, and proof that TPDP converges on optimal control policies will be presented.
Chapter 4	A practical form of TPDP will be described, and an algorithmic implementation of that practical form presented.
Chapter 5	The effectiveness of TPDP will be demonstrated on two problems. In the process, many characteristics of TPDP will be illustrated.
Chapter 6	TPDP was developed in the context of neural network control. The relationship of TPDP to neural networks will be described, as well as a neural network implementation of TPDP.

Chapter 7	The incorporation of TPDP into a complete hierarchical controller will be described, and some attributes and capabilities of such a controller will be discussed.
Chapter 8	In conclusion, the overall attributes of TPDP will be described.

1.4 Brief Summary of Contributions

The development of TPDP is the main contribution of this work. The originality of TPDP is that it specifies only the control action changes necessary for the optimal control of continuous stochastic dynamic systems. As a result, when learning optimal control policies for such systems, TPDP requires less time and less memory than conventional Q-learning.

In the course of developing TPDP a number of additional contributions have been made:

1. The development of "generalization by copying" to facilitate generalization during TPDP learning, and to thereby increase the rate at which optimal actions are learned.

2. The development of a neural network implementation strategy for direct DP controllers – including TPDP controllers.

3. The development of approaches for the incorporation of TPDP into a complete hierarchical controller, including an investigation of the ways in which high level control knowledge can aid TPDP in learning optimal policies for the lowest level of control.

Chapter 2

Q-learning

This Chapter will provide the background information about dynamic programming (DP) and Q-learning required for the explanation of transition point dynamic programming (TPDP). It will also describe some of the solutions to the problems of DP, and the characteristics of Q-learning that are shared with TPDP.

2.1 Dynamic Programming Control

2.1.1 Control as a Markov Decision Process

The purpose of this Chapter is to present a control paradigm in which optimal control can be achieved by treating the control action determination in each state as a Markov decision process (Ross, 1983). When control action decisions are framed as Markov decision processes, optimal control policies can be determined using dynamic programming (DP) methods. Q-learning (Watkins, 1989, Watkins *et al.*, 1992), which is the standard direct form of DP (see Section 2.2.1), and which is the basis for transition point dynamic programming, will be described fully.

Consider a discrete-time stochastic dynamic system with a finite state set $S = 1, ..., n$ (Barto *et al.*, 1991). A *controller* sends *action*[1] specifications to this system, which act as inputs to the system and affect its state in some way. At time step t this system is in state $i \in S$, where the controller selects an action u from the *possible actions* $U(i)$

[1]Control actions will be referred to as "actions" throughout this work.

in that state; $u \in U(i)$. As a result of the application of action u, the system makes a transition to some state $j \in S$ in the next time step $(t+1)$. The probability of a transition occurring from state i to each state j as a result of u is the *state transition probability* $p_j(i,u)$; $0 \leq p_j(i,u) \leq 1$. If $p_j(i,u)$ is 0, a state transition is not possible from state i to state j as a result of action u. If $p_j(i,u)$ is 1, the state transition resulting from action u is a deterministic one that always moves the system from i to j. For all actions u taken in state i it is true that:

$$\sum_{j \in S} p_j(i,u) = 1 \tag{2.1}$$

As a result of the application of action u in state i, an *immediate cost* $c(i,u)$ is incurred. The state transition probabilities $p_j(i,u)$ and the immediate costs $c(i,u)$ are both functions of only the system state i and the action u taken in that state (Ross, 1983, Barto *et al.*, 1991).

The goal of a controller applied to such a system is to determine, for each state i, a *policy action* $\mu(i) \in U(i)$ that should always be taken to fulfill some *control task* (or *task*) required of the system. The control task could be to move the system along some trajectory, or to direct the system to some target state. The set of policy actions over the entire state space S constitutes the *policy*[2] μ; $\mu = [\mu(1), ..., \mu(n)]$.

Given a policy μ, and some initial state i, the sequence of states that the system can move through after leaving i forms a "Markov chain" (Narendra *et al.*, 1989) based on the state transition probabilities. Because this Markov chain is the result of the policy action selection, the control process is referred to as a "Markov decision process" (Ross, 1983, Barto *et al.*, 1991).

[2]Control policies will be referred to as "policies" throughout this work.

2.1.2 Markov Decision Processes and Optimal Control

Given a discrete-time stochastic dynamic system as described, an *optimality criteria* is required that can be used as a basis for an *optimal policy* determination. The optimality criteria used throughout this research was the "expected total infinite-horizon discounted cost". Other criteria, like total cost or average cost, can be used in Markov decision processes (Ross, 1983, Watkins, 1989), but DP controllers typically use the expected total infinite-horizon discounted cost. The expected total infinite-horizon discounted cost resulting if policy μ is applied from state i onward is the *evaluation function value*:

$$V_\mu(i) = E_\mu\left[\sum_{t=0}^{\infty} \gamma^t c(s_t, \mu(s_t)) | s_0 = i\right] \tag{2.2}$$

Where γ, $0 \leq \gamma \leq 1$, is the *discount factor* of future immediate costs $c(i,u)$, and s_t is the state at time step t. The notation "E_μ" indicates that $V_\mu(i)$ is based on the expected costs that will be incurred, given that policy μ is applied from state i onward.

The set of evaluation function values $V_\mu(i)$ for each state $i \in S$ over the entire state space S is known as the *evaluation function* and is denoted V_μ; $V_\mu = [V_\mu(1), ..., V_\mu(n)]$. The evaluation function indicates, for each state, the expected total infinite-horizon discounted cost that will result from the existing policy μ.

To facilitate optimal control, a policy μ must be determined that minimizes the evaluation function value $V_\mu(i)$ for each and every state i. The policy that achieves this is the optimal policy, denoted μ^*; $\mu^* = [\mu^*(1), ..., \mu^*(n)]$. Such a policy depends on γ, and it may not be unique – more than one action in each state may result in the same minimal expected total infinite-horizon discounted cost (Ross, 1983).

The *optimal evaluation function values* for each state i that result from the optimal policy μ^* are denoted $V_{\mu^*}(i)$. The set of these values over the entire state space S constitutes the *optimal evaluation function* V_{μ^*}; $V_{\mu^*} = [V_{\mu^*}(1), ..., V_{\mu^*}(n)]$.

If an optimal policy is determined that minimizes $V_\mu(i)$ for each state i, and $V_\mu(i)$ is composed of future immediate costs as indicated in Equation 2.2, then the immediate costs (in conjunction with the discount rate γ) are the basis upon which the optimal policy is determined. As a result, the control task (see Section 2.1.1) that the policy leads the system to perform is inherently defined by the setting of the immediate costs. Generally, lower immediate costs are associated with target states, or with states along desired trajectories, resulting in policies that direct the system towards those states.

2.1.3 The Optimality Equation

A method of determining V_{μ^*} is required to determine an optimal policy μ^*, and to thereby facilitate optimal control. As will be explained, once V_{μ^*} is known, the optimal policy μ^* can readily be determined.

Q-values (Watkins' notation, 1989) are defined as[3]:

$$Q(i,u) = c(i,u) + \gamma \sum_{j \in S} p_j(i,u) V_\mu(j) \tag{2.3}$$

The Q-value $Q(i,u)$ is the expected total infinite-horizon discounted cost if action u is taken in state i and policy μ is followed in all states thereafter (Barto *et al.*, 1989).

To facilitate optimal control, the policy action $\mu(i)$ for each state i should be the action that has the lowest Q-value for that state. The policy action $\mu(i)$ should therefore be an action which satisfies:

$$Q(i,\mu(i)) = \min_{u \in U(i)} Q(i,u) \tag{2.4}$$

If the policy actions determined in this manner are always taken, the evaluation function value $V_\mu(i)$ for each state i will be the Q-value of the policy action $\mu(i)$:

[3]A more precise notation for Q-values would be "$Q_{V_\mu}(i,u)$", but for the purpose of clarity "$Q(i,u)$" will be used throughout this work.

$$
\begin{aligned}
V_\mu(i) &= Q(i, \mu(i)) && (2.5)\\
&= \min_{u \in U(i)} Q(i, u) && (2.6)\\
&= \min_{u \in U(i)} \left[c(i, u) + \gamma \sum_{j \in S} p_j(i, u) V_\mu(j) \right] && (2.7)
\end{aligned}
$$

The evaluation function V_μ resulting from the determination of policy actions in this manner will actually be an optimal evaluation function (Ross, 1983). This is true because, if Equation 2.7 is recursively expanded out along the Markov chain extending from state i, $V_\mu(i)$ will consist entirely of sums of minimum costs. As a result of this, the optimal evaluation function is defined by:

$$
\begin{aligned}
V_{\mu^*}(i) &= \min_{u \in U(i)} Q^*(i, u) && (2.8)\\
&= \min_{u \in U(i)} \left[c(i, u) + \gamma \sum_{j \in S} p_j(i, u) V_{\mu^*}(j) \right] && (2.9)
\end{aligned}
$$

Equation 2.9 is one form of the Bellman Optimality Equation" (Bellman, 1957). The goal of DP, in all its variations, is to solve Equation 2.9 for each state i to determine V_{μ^*} (Barto *et al.*, 1989). Once V_{μ^*} has been determined, the *optimal Q-values*[4] $Q^*(i, u)$ can be determined using Equation 2.3, and the *optimal action* $\mu^*(i) \in U(i)$ can be determined for each state i by finding those actions which satisfy Equation 2.4. The optimal policy μ^* is thereby determined.

2.1.4 Solving the Optimality Equation

The goal of DP is to solve Equation 2.9, the Bellman Optimality Equation, for each state i to determine the optimal evaluation function V_{μ^*} and thereby determine the

[4]A more precise notation for optimal Q-values would be "$Q_{V_{\mu^*}}(i, u)$", but for the purpose of clarity "$Q^*(i, u)$" will be used throughout this work.

optimal policy μ^*. The standard approach is that of ***value iteration*** (Barto *et al.*, 1991). Value iteration is a successive approximation procedure where an evaluation function approximation V_k is updated so that it converges on V_{μ^*}.

Following Barto *et al.* (1991), V_k is defined as the estimate of V_{μ^*} at stage k ($k = 0, 1, ...$) of the approximation computation. Where $V_k(i)$ is the approximate optimal evaluation function value of state i at stage k, the successive approximation procedure is defined as follows:

$$V_{k+1}(i) = \min_{u \in U(i)} Q_{V_k}(i,u) \tag{2.10}$$

$$= \min_{u \in U(i)} \left[c(i,u) + \gamma \sum_{j \in S} p_j(i,u) V_k(j) \right] \tag{2.11}$$

Updating the evaluation function in this manner is referred to as ***backing-up*** the costs because, for each state i, $V_k(i)$ is updated with the evaluation function value approximations $V_k(j)$ of the states j that the system may make a transition to after taking action $u \in U(i)$ in state i (Barto *et al.*, 1991).

2.2 Direct DP Control

2.2.1 Direct and Indirect DP

The description of DP so far has focused on systems where an explicit model of the system response is known or has been learned using system identification (Ogata, 1970). This model is defined by the state transition probability values $p_j(i,u)$, which indicate the probability of a transition occurring from state i to state j if action u is employed in state i. Controllers which use such explicit models are ***indirect controllers*** (Sutton, 1991, Sutton *et al.*, 1992, Barto *et al.*, 1991, Narendra, 1992). Explicit models can be constructed using methods more compact than simple probability tables, but for the sake of brevity it will be assumed throughout this work that they are not.

Direct controllers do not use explicit models of the system response. Instead they directly determine how to control the system by making observations of how the system responds to the various actions in the different system states (Sutton *et al.*, 1992, Barto *et al.*, 1991). Direct DP controllers function by repeatedly observing the actual costs that are incurred when an action $u \in U(i)$ is employed in state i (Barto *et al.*, 1989).

2.2.2 Direct DP in the form of Q-learning

A standard way to implement direct DP controllers is using *Q-learning* (Watkins, 1989, Watkins *et al.*, 1992). Q-learning operates as follows. In Equation 2.8, the optimal evaluation function value $V_{\mu^*}(i)$ is determined in each state i by equating it to the lowest of the optimal Q-values $Q^*(i,u)$ in that state. Direct DP control is thus possible if the optimal Q-values can be determined without the use of an explicit system model. This is done by making successive approximations of the optimal Q-values. Where $Q_t(i,u)$ is the approximate Q-value of taking action u in state i at time step t, the successive approximation update is defined as follows[5]:

$$Q_{t+1}(i,u) = \begin{cases} [1-\alpha_t(i,u)]\,Q_t(i,u) + \alpha_t(i,u)\,[c(i,u) + \gamma V_t(s_{t+1})] & \text{if } i = s_t, u = u_t \\ Q_t(i,u) & \text{otherwise} \end{cases} \tag{2.12}$$

Where $\alpha_t(i,u)$, $0 < \alpha_t(i,u) < 1$, is the *update rate*, and where $V_t(s_{t+1})$ is the approximation of the evaluation function value of the state s_{t+1} that the system (actually) makes a transition to after action u is applied in the current state s_t. Using successive approximations of the Q-values, successive approximations of the optimal evaluation function values $V_{\mu^*}(i)$ for each state i are determined with:

$$V_{t+1}(i) = \min_{u \in U(i)} Q_{t+1}(i,u) \tag{2.13}$$

[5]Because direct observation of the system response is being made, each time step t corresponds to an approximation computation stage k.

Because the system is stochastic, and Q-learning does not employ an explicit system model, s_{t+1} cannot be known or predicted until the state transition actually occurs. Q-value updating in Q-learning thus depends on observation of the actual response of the system in terms of transitions made from one state to the next, as well as the costs incurred during those transitions.

By using direct experience of the response of a system to update the Q-values of a direct DP controller (Equation 2.12), and then by using these Q-values to update the approximation of the optimal evaluation function values $V_t(i)$ for each state i (Equation 2.13), Q-learning can converge to the optimal evaluation function V_{μ^*} and thereby determine an optimal policy μ^* (Barto *et al.*, 1991). This is done without the formation of any explicit models of the system response. Information about the stochastic behavior of the system is contained implicitly in the Q-values as part of the costs that are experienced as a result of the stochastic state transitions.

2.2.3 Convergence of Q-learning

Watkins (1989, *et al.*, 1992) has proven that Q-learning will converge to the optimal Q-values necessary for optimal control under the following conditions:

1. Every state i and action $u \in U(i)$ combination must be attempted an infinite number of times as learning progresses.

2. The immediate costs $c(i, u)$ must be bounded for each action $u \in U(i)$ in state i.

3. It must be true that $0 \leq \gamma \leq 1$.

4. It must be true, for each action $u \in U(i)$ in each state i at time step t, that $0 < \alpha_t(i, u) < 1$, and that $\alpha_t(i, u) \to 0$ as $t \to \infty$.

5. Where n_k is the time step in which a Q-value $Q_{n_k}(i,u)$ is updated ($i = s_t$, $u = u_t$) for the k^{th} time, it must be true that, for each action $u \in U(i)$ in state i:

$$\sum_{k=1}^{\infty} \alpha_{n_k}(i,u) = \infty, \qquad \sum_{k=1}^{\infty} \left[\alpha_{n_k}(i,u)\right]^2 < \infty \tag{2.14}$$

2.2.4 Exploration in Q-learning

Because Q-learning relies on actual experience of state transitions and immediate costs to determine optimal Q-values, the possible policies must be thoroughly explored to ensure that optimal Q-values are converged to. To that end, it is required that Q-learning controllers have a non-zero probability of visiting every state $i \in S$, and employing every possible action $u \in U(i)$ in that state for all time (Watkins *et al.*, 1992). This is the first Q-learning convergence condition given in Section 2.2.3.

The exploration that is performed in Q-learning must result in a thorough search of the possible policies. While this exploration can be extensive, it is not strictly exhaustive because every possible *sequence* of action specifications does not have to be attempted (Barto *et al.*, 1989, 1993). The backing-up of the evaluation function values (see Section 2.1.4) facilitates thorough exploration using only transitions from one state to the next.

In Q-learning, when the evaluation function approximation V_t is close to being optimal, it is not normally worthwhile to continue exploring state and action combinations whose Q-values are large relative to other Q-values for the same state. Such state and action combinations are unlikely to be part of the optimal policy and the probability of exploring them should be reduced as learning progresses – although not to 0. Doing so is particularly desirable if reasonable, near optimal, control of the system is valued during the continuing learning process. Determining how frequently such state and action combinations should be explored is part of a fundamental problem in optimal adaptive control (Thrun, 1992, Barto *et al.*, 1991). This problem concerns balancing the exploration needed to ensure that optimal control is achieved with the desire to provide reasonable control (Michie *et al.*, 1968). In the context of Q-learning, many different exploration strategies have been investigated (Watkins, 1989, Sutton, 1990, Barto *et al.*, 1990*a*, 1990*c*, 1991, Kaelbling, 1990).

2.2.5 One-step Q-learning

The form of Q-learning presented thus far is "one-step Q-learning" (Watkins, 1989). It is the simplest form of Q-learning, where Q-values are updated immediately after each action is applied using the evaluation function value $V_\mu(i)$ of the next state encountered. More complex forms of Q-learning are described by Watkins (1989) and mentioned briefly in Section 2.3.6.

The simple one-step form of Q-learning, as well as discretized memory-based representation of the state space (see Section 2.3.2), will be the approach to Q-learning assumed throughout this work. The reason for this is that the transition point dynamic programming (TPDP) controller that will be described makes use of Q-learning in a complicated way. To keep the analysis of TPDP as clear as possible, it is therefore best to use the simplest form of Q-learning.

2.3 The Characteristics of Q-learning

2.3.1 Q-learning is Direct DP Control

As explained in Section 2.2.2, Q-learning is direct DP control. Q-learning operates without the use of explicit system models. This leads to Q-learning having a number of advantages and disadvantages when compared to indirect DP controllers. These will be described in this Section.

Q-learning Requires Less Memory

One significant advantage of Q-learning over indirect DP control is that, because the system model is implicitly contained in the Q-values, an explicit and separate representation of the model is not necessary (Watkins, 1989). This can greatly reduce the amount of memory required for a DP controller (Barto *et al.*, 1990*b*)[6]. In stochastic control applications, for each possible action $u \in U(i)$ in each state i, indirect DP controllers must store a state transition probability $p_j(i, u)$ for each state j that may be reached as a result of action u. If any action made in any state has some probability of causing a transition to every other state, the number of state transition probability values that must be stored is (Watkins, 1989):

$$\text{worst-case number of state transition probability values} = \sum_{i \in S} \sum_{u \in U(i)} |S|$$

Where $|S|$ is the number of states in the state space S. In addition to these state transition probabilities, indirect DP controllers must store an evaluation function value for each state i.

Q-learning must store a Q-value for each possible action $u \in U(i)$ in each state i:

$$\text{number of Q-values} = \sum_{i \in S} \sum_{u \in U(i)} 1$$

Q-learning must also store an evaluation function value for each state i. The worst-case difference then between the memory requirements of Q-learning and of indirect DP controllers consists of the difference between the worst-case number of state transition probabilities that must be stored for indirect DP control and the number of Q-values that must be stored for Q-learning. The former is $|S|$ times larger than the latter.

In practice it is unlikely that the worst-case number of indirect DP state transition probabilities need to be stored. Some state transitions may be impossible, or of such

[6]Under the assumption stated in Section 2.2.1 that explicit models are always constructed using probability tables.

low likelihood that they can be ignored. Even so, the best case memory requirement for indirect DP is when only one state transition is possible for each action $u \in U(i)$ in each state i. In this deterministic case the memory requirement for indirect DP control is only *as low as* that of Q-learning. Furthermore, considerable computational effort might be required to determine just which state transition probabilities are important for an indirect DP controller to store (Watkins, 1989).

Q-learning is Computationally Simpler

The implicit models in Q-learning controllers are updated continuously as state transitions, and their associated costs, are observed. Indirect DP controllers, with their explicit models, do not have this feature. As a result, computational steps must be taken in such controllers to ensure that the model is kept up to date (Barto *et al.*, 1990*b*, 1991). "This computation is inherently complex, making adaptive methods in which the optimal controls are estimated directly[7] more attractive" (Sutton *et al.*, 1992).

The Task-Specificity of Implicit System Models

While Q-learning controllers require less memory than indirect DP controllers because their system model is contained implicitly in their Q-values, there is a disadvantage in this approach to system modeling. The disadvantage is that, because an explicit model is not available, that model cannot be used in all the different control tasks that a system may be required to perform.

Most DP controllers, both indirect and direct (Q-learning), can only perform one control task at a time. If more than one control task is required of a system, control must be shifted from one DP controller to another. If indirect DP control is being applied to a system, the same system model can be used by all of the DP controllers fulfilling

[7]Of which Q-learning is one.

the various control tasks. In contrast, the system model in Q-learning cannot be used in different control tasks because that model is implicitly stored in the Q-values and the Q-values are specific to each control task.

The value of being able to develop a general purpose system model can be questioned however if the extensive memory requirements of explicit models are considered. Section 7.2.3 will present more analysis of the modeling issue.

Off-Line Updating and Q-learning

Another disadvantage of Q-learning is that the evaluation functions of controllers employing Q-learning cannot be updated *off-line* (Barto *et al.*, 1991). That is, they cannot be updated when the controller is not controlling the system. As described by Barto *et al.* (1991), off-line updating is the backing-up of costs that can be done when indirect DP control is being employed. Indirect DP controllers have explicit system models containing state transition probabilities. If these transition probabilities are accurate, they can be used at any time to facilitate the backing-up of costs (see Section 2.1.4). It is not necessary to make actual observations of system state transitions to perform back-ups as state transitions can effectively be simulated. By reducing the amount of actual experience required to learn optimal policies, off-line back-ups can greatly reduce the time required to learn those policies.

Because the system model in Q-learning controllers is contained implicitly in the Q-values, distinct state transition probabilities that could be used for off-line back-ups are not available. As a result, off-line updating cannot be performed. Updating in Q-learning can only occur when the system is being controlled and actual state transitions are being observed – unless of course an explicit system model is learned in addition to the implicit model of Q-learning (Sutton, 1991).

2.3.2 Q-learning Can Be Implemented as Memory-Based Control

Multi-dimensional tables that associate an action $u \in U(i)$ with each state i can be used to control systems regardless of any non-linearities in those systems (Atkeson, 1989, 1991). Such controllers, whose table dimensions correspond to the state dimensions of the system being controlled, are called *memory-based controllers.* They represent states in a completely localized way (Atkeson, 1989).

Q-learning controllers can be implemented as memory-based controllers that operate by associating a table entry with each state $i \in S$. In each entry they store Q-values $Q(i, u)$ that directly reflect the costs that will be incurred when each action $u \in U(i)$ is applied in state i. Q-learning is well suited to such implementation.

Like other memory-based DP controllers (Moore, 1991), memory-based Q-learning controllers typically handle continuous state spaces by discretizing them (Baker *et al.*, 1992). This discretization facilitates the representation of the states in a tabular format (Barto *et al.*, 1991).

The table entries associated with each state in a DP controller will be referred to as *DP elements.*

2.3.3 Q-learning is Reinforcement Learning

Q-learning is inherently reinforcement learning (Sutton *et al.*, 1992). Q-learning operates by determining Q-values, and from them evaluation function values (see Section 2.2.2). Referring to Equation 2.2, the costs in this Equation that are discounted and summed are the immediate costs $c(i, u)$. These immediate costs constitute a scalar reinforcement signal that the controller receives as a result of the actions that it specifies the system should take. Aside from observable changes in the state of the system, this is all the information which Q-learning controllers receive.

In contrast to reinforcement learning approaches like Q-learning, supervised learning approaches (error propagation methods for example, Rumelhart *et al.*, 1986) receive complete information about what the desired output (in the case of controllers this is the optimal action) is at any point in time (Williams, 1986, 1987*a*). When such full information is available it can result in faster learning than the scalar reinforcement signal of reinforcement learning (Williams, 1987*b*). If desired output information is not available however, supervised learning approaches cannot be utilized (Anderson, 1989*a*). As a result, "reinforcement learning is clearly more generally applicable than supervised learning" (Williams, 1987*b*).

An example of a learning application where reinforcement learning is necessary is that of a novice snowboarder. A person in such a situation may not know exactly what movements he should make to stay erect, but he certainly knows when he has fallen. In this case the discomfort of the fall constitutes a negative reinforcement signal.

The reinforcement learning capability of Q-learning facilitates the learning of optimal policies without any of the optimal actions being known in advance. Such learning is not possible with supervised learning approaches.

Another potential advantage of reinforcement learning approaches over supervised learning approaches occurs when many outputs are equally desirable in response to a given input (in the case of controllers this is when more than one action is optimal in a given state). In such a situation reinforcement learning approaches can learn any of the appropriate outputs, while supervised learning approaches must learn one arbitrarily chosen desired output (Williams, 1987*a*, 1987*b*). The increased flexibility of reinforcement learning approaches in such situations has the potential to result in faster learning.

Finally, an important difference between reinforcement learning and supervised learning approaches is that supervised approaches typically follow learning gradients to determine their input to output mappings (Rumelhart *et al.*, 1986). Reinforcement learning approaches do not, so they must employ mechanisms which ensure full exploration of the mapping possibilities (Williams, 1987*a*)[8]. An advantage of *not* following learning gradients is that (most) reinforcement learning approaches will not prematurely settle on local minima (or maxima) – a problem which confronts the gradient-following supervised learning approaches (Rumelhart *et al.*, 1986).

As described by Barto (1992), Q-learning is an Adaptive Critic Method. Adaptive Critic Methods are reinforcement learning approaches that include a mechanism for anticipating future reinforcements. Some common citations made to sketch out the course of development of Adaptive Critic Methods include: Samuel (1959), Michie *et al.* (1968), Widrow *et al.* (1973), Barto *et al.* (1983), Sutton (1988) and Watkins (1989).

2.3.4 Q-learning Addresses the Credit Assignment Problem

The basic idea of reinforcement learning (see Section 2.3.3) is to increase or decrease the likelihood of a controller specifying a given action based on whether positive or negative reinforcement normally results from that action (Mendel, 1973). The difficulty lies in determining how much each action, of all those taken in the time proceeding the reinforcement, should be credited (positively or negatively) with the outcome. This is the credit assignment problem (Minsky, 1985), and it has a structural and a temporal component (Williams, 1987*b*).

Q-learning addresses both components of the credit assignment problem by backing-up evaluation function values $V_\mu(i)$. This backing-up distributes, in reverse, credit for incurred costs, and the distribution is simultaneously structural and temporal. The state transition probabilities $p_j(i,u)$ and the discount factor γ determine the amount of credit assigned to each state for each incurred cost (see Sections 2.1.1 to 2.1.4).

8Exploration issues are discussed in Section 2.2.4.

2.3.5 Q-learning is Adaptive Optimal Control

Q-learning is an adaptive optimal control approach (Sutton *et al.*, 1992, Barto *et al.*, 1991). That Q-learning provides optimal control has been discussed throughout this Chapter (also see White *et al.*, 1992). Q-learning is also adaptive in providing optimal control because the model that Q-learning makes of a system is implicitly contained in the Q-values (Barto *et al.*, 1991). As the system changes, those changes will be reflected in the updating of the Q-values. At any instant in time, a controller using Q-learning will be converging on the optimal policy for the system that exists at that time.

Indirect DP controllers do not provide adaptive optimal control as readily as Q-learning controllers. To adaptively follow system changes such controllers must continually modify their explicit system model. Depending on the exact approach taken, this may require complete recalculation of both the model and the optimal policy based on that model (Barto *et al.*, 1991). This can be extremely computationally expensive if the system changes with any regularity.

2.3.6 Q-learning and Temporal Differences

Q-learning is a form of temporal difference learning (Sutton, 1988, Watkins, 1989, Dayan, 1991). Temporal difference learning is notated as TD(λ) (Sutton, 1988). To learn expected total infinite-horizon discounted costs, TD(λ) updating of Q-values is performed as follows (derived from Watkins, 1989):

$$\begin{aligned} Q_{t+1}(s_t,u_t) &= [1-\alpha_t(s_t,u_t)]\,Q_t(s_t,u_t) + \\ &\quad \alpha_t(s_t,u_t)\left[\sum_{k=0}^{\infty}\gamma^k\lambda^k\left(c(s_{t+k},u_{t+k})+\gamma(1-\lambda)V_{t+k}(s_{t+k+1})\right)\right] \end{aligned} \tag{2.15}$$

Where λ, $0 \le \lambda \le 1$, is the weighting of future experiences.

In the Q-learning methods described in this work, λ will be set to 0, resulting in TD(0) (Sutton, 1988). Setting λ to 0 in Equation 2.15 converts it into the familiar form of Equation 2.12. TD(0) updates the values being learned using only the information obtained in the next time step, $t+1$. In one application Sutton (1988) found that TD(λ) worked best at λ values around .3, but as explained by Watkins (1989) using anything but TD(0) to update Q-values requires considerably more computational effort.

Watkins (1989) also pointed out that setting λ to 1 in Equation 2.15 results in Q-value updating based simply on the sum of the actual infinite-horizon discounted costs experienced (see Equation 2.2). TD(1) learning then does not use any evaluation function values $V_\mu(i)$ in the Q-value updating. As the evaluation function values are only approximations of the optimal evaluation function until convergence on that function is complete (see Section 2.1.4), TD(1) avoids the use of approximate evaluation function values.

2.3.7 Q-learning and Noise

In a system being controlled with a Q-learning controller, where actions $u \in U(i)$ are specified by the controller for each state i that the system enters, there are four ways in which noise can affect control of the system:

1. Noise can perturb the controller specification of the action $u \in U(i)$ applied to the system.

2. Noise can perturb the response of the system to the action $u \in U(i)$.

3. Noise can perturb the immediate costs $c(i,u)$ returned to the controller as a result of the action u that it took in state i.

4. Noise can perturb controller observation of the system state so that the controller inaccurately assesses which state the system is in.

The first three potential effects of noise are intrinsic to the control of discrete-time stochastic dynamic systems. These noise components are what make the state transitions stochastic (Barto *et al.*, 1989). As such, these effects of noise are inherently handled by Q-learning. As Q-learning proceeds to learn optimal Q-values, these effects of noise are experienced and amalgamated into the expected costs that the Q-values reflect. As long as a system controlled by Q-learning is controllable to the extent desired (where controllability is in part affected by the first two types of noise), Q-learning will converge on a policy that provides optimal control in those noise conditions.

The fourth effect of noise is not addressed by Q-learning. Such noise affects the observability of the system, not its stochastic response, and it cannot be incorporated into the implicit model of the system (Kaelbling, 1990). All controllers are vulnerable to this type of noise however, and it is fairly safe to say that, as an optimal control approach, Q-learning can handle it as well as any.

More generally, "dynamic programming is the *only* exact and efficient method available to solve the problem of [cost minimization] over time in the general case where noise and nonlinearity may be present" (Werbos, 1990).

2.4 Practical DP Control

2.4.1 The Curse of Dimensionality

In order to facilitate DP control, memory must be allocated to store the DP elements necessary for every state $i \in S$. These memory entries constitute the table described in Section 2.3.2. Normally the table structure is defined by the state space dimensions and the resolution of each of those dimensions. As the number of dimensions increases, the total memory requirement increases exponentially (Barto *et al.*, 1991, Moore, 1991).

Bellman (1957) called this the "curse of dimensionality". This curse haunts DP controllers when they are applied to systems that have anything but the most restricted of state spaces.

As the number of states and their associated DP elements grow, the curse of dimensionality also increases the learning time required for DP controllers to converge to optimal policies (Moore, 1991, Yee, 1992). This is because DP controllers must perform a certain amount of computation for each state.

As described, the curse of dimensionality results in extensive memory usage and protracted learning times. To make DP control practical these problems must be addressed. Sections 2.4.2 to 2.4.6 describe work done by others to that end.

2.4.2 Associative Content Addressable Memories (ACAMs)

Associative content addressable memories (ACAMs) can be used to significantly reduce the memory requirements of memory-based controllers (Atkeson, 1989), including DP controllers. ACAMs are not addressed in the manner of standard memories, but are "content-addressable in that part of the contents of the memory are used to select the rest of the memory" (Atkeson *et al.*, 1988). In the case of DP controllers, an ACAM would use the state information to select the memory entry containing the DP element information associated with that state (see Section 2.3.2).

ACAMs are able to reduce the memory requirement of memory-based controllers, including DP controllers, because systems fulfilling specific control tasks may never enter significant portions of the fully expanded state space defined by their state space dimensions (see Section 2.3.2). It may even be *impossible* for the system to reach some of the defined states. As a result, if ACAMs are employed to contain the DP elements, memory entries need only be allocated for those states that actually are encountered as the control task is fulfilled (Atkeson, 1989). Such allocation may result in far less memory being used than would be required if a full tabular memory is employed (see Section 2.3.2).

ACAMs can be implemented using specialized hardware; although such hardware is rarely used. ACAMs can also be simulated using massively parallel computers like the Connection Machine (Atkeson, 1989, Hillis, 1985). Hashing techniques can be employed to simulate ACAMs (Standish, 1980). Korf (1990) applied hashing in a heuristic search algorithm that operated basically like an indirect DP controller. Neural networks can also be designed to operate as ACAMs, as will be described in Sections 6.1.2 and 6.1.3.

2.4.3 Amalgamation of States

Within a state space S there may be groups of neighboring states that are all relatively unimportant in terms of the fulfillment of the required control task (Moore, 1991). That is, these states may be ones that the system is not likely to enter during the normal course of task fulfillment and, as a result, the evaluation function values $V_\mu(i)$ associated with them need not be exact. If inaccuracies in the evaluation function are tolerable, it is possible to amalgamate groups of such states together together and treat each group like a single state for the purposes of DP control. Such amalgamation reduces the memory required for DP control (Chapman *et al.*, 1991, Moore, 1991) and increases the rate of learning (Singh, 1992*a*). The rate of learning is increased because it can progress in all of the amalgamated states simultaneously instead of just one state at a time (Chapman *et al.*, 1991).

There are two difficulties to such an approach (Buckland *et al.*, 1993). One is determining which states are of limited enough importance to be included in an amalgamation without optimal control being seriously affected. The other difficulty is ensuring that the amalgamated states each have similar enough evaluation function values $V_\mu(i)$ that the sharing of the same value across all of them will not distort the overall evaluation function V_μ (Anderson, 1989*a*) in a way that significantly perturbs the learning of the optimal policy throughout the entire state space (Buckland *et al.*, 1993, Moore, 1991). Determining which states can prudently be amalgamated, and then determining appropriate amalgamation groupings requires computational effort beyond that necessary to learn optimal policies. The extent of that extra effort must be weighed against the benefits of amalgamation.

Moore (1991) used "trie" data structures (Omohundro, 1987) to facilitate variable resolution in the evaluation function used by an indirect DP controller. The resolution of the evaluation function was made finer in states that were projected as being closer to the optimal trajectory of the system concerned. Such variable resolution effectively amalgamates the states that are coarsely resolved.

Chapman *et al.* (1991) recursively split the state space of a system that was represented by many binary dimensions. This splitting resulted in a binary tree of Q-values. Each node in this tree which was not split down to an irreducible size was effectively an amalgamation of the unsplit states it contained. The splitting decisions were based on statistics gathered regarding differences in the reinforcements received within unsplit states.

Mahadevan *et al.* (1992) investigated two techniques that involved state amalgamation. In one, all of the Q-values within a fixed Hamming distance of the current state were updated, resulting in an amalgamation effect. In the other, statistical clustering was used to flexibly group states in close proximity to each other that had similar Q-values.

Buckland *et al.* (1993) investigated a number of Q-learning approaches that used statistics gathered regarding differences in reinforcements received. These statistics facilitated both the splitting and combining of state amalgamations. These investigations were of limited success because the computational effort required to make the amalgamations was often extensive and it disturbed the computations being made to learn the optimal actions (Buckland *et al.*, 1993).

2.4.4 Approximation of Evaluation Functions

Up to this point the evaluation function V_μ of DP controllers has been described as consisting of separate evaluation function values $V_\mu(i)$ for each state i. The curse of dimensionality makes such discrete representation of the evaluation function impractical for state spaces of any size however (see Section 2.4.1), so the evaluation function is normally parametrically approximated in some way (Barto *et al.*, 1989, 1990*c*, Anderson, 1989*a*, 1989*b*, 1993, Werbos, 1990, Chinchuan *et al.*, 1990, Yee, 1992, White *et al.*, 1992, Thrun *et al.*, 1993). In addition to reducing memory usage, such parametric approximation facilitates generalization between the states (Barto *et al.*, 1989), thereby increasing the rate of learning.

Many different approaches to evaluation function parametric approximation have been taken. Barto *et al.* (1989) have formulated how to generally apply TD methods (Sutton, 1988) to evaluation function parametric approximation. This formulation includes approximation with supervised learning neural networks (Barto *et al.*, 1989). Werbos (1990) has done work utilizing such supervised learning in DP controllers. Watkins (1989) and Tham *et al.* (1993) used the CMAC neural model (Albus, 1975*a*, 1975*b*) for evaluation function and Q-value parametric approximation.

The evaluation function can also be approximated in other ways. Yee (1992) has used hierarchical binary trees and hierarchical neural networks to that end.

The main drawback of evaluation function approximation approaches is that the approximating mechanisms themselves typically require extensive computational effort to develop a reasonably accurate approximation. Further, evaluation function approximation often invalidates any proofs that have been made of the convergence of a DP controller to an optimal policy (Barto *et al.*, 1991).

Evaluation function approximation is made even more difficult in Q-learning controllers because of the fact that Q-values are a function of action as well as state, and an additional action dimension must be included in any approximation that represents Q-values.

2.4.5 Prioritized Exploration

Another way to increase the rate of learning in DP controllers is to explore the state space in ways that concentrate the learning effort on those states where learning will produce the best results. As exploration is essential to Q-learning, many researchers have developed highly effective exploration strategies (see Section 2.2.4). In addition to this work however, much work has been done that focuses on exploration in DP controllers in general.

Sutton (1990), Kaelbling (1990) and Yee (1992) have done work on associating accuracy information with the costs being backed-up to indicate which back-ups are the most likely to enhance learning. Along similar lines, Peng *et al.* (1992) and Moore *et al.* (1993) have used the size of the change that occurs with each cost update to prioritize the future exploration of the state that update was associated with. The largest changes in cost were given the highest priority.

2.4.6 Replaying Experiences

If real-time computation constraints make it possible, the rate of learning in DP controllers can be increased by replaying past experiences and making new updates based on those experiences (Lin, 1991*a*, 1991*b*). This approach requires that the details of past experiences be stored somehow.

Chapter 3

Transition Point Dynamic Programming (TPDP)

This Chapter will present a full description of transition point dynamic programming (TPDP), including proof of its convergence to optimal control policies. The description will rely on the explanation of DP and Q-learning presented in Chapter 2, but this Chapter will not be a direct extension of Chapter 2. Instead the background information of Chapter 2 will be utilized as required to explain TPDP.

3.1 General Description of TPDP

3.1.1 Inspiration

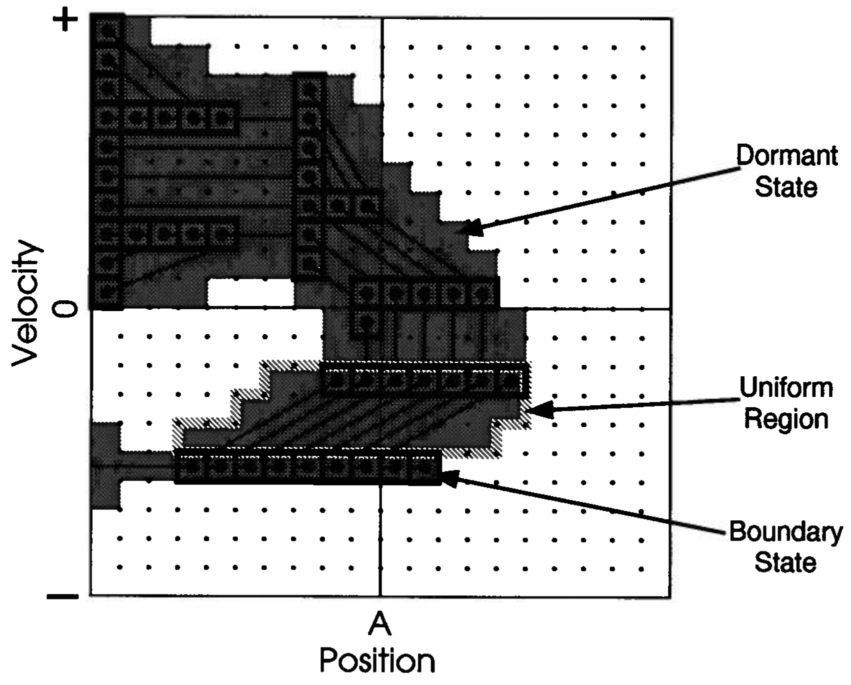
Transition point dynamic programming (TPDP) was developed as a solution to the extensive memory use and protracted learning time that results from the "curse of dimensionality" in DP controllers (see Section 2.4.1). The TPDP concept arose out of an approach (Buckland *et al.*, 1993) to direct DP where artificial neurons were connected in a network, with each neuron operating as a separate DP element (see Section 2.3.2) – in fact, each neuron operated as an ACAM (see Section 2.4.2). Various techniques were employed that modified the neural connections so that groups of neighboring states in the state space that had similar control requirements could be amalgamated (see Section 2.4.3). Such groups had to have evaluation function values $V_\mu(i)$ for each state i that were very similar, as well as the same optimal action (Buckland *et al.*, 1993).

These attempts at neural network amalgamation of neighboring states where largely unsuccessful because the computational effort required to make the amalgamations was often extensive and it disturbed the computations being made to learn the optimal actions (Buckland *et al.*, 1993). This research lead to the realization however that individual states on the *boundaries* of a state space region could be used to specify the action employed throughout that region. That is, if there existed *uniform regions* of states that all had the same optimal action (or the same set of optimal actions[1]), that action could be specified at the states on the boundary of a uniform region and then be maintained as long as the system remained within that uniform region. So at every *boundary state* where a uniform region might stochastically be entered from states outside that region, a new action is specified. The action thus specified is then maintained until the uniform region is left and another uniform region is entered (where another set of boundary states specifies the next action). As the system moves about each uniform region, it may pass through *dormant states* that are not on the boundary. These dormant states make no change in the action specification, they simply leave it the same. They have no DP elements associated with them at all (see Section 2.3.2). This is the fundamental idea behind TPDP.

States that are neither boundary states nor dormant states, are *external states.* As will be explained in Section 3.2.4, external states are not entered during system movement.

Figure 3.1 illustrates the TPDP concept when movement control of a "car" on a one dimensional track is desired. The car, with some initial positive velocity to the right, must pass Position A and return to the left. The "transition points" (see Section 3.2.1) in Figure 3.1 (represented by boxes) are located at all boundary states. The shaded regions indicate all of the states that the system can possibly move through given the actions specified at the boundary states and the stochastic response of the car. Shaded

[1]The simplifying assumption that there is only one optimal action in each uniform region will generally be made throughout this work. The existence of more than one optimal action does not alter the operation of TPDP in any way.



The control task shown on this phase plane is to start with a positive velocity, pass position A, and return.

Each ▣ is a transition point (TP).

Figure 3.1: Stylized Application of TPDP to a Phase Plane Control Task

states without transition points are therefore dormant states. Uniform regions consist of adjacent boundary states where the same action is specified, as well as the shaded region through which that action is maintained before another boundary is encountered (see Figure 3.1). Boundary states that do not seem to be on the main state transition routes (the one identified in Figure 3.1 for example) ensure that any stochastic deviations from those routes are realigned. Unshaded states are external states.

3.1.2 The Shape of the Uniform Region Boundaries

The boundaries that are constructed in TPDP are multi-dimensional surfaces that may range from smooth to highly irregular. The boundaries surfaces need not be closed, as boundaries only have to exist at states where a uniform region can possibly be entered from states outside that region.

3.1.3 The Benefits of TPDP

The main benefit of the TPDP approach is that, where uniform regions exist, they can be represented by a relatively small number of DP elements (depending on the shape of the boundaries and the size of the uniform regions they encompass). This reduction in memory usage results in an accompanying reduction in the learning time required to learn optimal policies (see Section 2.4.3 and Chapman *et al.*, 1991). Further, this reduction in memory usage is accomplished without having to expend any computational effort determining how states should be amalgamated (Buckland *et al.*, 1993). As will be explained in Sections 3.3.2 through 3.3.11, the determination of whether a state is a boundary state or a dormant state can be made as part of the computations that determine the optimal actions for each state.

Another benefit of TPDP is that the boundary states learn optimal actions independently of each other, and these determinations are made with the same fine resolution that they would be if the state space was fully represented with a DP element associated with every single state. In contrast, approaches that amalgamate states to achieve memory and learning time reductions result in coarse resolutions in some regions of the state space. These coarsely resolved regions can perturb the learning of the optimal policy locally and throughout the entire state space (see Section 2.4.3).

In general, TPDP learns optimal action change specifications where those specifications can be accurately placed in highly resolved state spaces. By learning only action changes, TPDP is able to "compress" the necessary action specifications and minimize its use of memory. Further, the time constant of those action change specifications (the temporal separation between them as the system moves through the state space) can vary in an unlimited way throughout the state space (as long as it is greater than the state space resolution).

3.1.4 TPDP and Inertia

TPDP is best suited to control applications where reasonably sized uniform regions exist. Such regions are most likely to be found in continuous control applications where the system exhibits inertia. In such applications, the inertia of the system limits the effects of any actions instantaneously specified by individual states if those action specifications are different from the specifications of neighboring states. In other words, no single state can make much of a difference in terms of controlling (optimally or otherwise) the system. As a result, optimal actions are initiated and maintained over relatively long periods of time as the system moves through many states.

Continuous dynamic systems with inertia are thus ideally suited to TPDP as it was described in Section 3.1.3. Applied to such systems, TPDP can determine the finely resolved uniform region boundaries necessary for optimal control, and it can learn the optimal actions for the boundary states. Some examples of systems to which TPDP control could be successfully applied are robot manipulators, chemical process control, hydraulic system control and flight control.

TPDP is *not* well suited to control applications that exhibit little or no system inertia, and that do not have large uniform regions. Much DP work has been concerned with optimal decision-making (backgammon playing is one example, Tesauro, 1991). The actions taken in decision-making tasks can drastically alter the state of the system, and it is not commonplace in such tasks for the optimal action to be the same for more than one time step. As a result, uniform regions in such control applications are ill-defined.

In general terms, TPDP performs best on continuous state space stochastic control applications (that have been discretized as described in Section 2.3.2) and performs poorly on inherently discrete control applications – applications that are better described as decision tasks. Kaelbling (1990) makes a distinction similar to this between "statistical learning" (the former), and "symbolic learning" (the latter).

3.2 The Goal of TPDP

3.2.1 Transition Points (TPs)

A *transition point* (TP) is simply the association of an action (a *TP action*, $u_{\mathrm{TP}} \in U(i)$) with a state. A state i with such an association is called a *TP state*[2]. More than one TP may be associated with each state, but one TP at each state must be chosen as the *policy TP* ($\mu(i) = u_{\mathrm{TP}}(i)$, the action of the policy TP). Dynamic systems can be entirely controlled using TPs.

Not all of the states in a system being controlled with TPs need have TPs associated with them. Those which do not are called *non-TP states*[2]. Controllers that employ TPs must maintain the last action specified by a TP when the system moves through non-TP states.

Both Barto *et al.* (1991) and Watkins (1989) have suggested something similar to TPs, but neither fully investigated the concept.

3.2.2 Environments

A system being controlled by TPs is defined as having an *environment* in which it operates. Environments include:

1. The system under consideration.
2. A set of *starting states* $S_S \subset S$ that the control task may start from, as well as the probability of it being started from each of these states and the specification of the actions to be taken in those states. The starting states S_S must include at least one state ($S_S \neq \emptyset$), and may include the entire state space S ($S_S = S$).

[2]These state definitions will be made somewhat more exclusive in Section 3.2.3.

3. A set of ***absorbing states*** $S_A \subset S$ at which the control task terminates. There need not be any absorbing states – S_A can be an empty set ($S_A = \emptyset$). When an absorbing state is encountered the task is restarted at one of the starting states S_S according to the starting probabilities.

4. A mechanism that ensures that each of the TP actions $u_{\text{TP}} \in U(i)$ at each TP state i has some finite non-zero probability of being employed each time state i is entered by the system.

5. A mechanism that ensures that each of the starting states S_S is revisited with some finite non-zero probability as the control task continues to run. Such a mechanism is necessary if states exist from which the system cannot reach *all* of the starting states S_S through some sequence of TP actions[3], and from which the system cannot reach *any* of the absorbing states S_A through some sequence of TP actions. Such states represent regions in the state space that can be entered but not left. The mechanism required when such states exist is one that, after some large number of state transitions, terminates the control task and restarts from one of the starting states S_S.

Barto *et. al.* (1991) describe something like an environment, although not to this level of specificity.

3.2.3 Closed State Spaces

Given a valid environment and a set of TPs, if all of the starting states S_S are continually revisited with some finite non-zero probability, and if state transitions are made from the starting states S_S using the TP actions (with each TP action $u_{\text{TP}} \in U(i)$ being employed in each state i with some finite non-zero probability), then eventually every state that

[3]The sequence of actions to each starting state may include passage through other starting states.

can be reached from the starting states S_S using the TP actions will be reached. This set of states is called the ***closed state space*** $S_C \subset S$. The closed state space S_C includes all the starting states S_S ($S_S \subset S_C \subset S$) and *any* states that the system can reach through a sequence of transitions from the starting states. In Figure 3.1 the shaded states represent a closed state space.

In a valid environment, the closed state space S_C is defined entirely by the starting states S_S, the existing set of TPs, and the system state transition probabilities (see Section 2.1.1). The closed state space S_C can include all of the states S. States not in the closed state space S_C are the external states $S_E \subset S$ ($S_E \cap S_C = \emptyset$). External states are never visited.

The TP states S_B are properly defined as being states within the closed state space S_C that have TPs ($S_B \subset S_C \subset S$)[4]. Similarly, the non-TP states S_D are properly defined as being states within the closed state space S_C that do not have TPs ($S_D \subset S_C \subset S$)[4].

TPs associated with external states are called ***ineffectual TPs*** because they cannot affect the movement of the system – being outside the closed state space S_C the system never reaches them.

The relationships between these various state sets are as follows:

$$S = S_C \cup S_E, \qquad S_C \cap S_E = \emptyset \tag{3.16}$$

$$S_C = S_B \cup S_D, \qquad S_B \cap S_D = \emptyset \tag{3.17}$$

That is, none of the state sets S_B, S_D, S_E overlap, and together they constitute the entire state space S.

[4]The containment of these states within S_C was not part of the Section 3.2.1 state definitions.

3.2.4 Optimal Closed State Spaces

If an optimal control policy μ^* has been determined that specifies an optimal action $\mu^*(i)$ for each state i, those optimal actions can be used to completely define a set of TPs. That is, one action (the optimal action) can be made the TP action in each state i ($u_{\mathrm{TP}}(i) = \mu^*(i)$). If the TPs so defined are employed in a valid environment that has starting states S_S and absorbing states S_A, a closed state space $S_{C^*} \subset S$ will result that is the *optimal closed state space.* This optimal closed state space includes all of the states that can be visited if the optimal actions (the TP actions) are employed after the system moves from one of the starting states S_S.

Any states outside of the optimal closed state space S_{C^*} are *optimal external states* $S_{E^*} \subset S$ ($S_{C^*} \cap S_{E^*} = \emptyset$) which are never visited if the optimal control policy μ^* is followed. As a result, using memory to represent these optimal external states in any way is unnecessary for the purposes of optimal control.

3.2.5 Minimal TP Optimal Control

Consider a state i in the optimal closed state space S_{C^*} defined by an optimal set of TPs. The *entry actions* $U_e(i)$ of that state consist of all the actions that have some non-zero probability of leading to a transition from some other state in S_{C^*} to state i[5]. If every entry action $u \in U_e(i)$ is one of the possible actions in state i ($U_e(i) \subset U(i)$)[6] and is also an optimal action if applied in state i, then no TP is necessary at state i. That is, if no action is specified at state i, actions specified previously and maintained through that state (see Section 3.2.1) will be optimal.

This is again the fundamental idea behind TPDP: that optimal control can be facilitated even in states that specify no actions as long as all the actions that cause a

[5]This definition of entry actions $U_e(i)$ also holds in closed state spaces that are not optimal.

[6]This assumption shall be used throughout this work.

transition to such states, if maintained, are optimal actions for those states.

In an optimal closed state space S_{C^*} defined by an optimal set of TPs, the ***optimal TP states*** are $S_{B^*} \subset S_{C^*}$. If every unnecessary TP is removed from S_{C^*}, a minimal set of TPs results, and ***minimal TP optimal control*** has been achieved. The optimal TP states in this case are the boundary states $S_{B^\diamond} \subset S_{C^*}$ (see Section 3.1.1). The boundary states $S_{B^\diamond}$ are thus a special case of the optimal TP states S_{B^*} when optimal control has been achieved with a minimal set of TPs.

Correspondingly, in an optimal closed state space S_{C^*} defined by an optimal set of TPs, the ***optimal non-TP states*** are $S_{D^*} \subset S_{C^*}$. When minimal TP optimal control has been achieved the non-TP states are the dormant states $S_{D^\diamond} \subset S_{C^*}$ (see Section 3.1.1), a special case of S_{D^*}. As explained in Section 3.1.1, the TPs at the boundary states ensure that optimal actions are employed throughout the uniform regions that these dormant states reside in.

3.2.6 Summary of the TPDP State Sets

This Section will summarize the various state sets involved in TPDP and describe how they change when minimal TP optimal control is achieved. Given an arbitrary set of TPs, the various state sets are:

S_C	Closed state space that results from the environment and the existing TPs.
S_B	TP states in the closed state space S_C.
S_D	Non-TP states in the closed state space S_C.
S_E	External states outside S_C (which may or may not have TPs).

As described in Section 3.2.5, minimal TP optimal control involves these state sets:

S_{C^*}	Optimal closed state space that results from the environment and the TPs.
$S_{B^\diamond}$	Boundary states in the optimal closed state space S_{C^*} that have the minimal TPs required for optimal control (special case of the optimal TP states S_{B^*}).
$S_{D^\diamond}$	Dormant states in the optimal closed state space S_{C^*} that do not require TPs (special case of the optimal non-TP states S_{D^*}).
S_{E^*}	External states outside S_{C^*} (which may or may not have TPs).

In terms of the various state sets, the following is achieved when the set of TPs is transformed into a minimal TP optimal control set:

1. The closed state space S_C defined by the environment and the TPs becomes an optimal closed state space S_{C^*}.

2. The set of TP states $S_B \subset S_C$ becomes the set of boundary states $S_{B^\diamond} \subset S_{C^*}$.

3. The set of non-TP states $S_D \subset S_C$ becomes the set of dormant states $S_{D^\diamond} \subset S_{C^*}$, leaving no unnecessary TPs within S_{C^*}.

4. The set of external states S_E becomes the set of optimal external states S_{E^*}.

3.3 The Specific Operation of TPDP

3.3.1 Pursuing Minimal TP Optimal Control

The goal of TPDP is to achieve minimal TP optimal control for any system operating within a valid environment. TPDP pursues this goal by performing two main tasks:

1. Locating the set of boundary states $S_{B^\diamond}$ that require TPs for optimal control.

2. Determining an optimal TP action for each boundary state $i \in S_{B^\diamond}$.

In other words, the right TPs must be found for the right states. Given an arbitrary set of initial TPs, TPDP modifies that set so that it is transformed into a minimal TP optimal control set. Modifications can include the *addition* and *removal* of TPs, and the *swapping* of one TP for another (each specifying different actions) at the same state i. These modifications are performed one at a time in arbitrary order, and can continue indefinitely.

During the modification process at most one TP is associated with each state at any given time. Although Section 3.2.1 stated that more than one TP could be associated with each state, the single TP restriction is part of the formal definition of TPDP – it will be relaxed in Chapter 4. This restriction implies that the one TP at each state will always be the policy TP ($\mu(i) = u_{\mathrm{TP}}(i)$ for each TP state i).

Clearly a sequence of TP modifications can be specified that will transform any initial set of TPs into a minimal TP optimal control set. For example, if the minimal TP optimal control set were known, TPs could be added to any states that were known to require them, and then all of the TPs could be swapped for ones specifying optimal actions. The difficulty is that the minimal TP optimal control set is not likely to be known. The purpose of TPDP then is to discover it. Sections 3.3.2 through 3.3.11 will explain how TPDP does so, and provide proof that TPDP will always achieve minimal TP optimal control.

3.3.2 TPDP is a Form of Q-learning

Q-learning (see Chapter 2) cannot be directly applied to learn optimal policies in controllers that use TPs. The irregularities caused by the addition, swapping, and removal of TPs would perturb any convergence of Q-learning to an optimal policy. Q-learning is used as a component process of TPDP however. Specifically, Q-learning can be employed to learn the Q-values (see Section 2.1.3) of TPs (see Section 3.3.3). Q-values can be associated with TPs to indicate the costs that result when the actions specified by those TPs are employed. In TPDP, Q-values are utilized in this manner to determine the relative merits of different TPs and to thereby indicate prudent modifications to the existing set of TPs. Sections 3.3.3 through 3.3.11 will describe fully how this is done.

Because TPDP employs Q-values, and because it updates these values using a variation of the Q-value updating Equation 2.12 (see Section 3.3.3), it is a form of Q-learning (see Chapter 2). TPDP is not strictly Q-learning however, because it does not associate a Q-value $Q(i,u)$ with each possible action $u \in U(i)$ in each state i. Instead it associates Q-values only with the state and action combinations defined by the existing set of TPs.

3.3.3 Determining the Q-values of TPs

Given an arbitrary set of TPs, and the closed state space S_C that they define in a valid environment, Q-learning can be employed to determine the Q-values of each TP in S_C. That is, Q-learning can be used to determine the exact expected infinite-horizon discounted cost (see Section 2.1.2) that results from the action that each TP specifies. This is done by performing Q-learning while using only the actions specified by the existing set of TPs to control the system. Unlike conventional applications of Q-learning, the different actions possible in each state are not randomly attempted. Only the TP actions are used.

As a result, the set of possible actions in each state i is effectively reduced to $U(i) = \{u_{\mathrm{TP}}(i)\}$. In this context only one action is possible in each state, so it must be the "optimal" action ($\mu^*(i) = u_{\mathrm{TP}}(i)$ for each state i). The motivation behind using Q-learning in TPDP is therefore to have it converge to the exact Q-values for each action specified by the TPs in the course of trivially deciding that these actions are the "optimal" ones (see Chapter 2).

Used in this manner, Q-learning cannot learn overall optimal policies. As explained in Section 3.3.2 however, it is not intended that Q-learning do so in TPDP. Instead it is used as a component process of TPDP.

Determining TP Q-values in this manner has two significant ramifications regarding ineffectual TPs and non-TP states respectively. Ineffectual TPs, like all of the external states S_E, will not be included in the Q-learning process. As defined by the existing set of TPs and the environment, the reduced Markov decision problem addressed by Q-learning in the manner described simply does not reach these external states (whether they have ineffectual TPs or not). As a result, these states do not in any way affect the Q-values of the TPs at TP states in S_C.

The second ramification is that because non-TP states S_D do not have any TPs or TP actions, there are no Q-values that can be determined for these states. Unlike the external states however, the non-TP states do affect the Q-values being determined for TPs at the TP states. This is because actions specified in the TP states and then maintained through non-TP states result in overall state transition probabilities to other TP states that are a result of the state transition probabilities of the intermediate non-TP states. But regardless of the various routes that the system may stochastically take from one TP state, through non-TP states, to another TP state, the overall probability of each such transition is fixed. If Q-learning is employed, all of the fixed overall TP state transition probabilities will be included in the various TP Q-values as direct observation of the state transitions and associated costs are made (see Section 2.2.2). As a result, the Q-value updating Equation 2.12 need only be modified to bypass the non-TP states:

$$Q_{t+d}(i,u) = \begin{cases} [1-\alpha_t(i,u)]\,Q_t(i,u) \;+\; \alpha_t(i,u)\left[\left(\sum_{n=0}^{d-1}\gamma^n c(s_{t+n},u)\right) + \gamma^d V_t(s_{t+d})\right] & \\ & \text{if } i = s_t, u = u_t \\ Q_t(i,u) & \text{otherwise} \end{cases} \tag{3.18}$$

Where d is the number of time steps after a TP state is left before another TP state is encountered. During that time the system moves through non-TP states. If Equation 3.18 is used for Q-learning, the non-TP states will be treated as inherent parts of the dynamic response of the system. The difference between Equation 2.12 and Equation 3.18 is the replacement of:

$$c(i,u) + \gamma V_t(s_{t+1}) \qquad \text{with} \qquad \left(\sum_{n=0}^{d-1} \gamma^n c(s_{t+n}, u)\right) + \gamma^d V_t(s_{t+d})$$

If d is set to 1, reflecting the case where a transition is made directly from one TP state to another with no intervening non-TP states, these Equations are the same. The Q-value updating Equation 2.12 is thus a special case of the TP updating Equation 3.18.

Proof of the convergence of Q-learning using the TP updating Equation 3.18 is presented in Appendix A. This proof is based on the work of Jaakkola *et al.* (1993).

3.3.4 Determining the Evaluation Function Values of TP States

As explained in Section 3.3.1, at most one TP is associated with each state during the operation of TPDP. As a result, the evaluation function value $V_\mu(i)$ of each TP state $i \in S_B \subset S_C$ is (based on Equation 2.6):

$$V_\mu(i) = Q(i, \mu(i)) \tag{3.19}$$

Where $\mu(i)$ is the one TP action $u_{\text{TP}}(i)$ at each state i.

Further, the following relationship can be established between the evaluation function values of two TP states:

$$V_\mu(i) = Q(i, \mu(i)) = C_\mu(i, \{j\}) + \eta_\mu(i,j) V_\mu(j) \tag{3.20}$$

Where the *excluded cost* $C_\mu(i, \{j\})$ is the expected cost of all state space transitions from state i that can occur with policy μ (see Equation 2.2) – *excluding* those *after* state j has been encountered, and $\eta_\mu(i,j)$ is the *participation factor* of $V_\mu(j)$ in $V_\mu(i)$:

$$\eta_\mu(i,j) = \sum_{\vec{x}\in X(i,j)} \left[\prod_{k=0}^{|\vec{x}|-1} \gamma p_{x_{k+1}}(x_k, \mu(i)) \right] \tag{3.21}$$

Where $X(i,j)$ is the set of all possible state transition routes $\vec{x}$ from state i to state j ($\vec{x} \in X(i,j)$), $\vec{x} = [x_0, x_1, ..., x_n]$ is one possible state transition route from state i to state j of variable length n ($x_0 = i, x_n = j$), and $|\vec{x}|$ is the number of states along each such route.

The participation factor Equation 3.21 represents the summed overall probability of each state transition route that can be taken from state i to state j, attenuated by the discount factor γ at each transition step. Participation factors will always fall between 0 and 1 ($0 \leq \eta_\mu(i,j) \leq 1$). The maximum value of participation factors is 1 because Equation 3.21 has its maximum value when γ is 1. When γ is 1, Equation 3.21 becomes the overall probability of eventually making a transition from state i to state j. This probability is between 0 and 1. The minimum value of participation factors is 0 because all the elements of Equation 3.21 are positive.

Returning to Equation 3.20, the term $\eta_\mu(i,j)V_\mu(j)$ then represents the portion of the expected total infinite-horizon discounted cost (see Section 2.1.2) at state i that occurs after the system has made a transition to state j. The evaluation function value $V_\mu(j)$ of state j reflects those costs, and the participation factor ensures that they are discounted appropriately for inclusion in the determination of $V_\mu(i)$.

Equation 3.20 can be readily extended to include an arbitrary number of states:

$$V_\mu(i) = C_\mu(i,J) + \sum_{j\in J} \eta_\mu(i,j)V_\mu(j) \tag{3.22}$$

Where $J \subset S$ is the arbitrarily chosen set of states for which explicit representation is desired in the evaluation function value computation of state i.

3.3.5 Swapping TPs

By making use of TP Q-values (determined as described in Section 3.3.3), the swapping of TPs can be facilitated. TP swapping occurs when a TP associated with a state is exchanged for another TP, specifying a different action, at that same state.

Swapping Rule: TP swapping is performed by using Q-learning to determine the Q-value of an existing TP. Then that TP is replaced with a trial TP and Q-learning is used to determine the Q-value of the trial TP. If the Q-value of the trial TP is less than that of the old TP, a swap is made. Otherwise the swap attempt is aborted.

Theorem 3.1: When a TP swap is made according to the Swapping Rule in a valid environment, the evaluation function values $V_\mu(i)$ of all of the TP states $i \in S_B \subset S_C$ will be monotonically reduced[7].

Proof: When a TP swap is made, the evaluation function value $V_\mu(j)$ of the swapped TP at state j is reduced as a direct result of the Swapping Rule and of the fact that $V_\mu(j) = Q(j, \mu(j))$ where $\mu(j)$ is the one TP action $u_{\mathrm{TP}}(j)$ at state j (Equation 3.19).

The evaluation function values $V_\mu(i)$ of all the other states i can be expressed in terms of $V_\mu(j)$ (Equation 3.20):

$$V_\mu(i) = C_\mu(i, \{j\}) + \eta_\mu(i, j)V_\mu(j)$$

Where neither $C_\mu(i, \{j\})$ nor $\eta_\mu(i, j)$ will be altered as a result of the TP swap at state j. This is because both of these values are the result of costs and state transitions probabilities encountered *before* a transition is made to state j (see Section 3.3.4).

[7]"Monotonically reduced" will mean "reduced or kept the same" throughout this work.

As a result, when $V_\mu(j)$ is reduced, the evaluation function values $V_\mu(i)$ of all the other states $i \in S_B \subset S_C$ will also be monotonically reduced. □

It is possible to attempt TP swaps at more than one state concurrently. Such concurrent swapping may be necessary to break up deadlocks which can occur if the costs experienced by more than one TP are dependent on each other, and none of the TPs can make a beneficial swap until the others have done so. To make concurrent swap attempts the Swapping Rule is applied simultaneously at each of the states concerned.

Theorem 3.2: Any and all TP swaps that are made according to a concurrent, but individual application of the Swapping Rule in a valid environment will result in the evaluation function values $V_\mu(i)$ of all of the TP states $i \in S_B \subset S_C$ being monotonically reduced.

Proof: Equation 3.22 can be readily expanded to:

$$V_\mu(i) = C_\mu(i, (S_G \cup S_{G'})) + \sum_{j \in S_G} \eta_\mu(i,j) V_\mu(j) + \sum_{j \in S_{G'}} \eta_\mu(i,j) V_\mu(j) \tag{3.23}$$

Where S_G (the *aborted swap states*) is the set of TP states for which a TP swap was attempted but aborted ($S_G \subset S_B$), and $S_{G'}$ (the *accepted swap states*) is the set of TP states for which a TP swap was attempted and accepted ($S_{G'} \subset S_B$; $S_G \cap S_{G'} = \emptyset$).

Equation 3.23 indicates the evaluation function value $V_\mu(i)$ of each TP state i before any TP swaps were made. When the complete set of TP swaps was attempted, the evaluation function value of each TP state i was the following:

$$V_{\mu'}(i) = C_{\mu'}(i, (S_G \cup S_{G'})) + \sum_{j \in S_G} \eta_{\mu'}(i,j) V_{\mu'}(j) + \sum_{j \in S_{G'}} \eta_{\mu'}(i,j) V_{\mu'}(j) \tag{3.24}$$

Where μ' indicates that the attempted TP swaps altered existing policy μ.

Now to prove that the evaluation function values of all the TP states i are monotonically reduced if a *subset* of the attempted TP swaps is accepted (based on individual application of the Swapping Rule to each swap attempt), an iterative approach will be employed determine what the resultant evaluation function values are – based on the evaluation function values before the swap attempt and on those determined during the swap attempt[8]. The iterative calculation will concern only the evaluation function values $V_\mu(i)$ of the states i for which a TP swap was attempted $(i \in (S_G \cup S_{G'}))$. The iterative calculations are defined as follows:

$$V_{k+1}(i) = \begin{cases} C_\mu(i,(S_G \cup S_{G'})) + \sum\limits_{j \in S_G} \eta_\mu(i,j)V_k(j) + \sum\limits_{j \in S_{G'}} \eta_\mu(i,j)V_k(j) & \forall\, i \in S_G \\ C_{\mu'}(i,(S_G \cup S_{G'})) + \sum\limits_{j \in S_G} \eta_{\mu'}(i,j)V_k(j) + \sum\limits_{j \in S_{G'}} \eta_{\mu'}(i,j)V_k(j) & \forall\, i \in S_{G'} \end{cases} \tag{3.25}$$

The combined usage of $C_\mu(i,(S_G \cup S_{G'}))$, $C_{\mu'}(i,(S_G \cup S_{G'}))$, $\eta_\mu(i,j)$ and $\eta_{\mu'}(i,j)$ values in Equation 3.25 is valid because of the nature of the composition of the evaluation function values. The state space transitions occurring from aborted swap states $i \in S_G$ will be the same up to the point where they encounter other states $j \in (S_G \cup S_{G'})$ where swaps were attempted. Thus, for those states $(i \in S_G)$, $C_\mu(i,(S_G \cup S_{G'}))$ and $\eta_\mu(i,j)$ will remain the same.

Similarly, after a subset of the swaps are accepted, the state space transitions occurring from accepted swap states $i \in S_{G'}$ will be the same as those experienced during the concurrent swap attempt up to the point where they encounter other states $j \in (S_G \cup S_{G'})$ where swaps were attempted. Thus, for those states $(i \in S_{G'})$, $C_{\mu'}(i,(S_G \cup S_{G'}))$ and $\eta_{\mu'}(i,j)$ will remain the same.

[8]This iterative approach does not represent a form of Q-learning, as it has nothing directly to do with the learning of an optimal policy. It is employed simply in the context of this proof.

Equation 3.25 is therefore valid for use in iterative calculations. The initial values for these calculations are:

$$V_0(i) = \begin{cases} V_\mu(i) & \forall\, i \in S_G \\ V_{\mu'}(i) & \forall\, i \in S_{G'} \end{cases} \tag{3.26}$$

Now since $V_\mu(i)$ for states $i \in S_G$ is based on evaluation function values $V_\mu(j)$ for states $j \in S_{G'}$ (see Equation 3.23) that existed before TP swaps were made at those states ($j \in S_{G'}$), and since the initial iterative value $V_0(j)$ is lower than $V_\mu(j)$ for each state $j \in S_{G'}$, $V_1(i)$ for all states $i \in S_G$ is sure to be lower than $V_0(i)$ ($V_0(i) = V_\mu(i)$). Similar analysis can be made to show that $V_1(i)$ for all states $i \in S_{G'}$ is sure to be lower than $V_0(i)$ ($V_0(i) = V_{\mu'}(i)$).

In turn, since $V_1(i)$ is lower than $V_0(i)$ for all states $i \in (S_G \cup S_{G'})$, $V_2(i)$ will be lower than $V_1(i)$ for all states i. Each iterative cycle k will likewise result in a monotonic reduction of $V_k(i)$ for each state $i \in (S_G \cup S_{G'})$. None of the iterative evaluation function values $V_k(i)$ will ever go below 0 because all of the elements of Equation 3.25 are positive. Therefore, $V_k(i)$ for each state $i \in (S_G \cup S_{G'})$ will converge on some positive value below $V_0(i)$. The exact value each converges on is not important – what is important is that it is lower than $V_\mu(i)$ for all states $i \in S_G$, and lower than $V_{\mu'}(i)$ (with $V_{\mu'}(i) < V_\mu(i)$) for all states $i \in S_{G'}$.

Because the evaluation function values for all states $i \in (S_G \cup S_{G'})$ will be monotonically reduced when a subset of the swaps are accepted, and because evaluation function values at all TP states where a swap was not attempted will also be monotonically reduced (based on a simple application of Equation 3.22 to those states), the evaluation function values for all TP states will be monotonically reduced. $\square$

Swapping can change the set of states in the closed state space S_C. That is, TP swaps can change the shape of S_C. The actions specified by swapped TPs may direct the system into states that had previously been external to S_C, and it may also direct the system entirely away from states that had previously been part of S_C. Such changes are desired if an optimal policy is sought. They result when new state transition routes outside S_C have been found that result in lower costs than those available inside S_C. The swapping of TPs makes these lower cost routes permanently available.

Changes in the closed state space S_C do not affect the validity of Theorems 3.1 and 3.2 because ineffectual TPs associated with states outside S_C (see Section 3.2.3) do not have Q-values. These states are never reached with the existing set of TPs, and it is thus meaningless to associate any sort of costs with them. As a result, if TPs are included in (or excluded from) S_C as the result of TP swapping, nothing can really be said about the changes in their Q-values.

3.3.6 The Result of Continued TP Swapping

Using the Swapping Rule, swaps can be concurrently attempted at one or more TP states in S_C (see Section 3.3.5). Each time swaps are made the evaluation function value $V_\mu(i)$ (Equation 3.19: $V_\mu(i) = Q(i, \mu(i))$) of each TP state $i \in S_C$ will be monotonically reduced. Consecutive TPs swaps therefore result in a continuous monotonic reduction of the evaluation function values of the TP states within S_C.

Theorem 3.3: Given any initial set of states with TPs, including TP states $S_B \subset S_C$ and external states S_E with ineffectual TPs, successive randomly attempted concurrent TP swaps at those states, each made according to the Swapping Rule in a valid environment, will result in a TP action being associated with every TP state in the closed state space S_C that has the lowest expected total infinite-horizon discounted cost possible at that state (for that set of states with TPs).

Intuitively Theorem 3.3 seems valid because if monotonic reduction of the evaluation function values continues for all of the TP states in S_C (according to Theorem 3.2), eventually the lowest possible value will be reached for each TP state. The issue is made complicated however by the fact that TP swaps can change the states included in S_C (see Section 3.3.5). Thus it has to be proven that, regardless of such changes to S_C, the lowest cost action will be determined for every TP state in some sort of lowest cost S_C. Basically, S_C has to be "anchored" in some way.

Proof: There are three cases to be considered for each of the starting states $i \in S_S$ from which state transitions begin:

1. The starting state $i \in S_S$ is a TP state.
2. The starting state $i \in S_S$ is *not* a TP state, but the actions specified in this state (defined as part of the environment) result in stochastic state transitions that lead to one or more TP states.
3. The starting state $i \in S_S$ is *not* a TP state, and the actions specified in this state (defined as part of the environment) *do not* result in stochastic state transitions that lead to any TP states.

In the third case actions are never specified by TPs as the system moves through the state space, so there is no way to swap TPs. The claims made in Theorem 3.3 are therefore moot.

In the first and second case the system will either start at a TP state, or a TP state will be reached after a number of state transitions from the starting state $i \in S_S$. In the second case the costs experienced *before* a TP state is encountered cannot be altered by any TP swaps. In both cases the environmental definition and the state transition probabilities ensure that a fixed set of initial *anchored states* (TP states) will always be reached. As a result, these anchored states will remain part of S_C – regardless of any TP swaps made.

Considering the anchored states, Theorem 3.2 guarantees that the evaluation function values $V_\mu(i)$ of these states will be monotonically reduced by any combination of TP swaps made in S_C (whichever other states may included in S_C at any time). Successive randomly attempted TP swaps will eventually reduce the evaluation function values of the anchored states to their lowest possible levels (given the set of states with TPs). This is true because the monotonic reduction process will never stop before that point. Any interdependent relationships that may develop between the TPs, preventing further evaluation function value reductions, can and will always be broken by some randomly attempted simultaneous swapping of all of the interdependent TPs. In the worst case, given enough random attempts, a simultaneous TP swap attempt will inevitably be made where the TP at every TP state is swapped for a TP specifying (coincidentally) the lowest cost action for that state.

Once the evaluation function values of the anchored states have been monotonically reduced to their lowest possible levels (given the set of states with TPs), it will also be true that the evaluation function values of all of the other TP states in S_C will be at their lowest possible levels (regardless of the shape of S_C at that time). This is true because, according to Equation 3.20, the evaluation function value of every anchored state can be defined using the evaluation function value of every other TP state. Therefore, for the evaluation function value of every anchored state to be minimized, the evaluation function value of all other TP states in S_C must be minimized (noting that every TP state in S_C is reached through transitions from at least one anchored state – otherwise it would not be part of S_C).

Finally, since the evaluation function value of every TP state in the closed state space S_C will inevitably be reduced to its lowest possible level (given the set of states with TPs), it must be true that an action having the lowest expected total infinite-horizon discounted cost is associated with every TP state in S_C (see Equation 2.6). □

3.3.7 The Limitations of TP Swapping

Section 3.3.6 presented proof (Theorem 3.3) that, in a valid environment with a given initial set of TP states, continual randomly attempted TP swapping eventually leads to the lowest cost action being associated with all TP states $i \in S_C$. There is no guarantee however that the resultant set of TPs are a minimal TP optimal control set (see Section 3.2.5), or that the closed state space S_C they define is the optimal closed state space S_{C^*}. There are two reasons for this:

1. It may be necessary to associate new TPs with some of the non-TP states S_D (converting them to TP states) to make S_B match S_{B^*} (see Section 3.2.6).

2. There may be state transition routes through the external states S_E that result in lower costs than those available through the closed state space S_C, and it may not be possible to discover those routes without first associating some ineffectual TPs (see Section 3.2.3) with external states. These ineffectual TPs may be necessary so that the system can be directed through the available low cost routes in the external states.

The first reason why minimal TP optimal control might be prevented, which necessitates the addition of TPs, will be addressed in Section 3.3.8. The second reason, which requires some sort of structuring of the ineffectual TP configurations in the external states S_E, will be addressed in Section 3.3.10.

3.3.8 Adding TPs

TPs are added in a manner very similar to how they are swapped. The Q-value $Q(i, u_{\text{TP}})$ that results when a trial TP is associated with a non-TP state $i \in S_D$ is determined and compared to an assessment of the costs that result when no TP is associated with state i. This comparison procedure is made difficult however by the fact that, since a non-TP state has no TP, it has no Q-value associated with it that can be used to determine the costs incurred when there is no TP. As a result an *R-value* is instead determined. R-values are determined the same way that TP Q-values are, using the Q-value updating Equation 3.18 with only notation change:

$$R_{t+d}(i) = \begin{cases} [1 - \alpha_t(i)]\, R_t(i) + \alpha_t(i) \left[\left(\sum_{n=0}^{d-1} \gamma^n c(s_{t+n}, u) \right) + \gamma^d V_t(s_{t+d}) \right] & \text{if } i = s_t \\ R_t(i) & \text{otherwise} \end{cases} \tag{3.27}$$

Where d is the number of time steps after the non-TP state i is left before a TP state is encountered, and u is an action specified at some time step before t.

Evaluation function values are not generated from R-values. It is not necessary to do so as neither R-value nor Q-value updates make use of evaluation function values at non-TP states – the states R-values are associated with.

To illustrate that the R-value updating Equation 3.27 will converge on exact R-values that indicate the costs incurred if no action is specified in a given non-TP state, *entry action probabilities* first have to be explained. Entry action probabilities $\rho(i, u)$ indicate, for a given state i, the probability of the system arriving at i as the result of an action $u \in U_e(i)$ specified in some previous TP state, relative to the probability that the system will arrive at i at all. This means that:

$$\sum_{u \in U_e(i)} \rho(i,u) = 1 \tag{3.28}$$

Entry action probabilities are entirely a function of the existing set of TPs and the environment, as these determine the probability of the system making state transitions from the starting states S_S to TP states where action $u \in U_e(i)$ can result in a transition to i. The entry action probabilities $\rho(i,u)$ for any state i can thus be altered by any modification of the existing set of TPs.

The entry action probabilities $\rho(i,u)$ for a system operating with a given set of TPs in a valid environment will be fixed. As a result, a non-TP state i can be considered to be a TP state that specifies a non-action action $\tilde{u}$. The state transition probabilities $p_j(i,\tilde{u})$ that result from this non-action action will be a product of the entry action probabilities and the system state transition probabilities, and they too will be fixed:

$$p_j(i,\tilde{u}) = \sum_{u \in U_e(i)} \rho(i,u) p_j(i,u) \tag{3.29}$$

If a non-TP state i can be considered to be a TP state that specifies a non-action action $\tilde{u}$ which results in fixed state transition probabilities $p_j(i,\tilde{u})$, then Q-learning can be applied (as described in Section 3.3.3) to learn the Q-value $Q(i,\tilde{u})$ that results from that action. A Q-value determined in this manner is in fact an R-value. So Equation 3.27, which is essentially the TP updating Equation 3.18, can be employed to learn the R-value of a non-TP state.

With exact R-values available, the addition of TPs can be made according to the following Addition Rule.

Addition Rule: TP addition is performed by using Q-learning to determine the R-value of a non-TP state. Then a trial TP is placed at that state and Q-learning is used to determine the Q-value of that TP. If the Q-value of the trial TP is less than or equal to the R-value, the TP is added. Otherwise the addition is aborted.

Even though the evaluation function value $V_\mu(i)$ (Equation 3.19: $V_\mu(i) = Q(i, \mu(i))$) of a state i with a newly added TP will be less than or equal to the R-value of that state before the TP was added (as a result of the Addition Rule), the addition of a TP to a state does not necessarily result in monotonic reduction of the evaluation function values of the other TP states. Consider that the R-value $R(i)$ for a non-TP state i is a product of the Q-values of the entry actions to that state:

$$R(i) = \sum_{u \in U_e(i)} \rho(i,u) Q(i,u) \tag{3.30}$$

Now if there existed two TP states a and b that always made a transition to non-TP state i with the same action, $u_a \in U_e(i)$ for one and $u_b \in U_e(i)$ for the other, with $Q(i, u_a)$ being lower than $Q(i, u_b)$, then it may be possible to add a TP to state i that has an intermediate Q-value $Q(i, u_{\mathrm{TP}})$ between $Q(i, u_a)$ and $Q(i, u_b)$. In this case, depending on the entry action probabilities (see Equation 3.30), the R-value $R(i)$ of state i might be higher than $Q(i, u_{\mathrm{TP}})$. The TP would then be added (as a result of the Addition Rule), increasing the costs experienced by TP state a as it reduced the costs experienced by TP state b (see Equation 2.2). In this example the low costs of entry action u_a are balanced by the high costs of entry action u_b so that the overall R-value (see Equation 3.30) of the non-TP state is higher than the Q-value of the TP added to it. Such a scenario may be unlikely, but this example illustrates that the addition of a TP to a state, done according to the Addition Rule, may result in an increase in the costs experienced by other TP states. These increased costs will be reflected in the evaluation function values of those states.

Theorem 3.4: When a TP addition is made to state $i \in S_D \subset S_C$ according to the Addition Rule in a valid environment, either the Q-value $Q(i, u_{\text{TP}})$ of the added TP will be less than the Q-value $Q(i, u)$ of at least one entry action $u \in U_e(i)$, or $Q(i, u_{\text{TP}}) = Q(i, u) \; \forall \; u \in U_e(i)$.

Proof: By contradiction. For Theorem 3.4 *not* to be true $Q(i, u_{\text{TP}})$ must be greater than $Q(i, u)$ for at least one entry action $u \in U_e(i)$, and equal to all of the rest. If such were the case then:

$$\sum_{u \in U_e(i)} \rho(i,u) Q(i, u_{\text{TP}}) > \sum_{u \in U_e(i)} \rho(i,u) Q(i,u) \tag{3.31}$$

Which, because of Equation 3.28, means that:

$$Q(i, u_{\text{TP}}) > \sum_{u \in U_e(i)} \rho(i,u) Q(i,u) \tag{3.32}$$

Considering Equation 3.30, this means that:

$$Q(i, u_{\text{TP}}) > R(i) \tag{3.33}$$

Which is a violation of the Theorem 3.4 assumption that the Addition Rule is adhered to. □

Theorem 3.4 illustrates why TPs must be added. If an addition attempt is successful then the costs incurred when the system encounters state i will either remain the same for all entry actions $u \in U_e(i)$, or will be reduced for at least one entry action. In the former case the evaluation function values of all the TP states in S_C will remain the same and the addition of the TP will do no harm. In the latter case some of the evaluation function values of the TP states may be increased as a result of the TP addition, but the addition is necessary to facilitate eventual optimal control of the system. This is because, without a TP at state i, there would be no way to reduce the costs incurred when the system encounters that state as a result of any entry actions $u \in U_e(i)$ whose Q-values are higher than $Q(i, u_{\mathrm{TP}})$. If some of the entry action Q-values are lower than $Q(i, u_{\mathrm{TP}})$, a swap will eventually occur at state i for a TP with a Q-value the same or lower than that of the lowest entry action.

Theorem 3.5: A TP addition can always be made to a state $i \in S_D \subset S_C$ according to the Addition Rule in a valid environment.

Proof: A TP addition can always be attempted where the action u_{TP} associated with that TP has a Q-value $Q(i, u_{\mathrm{TP}})$ that is as low as the lowest Q-value $Q(i, u)$ of all the entry actions $u \in U_e(i)$ in state i. The Q-value $Q(i, u_{\mathrm{TP}})$ of such a TP is certain to be less than or equal to the R-value $R(i)$ of non-TP state i because the R-value, according to Equation 3.30, is composed of the entry action Q-values – all of which are the same or higher than the Q-value of the action the TP specifies. As a result, the Addition Rule will ensure that the TP is added. This will occur even if all of the entry action Q-values are the same as $Q(i, u_{\mathrm{TP}})$. In that case the R-value $R(i)$ of non-TP state i will be equal to $Q(i, u_{\mathrm{TP}})$, which is sufficient for the Addition Rule. □

3.3.9 Removing TPs

In order to achieve minimal TP optimal control it may be necessary to remove TPs. Some TPs may be unnecessarily associated with dormant states $S_{D^\diamond}$. TP removal is performed in a manner reverse that of TP addition.

Removal Rule: TP removal is performed by using Q-learning to determine the Q-value of a TP. Then removal of that TP is attempted and Q-learning is used to determine the R-value of the resultant non-TP state. If the R-value of the trial non-TP state is less than or equal to the Q-value of the TP, the TP is removed. Otherwise the removal is aborted.

Even though the R-value of a state i with a recently removed TP will be less than or equal to the evaluation function value $V_\mu(i)$ of that state when it had a TP, the removal of TPs does not necessarily result in monotonic reduction of the evaluation function values of the other TP states. This is true for the same reasons that it is true of TP addition (see Section 3.3.8). Briefly, TP removal can reduce the costs incurred when the system encounters a state with some entry actions while increasing those incurred with others.

Theorem 3.6: A TP can only be removed from the state $i \in S_B \subset S_C$ it is associated with, according to the Removal Rule in a valid environment, if at least one entry action $u \in U_e(i)$ has a Q-value $Q(i,u)$ less than the Q-value $Q(i, u_{\mathrm{TP}})$ of that TP, or if $Q(i, u_{\mathrm{TP}}) = Q(i,u) \; \forall \; u \in U_e(i)$.

Proof: By contradiction. For Theorem 3.6 *not* to be true the Q-value $Q(i, u_{\mathrm{TP}})$ of a removed TP must be less than $Q(i,u)$ for at least one entry action $u \in U_e(i)$, and equal to all of the rest. If such were the case then:

$$\sum_{u \in U_e(i)} \rho(i,u) Q(i, u_{\mathrm{TP}}) < \sum_{u \in U_e(i)} \rho(i,u) Q(i,u) \tag{3.34}$$

Which, because of Equation 3.28, means that:

$$Q(i, u_{\mathrm{TP}}) < \sum_{u \in U_e(i)} \rho(i,u) Q(i,u) \tag{3.35}$$

Considering Equation 3.30, this means that:

$$Q(i, u_{\mathrm{TP}}) < R(i) \tag{3.36}$$

Which is a violation of the Theorem 3.6 assumption that the Removal Rule is adhered to. □

The overall effect of the Addition and Removal Rules is that TPs may be added and removed from states continuously depending on how their Q-values compare with the R-values of the states concerned at any given time. Once the lowest Q-value TP is associated with a state however, it cannot be removed unless all of the entry actions $u \in U_e(i)$ have the same low Q-value. If the state is not a dormant state, this will never occur.

3.3.10 Preparing External States

Sections 3.3.8 and 3.3.9 described how TPs are added to non-TP states S_D within S_C, and how they will be adopted permanently if those states are not dormant states. Such additions ensure that, after extended TP swapping, the lowest cost actions are associated with every TP state in S_C (Theorem 3.3). There is no guarantee however that the resulting set of TPs defines an optimal closed state space S_{C*}. To guarantee this, all of the state transition routes external to S_C (through S_E) must be tried to ensure that none exist that have lower costs than those available within S_C. If lower cost routes are found, TP modifications must be made to change the shape of S_C so that it will include them.

In a given environment, access to external states S_E is facilitated by making TP modifications in S_C that direct the system into the external states. External states that cannot be accessed by TP modifications are not of concern as they simply cannot be reached in the existing environment.

Theorem 3.7: If some external states are entered in a valid environment as the result of a TP addition, swap or removal attempt at state $i \in S_C \subset S$, and the Q-value or R-value of the attempted TP modification is less than the Q-value or R-value normally experienced at state i, then the TP will be modified according to the Addition, Swapping or Removal Rule (whichever is appropriate) to include the external states in S_C.

Proof: The immediate costs experienced in the external states S_E are no different from those experienced in S_C (see Equation 2.2). As a result, the Q-values and R-values determined during TP addition, swap or removal attempts are the same regardless of whether external states are entered or not. The Addition, Swapping and Removal Rules will therefore operate in exactly the same manner, making the TP modification if it has been found to result in a lower Q-value or R-value than that normally experienced. □

Theorem 3.7 is fairly self-evident. It was presented mainly to clarify the situation with external states – specifically that they are no different from internal states in terms of experienced costs. This assumption was implicitly made in Theorems 3.1 through 3.6 because the TP modifications that each of these Theorems described could have included entry into external states.

So external states can readily be included in S_C if state transition routes through them result in low enough costs. TP additions, swaps and removals in the existing closed state space S_C will facilitate the inclusion of any low cost external state routes. There is a difficulty that arises from assessing external state routes in this manner however. This is that lower cost transition routes through the external states may not be readily available. It may not be a simple matter of just entering the external states to discover the lower cost routes. Instead it may be necessary to have some ineffectual TPs already in place that can guide the system through convoluted low cost routes. Only then might some low cost routes be discovered.

TP modifications in the external states do not affect the costs experienced in the closed state space S_C (see Section 3.3.3), so ineffectual TP configurations can be modified freely in the external states to prepare low cost routes.

If random TP configurations are successively adopted in the external states and if a TP configuration is possible that results in a lower cost state transition route through these states, eventually this configuration will be discovered by a TP modification in S_C and the lower cost route will be adopted. It may be desirable to prepare external state TP configurations in a more structured manner however – a manner that facilitates quicker discovery of the lower cost routes. This can be done by somehow modifying external state TP configurations that have already been found to be of relatively low cost, or it can be done by employing some knowledge of the system response.

3.3.11 Minimal TP Optimal Control is Guaranteed

Combining the results of Sections 3.3.3 through 3.3.10 a guarantee of TPDP converging to optimal control and achieving minimal TP optimal control (see Section 3.2.5) can be made[9]:

> **Theorem 3.8:** Given any initial set of TPs in a valid environment, by sequentially making random TP addition, swap and removal attempts at states in the closed state space S_C according to the Addition, Swapping and Removal Rules, and by simultaneously randomly configuring ineffectual TPs in the external states S_E, optimal control of the system will be converged to and minimal TP optimal control will be achieved.
>
> **Proof:** Theorem 3.8 is true because a worst-case achievement of minimal TP optimal control is always possible given a long enough random sequence of TP modifications. It can occur as follows:
>
> 1. TPs are randomly and sequentially added to every boundary state $S_{B^\diamond}$ in S_C. Such additions can always be made (Theorem 3.5).

[9]Assuming that accurate Q-values and R-values can be determined during the TPDP process (see Sections 3.3.3 and 2.2.3, and Appendix A).

2. Ineffectual TPs are randomly configured to be at every boundary state $S_{B^\diamond}$ in S_E. Such TPs can be added at any time (see Section 3.3.10).

3. Repeated, randomly attempted TP swapping monotonically reduces the evaluation function values of all TP states in S_C to their lowest possible levels, and thereby results in the lowest cost action being associated with every TP state (Theorem 3.3). During this process all low cost routes through the external states will be incorporated into S_C (Theorem 3.7).

4. As TPs are associated with all of the boundary states $S_{B^\diamond}$, when lowest cost actions are associated with all of the TP states, optimal control is achieved (see Section 3.2.4). This optimal control will be permanently maintained because TP additions, swaps and removals cannot be made that will increase costs above the optimal costs experienced with optimal control (Theorems 3.4, 3.2 and 3.6 respectively).

5. Randomly attempted TP removals eliminate all of the unnecessary TPs at the dormant states $S_{D^\diamond}$ in S_C, reducing the TP states to the boundary states $S_{B^\diamond}$. At this point minimal TP optimal control is achieved. Such unnecessary TPs can always be removed (Theorem 3.6).

After minimal TP optimal control is achieved, random TP additions may repeatedly add unnecessary TPs to the dormant states $S_{D^\diamond}$. Such is possible according to Theorem 3.4, but it will not affect the optimal control of the system and eventually these TPs will be randomly removed again.

While TP additions and removals may continue after minimal TP optimal control is achieved, no TP swaps will occur. This is because the Swapping Rule prevents TP swaps after the lowest cost action is associated with any TP state concerned. $\square$

The proof of Theorem 3.8 is based on an unlikely sequence of random TP modifications being made. This sequence will inevitably occur however, given enough random TP modifications. Unfortunately the complexities of the TPDP process require that such a brutish proof be made.

In practice however, it is highly probable that TPDP can achieve minimal TP optimal control in a much more straight-forward way. This is because, if Theorems 3.1 through 3.7 are considered, the Addition, Swapping and Removal Rules all tend to either monotonically reduce costs or to ensure the continuance of important low cost TPs. Effectively then, TPDP "locks-in" any cost reductions discovered. As a result, TPDP steadily determines a set of TPs that will provide minimal TP optimal control. Further, in contrast to the sequence of events described in the proof of Theorem 3.8 that span the entire state space, in most applications (ones without any complex cost interdependencies between large numbers of states) TPDP can operate in a piecemeal manner, reducing the costs experienced in small regions of the state space independently of other small regions. Every such set of small reductions contributes to the overall advancement towards minimal TP optimal control.

3.3.12 Summary of TPDP Operation

Section 3.3.8 described how TPs can be added to non-TP states in the closed state space S_C so that actions can be specified at every state where such specification is required for optimal control. Section 3.3.10 detailed how ineffectual TPs can be configured in the external states S_E to prepare state transition routes through S_E which may result in lower costs than those available in S_C. Section 3.3.5 described how TP swaps are made, and Section 3.3.6 explained how repeated TP swaps monotonically reduce the evaluation function values of the TP states within S_C. Section 3.3.9 explained how TPs could be removed from states if they were not required for optimal control, facilitating the minimization of the set of TPs.

Together these TPDP operations result in: the "filling out" of S_C so that it acquires all of the TPs necessary for optimal control; the configuration of TPs in S_E to facilitate the inclusion of lower cost external state transition routes in S_C; the swapping of TPs so that optimal control can be achieved; and the removal of unnecessary TPs. When combined, these operations lead to minimal TP optimal control.

3.4 The Characteristics of TPDP

3.4.1 The Characteristics Shared with Q-learning

Because TPDP is a modified form of Q-learning, it has many of the characteristics that Q-learning has. As described in Chapter 2, these include the fact that TPDP is:

- direct dynamic programming control
- memory-based control
- reinforcement learning control
- a solution to the credit assignment problem
- adaptive control

3.4.2 A Minimal Form of Direct DP Control

Like Q-learning, TPDP employs no explicit system models (see Section 2.2.1), so it uses less memory than indirect dynamic programming controllers (see Section 2.3.1). Beyond this however, TPDP is likely to use substantially less memory than Q-learning in the same continuous control applications. This is true for two reasons. The first is that TPDP stores only individual TPs; that is, the single Q-value, R-value and action that a

TP associates with a given state. Q-learning on the other hand stores a complete set of Q-values for each state – although it does not have to store R-values.

The second reason that TPDP can use less memory than Q-learning is that, if an ACAM is employed to store the TPs (see Section 2.4.2), TPDP can take advantage of uniform regions in which dormant states exist. Such dormant states require no TPs, and TPDP can thus save memory for each dormant state. Even though Q-learning can take advantage of the ACAM storage of Q-values (see Section 2.4.2), it cannot take advantage of this further saving. It must store the Q-values of every state encountered during control of the system.

The nature of the system being controlled determines how much of a memory saving is possible if TPDP is employed instead of Q-learning. The advantages of TPDP are reduced if there are few uniform regions in which dormant states exist (see Section 3.1.4), or if there are few possible actions $U(i)$ in each state i.

3.4.3 TPDP is Action Centered

Unlike most DP approaches to control, TPDP is highly focused on the consequences of actions. Determining optimal actions is the final goal of any DP controller, but most are primary concerned with learning an evaluation function that leads indirectly to optimal policy determination. TPDP is directly concerned with evaluating specific TPs; that is, evaluating specific actions taken in specific states. This is true of Q-learning controllers in general, but it is especially so with TPDP because of the emphasis on adding, swapping and removing TPs that are associated with specific actions.

Because TPDP focuses on the effects of actions instead of on the determination of an evaluation function, the evaluation function approximation techniques described in Section 2.4.4 are not really appropriate for TPDP. In general, TPDP is a very concrete, memory-based approach to control.

3.4.4 TPDP and Temporal Differences

Section 2.3.6 described Q-learning in the context of temporal difference learning. It was explained in that Section how conventional one-step Q-learning (see Section 2.2.5) is a form of TD(0) learning. The relationship of TPDP to temporal difference learning is less clear. This is because the updating that occurs in TPDP involves evaluation function values that can be further than a single time step away from the state being updated (see Equation 3.18). TPDP is thus not TD(0) learning. TPDP is also not TD(1) learning because it does not update using only the actual costs experienced after an action has been specified in a given state (see Section 2.3.6). TPDP updates with a series of immediate costs followed by a single evaluation function value, so it is some hybrid of TD(0) and TD(1) learning – maybe "stretched TD(0) learning".

Even though TPDP lies in the middle ground between TD(0) and TD(1), it would be inaccurate to describe it as TD(λ) learning (Sutton, 1988). This is because it does not make updates using combinations of immediate costs and evaluation function values (see Equation 2.15).

3.4.5 Continuous State Spaces and TPDP

As explained in Section 3.1.4, TPDP performs best on continuous state space stochastic control applications. To facilitate control on these applications however, the continuous state space must be discretized, with each state dimension being divided into a number of intervals (see Section 2.3.2). Furthermore, TPDP operates by sampling the system state in discrete time intervals. So while TPDP operates on continuous systems, it does so by sampling a discretized state space in discrete time intervals. As long as the level of discretization is fine enough in both space and time, this approach can be successfully employed to control continuous systems.

Up to this point the description of TPDP has involved only the time step interval of 1 that is conventionally used with discrete state spaces. In continuous state spaces control is resolved with a general *time step interval* T. The time step interval can be made as short as is necessary (given the practical limitations of the controller hardware) to effectively follow the continuous response of the system. As TPDP performs best in continuous state spaces, the general time step interval T will be used henceforth in the description of TPDP.

An incidental result of reducing the time step interval T is that the Q-values determined by any DP controller employing Q-learning will be changed. Even though the same immediate costs $c(i,u)$ will be experienced as the system moves about the state space, the immediate costs experienced over shorter intervals will be added to the total infinite-horizon discounted cost and discounted more frequently. This can greatly alter the Q-values $Q(i,u)$ that are learned for each action u taken in each state i if updating Equations 2.12 or 3.18 is employed. To roughly compensate for this effect the following equations can be used to calculate new $c'(i,u)$ and γ' values from the old ones when a time step interval change has been made and it is desired that the same total infinite-horizon discounted costs be experienced:

$$c'(i,u) = c(i,u)\frac{T'}{T} \tag{3.37}$$

$$\gamma' = \gamma^{\frac{T'}{T}} \tag{3.38}$$

Equation 3.38 will result in accurate discounting of the evaluation function value $V_t(s_{t+d})$ that is included in updating Equations 2.12 and 3.18[10]. The immediate cost components in these equations will not be discounted entirely accurately however. Assuming that a constant immediate cost per unit time is experienced over all time step intervals, and that $\frac{T}{T'}$ is an integer, the following factor of error will be incurred in the

[10]When the update occurs after exactly $\frac{T}{T'}$ and $d\frac{T}{T'}$ time steps respectively.

	$\gamma = 0.8$		$\gamma = 0.9$	
	$\frac{T}{T'} = 10$	$\frac{T}{T'} = 100$	$\frac{T}{T'} = 10$	$\frac{T}{T'} = 100$
$d = 1$	-9.4%	-10.3%	-4.6%	-5.0%
$d = 10$	-41.8%	-45.8%	-29.9%	-32.8%

Figure 3.2: Sample Immediate Cost Error Factors

immediate cost portion of updating Equations 2.12[11] and 3.18:

$$\text{immediate cost error factor} = \frac{T'}{T} \cdot \frac{\gamma^d - 1}{\gamma^{\frac{T'}{T}} - 1} - \frac{\gamma^d - 1}{\gamma - 1} \tag{3.39}$$

Figure 3.2 presents some immediate cost error factors predicted by Equation 3.39 for some γ and $\frac{T}{T'}$ values.

[11]For Equation 2.12 d is always 1.

Chapter 4

Practical TPDP

This Chapter will present a description of a practical form of TPDP. The theoretical form of TPDP presented in Chapter 3 has a number of limitations that make it impractical for use in real control applications. These limitations are addressed in the practical form of TPDP.

4.1 The Practical TPDP Approach

4.1.1 The Problem With the Theoretical Form of TPDP

TPDP was theoretically described in Chapter 3 as a method of learning a minimal set of TPs that could provide optimal control of a system in a valid environment (see Section 3.2.2). As such it was shown that TPDP is certain to arrive at this minimal set if random TP additions, swaps and removals are continually made over an indefinite but finite period of time (see Section 3.3.11). The problem with this approach is that each TP addition, swap[1] and removal requires the determination of exact Q-values and R-values. For such values to be determined, Q-learning must be allowed to progress with the existing set of TPs until Q-values and R-values have been converged upon which are exact or negligibly close to being exact. This can take a considerable period of time, and this process must be repeated for each TP addition, swap and removal. As a result, the theoretical form of TPDP described in Chapter 3 is not very practical in terms of being

[1]Of which more than one can be made concurrently (see Section 3.3.5).

a viable control approach.

4.1.2 Concurrent Assessment of TP Modifications

To solve the problem of the protracted learning time required by the theoretical form of TPDP, many TP modifications (additions, swaps and removals) can be assessed concurrently. That is, Q-learning can be employed not just to determine the Q-values and R-values for a single TP modification, but instead to learn these values for a number of concurrent modification attempts. Further, modification attempts, and the learning of the values required for them, need not be initiated simultaneously. The determination of each value can be made part of the Q-learning process whenever new TP modifications are randomly attempted. This approach is called *Practical TPDP.*

Practical TPDP basically consists of a continually running Q-learning process, where Q-learning is used to learn a constantly changing set of TP Q-values and R-values[2]. These values are used in the assessment of randomly attempted TP modifications. The TPs being assessed at any one time (with their associated Q-values and R-values) constitute the *assessment group.*

Practical TPDP can result in more than one TP being simultaneously associated with a state. As a result a policy TP (see Section 3.2.1) must be determined that defines the policy action $\mu(i)$ (see Section 2.1.1) for each state i.

4.1.3 Conventional Q-learning and Practical TPDP

To the extent that the Q-learning process is continuous in Practical TPDP (instead of consisting of successive rounds of Q-learning in the manner of the theoretical form of TPDP), and to the extent that a policy action $\mu(i)$ may have to be chosen from actions

[2]A separate R-value must be associated with each TP at the same state when concurrent evaluations are made (see Appendix Section B.3, Lines 20 to 33). As a result, the R-value notation must be modified accordingly: $R(i)$ becomes $R(i,u)$.

specified by a number of TPs associated with the same state i, Practical TPDP is like conventional Q-learning. Practical TPDP is unlike conventional Q-learning however in that the addition, swapping and removal of TPs completely disrupts the convergence to optimal values that occurs in conventional Q-learning (see Section 2.2.3).

4.1.4 Minimal TP Optimal Control and Practical TPDP

As explained in Section 4.1.3, Practical TPDP disrupts the convergence to optimal values that occurs in conventional Q-learning. As a result, exact Q-values and R-values are not sure to be determined in Practical TPDP (see Sections 3.3.3 and 2.2.3, and Appendix A), making the proof that minimal TP optimal control will be achieved with the theoretical form of TPDP (Theorem 3.8) inapplicable to Practical TPDP. No extended proof has been developed that can surmount the complex Q-value and R-value interdependencies that exist in Practical TPDP. So no proof exists that Practical TPDP is certain to achieve minimal TP optimal control.

Nonetheless, it is reasonable to assume that Practical TPDP will achieve minimal TP optimal control. The reasons why this is true are the same reasons why the theoretical form of TPDP is likely to achieve minimal TP optimal control more readily than through the unlikely sequence of events described in Theorem 3.8 (see Section 3.3.11). Basically, the addition, swapping and removal of TPs tends to steadily reduce the costs experienced during system control and to "lock-in" these cost reductions[3]. This will occur whether these TP modifications are made based on the exactly determined Q-values and R-values of the theoretical form of TPDP, or on values that are determined in the concurrent manner of Practical TPDP. Chapter 5 will present some demonstrations of the effectiveness of Practical TPDP.

[3]In fact, the proofs presented in Chapter 3 were developed mainly to indicate what could be expected (although not guaranteed) with Practical TPDP.

4.2 The Specific Operation of Practical TPDP

4.2.1 Using Weights to Concurrently Assess TPs

The main difficulty that arises when TPs are concurrently assessed is that of determining when an assessment is complete. That is, when the Q-values and R-values associated with each TP have been learned well enough for a TP modification to be made based on them. The technique employed to address this problem is to associate a *weight* $w(i, u)$ with each TP that indicates the general merit of that TP. The basic idea of weights is to facilitate the random addition of trial TPs to a TP assessment group with a low initial weight $w_{\textbf{initial}}$. The Q-values and R-values of the TPs in the assessment group are then learned in an ongoing Q-learning process, and the TP weights are adjusted heuristically using those values:

1. New TPs are given an initial weight of $w_{\textbf{initial}}$; $0 < w_{\textbf{initial}} < w_{\textbf{max}}$.
2. Each time the Q-value of a TP is updated (whenever the system enters the state associated with that TP and takes the action it specifies), the weight $w(i, u)$ of that TP is incremented if $Q(i, u) < R(i, u)$ and decremented otherwise.
3. Each TP weight $w(i, u)$ is limited to a maximum value of $w_{\textbf{max}}$.
4. If a TP weight $w(i, u)$ is decremented to 0, the TP is removed.

The TP weights are limited to a maximum value of $w_{\textbf{max}}$ to prevent them from becoming so large that they cannot readily be reduced again if the system or the learning situation changes.

A simpler approach to TP modification assessment might have been to associate some sort of counter with the Q-values and R-values of each TP that indicated how many times they had been updated. These counters could then be used in assessment decisions. Similar techniques were discussed in Section 2.4.5. The approach using weights was chosen because it facilitated the prompt removal of TPs that were clearly not prudent for a state to employ, and because it allowed for an ongoing assessment of each TP as conditions changed.

4.2.2 Policy TP Determination

After each change in TP weights or Q-values, the policy TP for the state associated with that TP has to be redetermined. This is done by considering all of the TPs with weights $w(i,u)$ greater than or equal to w_{thr} ($w_{\text{initial}} < w_{\text{thr}} < w_{\text{max}}$), and finding the one with the lowest Q-value $Q(i,u)$. The threshold value w_{thr} prevents new and untested TPs from being made policy TPs (see Section 4.1.2). This decision process is formalized as finding a TP whose associated action u_{TP} fulfills:

$$\min_{u \in U(i)} Q(i,u) \qquad \forall\, u \text{ with } w(i,u) \geq w_{\text{thr}} \tag{4.40}$$

If no TPs exist whose weights fulfill this criteria ($w(i,u) \geq w_{\text{thr}}$), the policy TP is determined by finding the TP whose weight is closest to w_{thr}. This decision process is formalized as finding the TP whose associated action fulfills:

$$\max_{u \in U(i)} w(i,u) \qquad \forall\, u \text{ with } w(i,u) > 0 \tag{4.41}$$

If a state has only one TP associated with it, that TP will always be the policy TP.

Regarding the formalities of the theoretical form of TPDP presented in Chapter 3, this combined approach to TP assessment (see Section 4.2.1) and policy TP determination results in TP additions when a TP is associated with a non-TP state. TP swaps result when, according to Equations 4.40 and 4.41, the policy TP at a state is changed. And TP removals result when the single TP associated with a state (which must be the policy TP) is removed.

4.2.3 Exploration in Practical TPDP

There are two basic types of exploration which must be performed in Practical TPDP. The first, as described in general terms in Section 2.2.4, is to continually try the actions that can be performed in each state i. In the case of Practical TPDP, this means that the actions specified by all of the TPs associated with each state must be continually tried. Thorough exploration of this type is readily ensured by randomly selecting between the TP actions associated with each state encountered. Practical TPDP makes such random selection, so this type of exploration will be discussed no further.

The second type of exploration involves the identification of new TPs that should be assessed by Q-learning. Such TPs can lead to policy improvements, and they can be associated both with TP states and non-TP states. There are two modes of this type of exploration, *internal exploration* and *external exploration.*

The purpose of internal exploration is to identify TPs that may reduce the costs experienced within the closed state space S_C defined by the existing set of policy TPs (see Section 3.2.1). Internal exploration is performed by randomly trying sequences of actions in S_C that are not specified by the existing set of TPs. If such experimental sequences of actions result in lower costs than would have been experienced with the existing set of TPs, new TPs that specify the experimental actions are associated with the states where those actions were applied. Those TPs, and the actions they specify, are then fully assessed using Q-learning – they are made part of the assessment group. This process is described in detail in Appendix Section B.3.

The purpose of external exploration is to associate TPs with external states S_E (see Section 3.2.1) to prepare state transition routes that can be followed by the system through those states. Such preparation may be necessary for the discovery of lower cost routes through the external states (see Section 3.3.10). External exploration is performed by having the system make random action specifications in S_E until a TP is encountered. When a TP is encountered, a new TP is associated with the state where the last random action was specified. This TP then indicates a route that the system can follow to return to the closed state space S_C when it is moving through the external states S_E. Over time such TP allocations will build upon each other to indicate complete routes through the external states S_E. TP allocations are made in the external states in this conservative manner to ensure that the number of TPs allocated does not become excessive. Excessive allocation would occur, for example, if TPs were allocated every time a new action was specified during external exploration.

Generally, internal exploration can be viewed as a way of ensuring that uniform region boundaries (see Section 3.1.1) are located correctly. That is, ensuring that the TPs that define these boundaries are associated with exactly the right states and that they should not be shifted slightly in the state space. Internal exploration also ensures that absolutely optimal actions are specified at the boundaries. External exploration can be viewed as a way of ensuring that no state space transition routes are available in S_E that result in lower costs than those already being followed in S_C.

As described, internal and external exploration result respectively in the allocation of new, lower cost TPs in the closed state space S_C, and in the arbitrary allocation of TPs in the external states S_E. In reality the correspondence between the two modes of exploration and the portion of the state space S in which the TPs are allocated is rather rough. This is because there is no practical way to fully discriminate between S_C and S_E during the operation of Practical TPDP. This is not a problem however, because the clear demarcation of these two regions was not found to be required for the successful operation of Practical TPDP (see Chapter 5).

4.2.4 General Operation of the Practical TPDP Algorithm

Figure 4.3 presents a complete algorithm for Practical TPDP that makes use of the TP assessment method described in Section 4.2.1, the policy TP determination method described in Section 4.2.2, and the internal and external exploration described in Section 4.2.3. This *Practical TPDP Algorithm* is employed once for every learning *trial* (sequence of state transitions from the starting states S_S to the absorbing states S_A[4]), and it repeatedly calls the *Stack Updating Procedure* presented in Figure 4.4. The "stack" (Standish, 1980) contains time step, state and action information about all TPs that are encountered in the period between every set of Q-value, R-value and weight updates requested by the Practical TPDP Algorithm (see Appendix Section B.2). When the Practical TPDP Algorithm requests an updating, the Stack Updating Procedure performs it using the contents of the stack. Appendix Section B.2 fully describes the Practical TPDP Algorithm, and Appendix Section B.3 fully describes the Stack Updating Procedure it calls.

The general operation of the Practical TPDP Algorithm is as follows. The Practical TPDP controller can be in one of three exploration modes. The mode in effect at any time is identified by the variable *explore* as 'none', 'internal' or 'external'. When no exploration is occurring actions are randomly chosen from those specified by the TPs at the states encountered. The immediate costs incurred when those actions are taken are observed, and the Q-values, R-values and weights of the TPs specifying those actions are updated accordingly. Internal and external exploration (see Section 4.2.3) are randomly initiated in the midst of this process and are allowed to continue until a TP state is encountered. Internal and external exploration facilitate the allocation of new TPs that can be further assessed.

[4]A trial can also be terminated by a mechanism that ensures that all of the states are continually visited, as explained in Section 3.2.2, but such terminations will not be dealt with in this Chapter.

```
1.  randomly choose starting state $s_0 \in S_S$
2.  choose a starting action $u_0 \in U(s_0)$
3.  if (state $s_0$ has a TP): 'none' $\Rightarrow$ $explore$
4.  otherwise: 'external' $\Rightarrow$ $explore$
5.  $0 \Rightarrow t$, $0 \Rightarrow t_{\text{update}}$, $0 \Rightarrow t_{\text{last-TP}}$
10. while $s_t$ is not an absorbing state $S_A$:
11.     if (state $s_t \neq s_{t-T}$) or (state $s_t = s_{t-T}$ for time $T_{\text{stuck}}$):
20.         if (state $s_t$ has a TP):
21.             if ($\sigma_{\text{swap-TP}} > 0$):
22.                 update-stack($explore$, $t$, $Q(s_t, \mu(s_t))$, $V_{\text{expected}}$)
23.                 'internal' $\Rightarrow$ $explore$
24.                 randomly choose action $u_t \in U(s_t)$
25.                 push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
26.             otherwise if ($explore \neq$ 'none') or ($t > t_{\text{update}}$):
27.                 update-stack($explore$, $t$, $Q(s_t, \mu(s_t))$, $V_{\text{expected}}$)
28.                 'none' $\Rightarrow$ $explore$
29.                 $t + \sigma_{\text{delay}} \Rightarrow t_{\text{update}}$
30.                 randomly choose action $u_t$ from TP actions in $U(s_t)$
31.                 push-on-stack($t$, $s_t$, $u_t$, 'true')
32.             otherwise: push-on-stack($t$, $s_t$, $u_t$, 'false')
33.             $Q(s_t, \mu(s_t)) \Rightarrow V_{\text{expected}}$, $Q(s_t, \mu(s_t)) \Rightarrow V_{\text{last-TP}}$
34.             $t \Rightarrow t_{\text{last-TP}}$
40.         if (state $s_t$ has no TP) and ($explore \neq$ 'none') and ($\sigma_{\text{change}} > 0$):
41.             if ($explore$ = 'external'): flush-stack
42.             randomly choose action $u_t \in U(s_t)$
43.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
50.         if (state $s_t$ has no TP) and ($explore$ = 'none') and ($\sigma_{\text{add-TP}} > 0$):
51.             update-stack('none', $t_{\text{last-TP}}$, $V_{\text{last-TP}}$, $\emptyset$)
52.             if ($\sigma_{\text{external}} > 0$): 'external' $\Rightarrow$ $explore$
53.             otherwise: 'internal' $\Rightarrow$ $explore$
54.             randomly choose action $u_t \in U(s_t)$
55.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
60.     $t + T \Rightarrow t$
61.     $\frac{1}{\gamma}(V_{\text{expected}} - c(s_t, u_t)) \Rightarrow V_{\text{expected}}$
62.     observe system for new state $s_t$
63. update-stack($explore$, $t$, $s_t$, $V_{\text{expected}}$)
```

Figure 4.3: The Practical TPDP Algorithm

```
                [Parameters passed: $explore$, $t$, $V_{\text{update}}$, $V_{\text{expected}}$]
1.  while (time at top of stack $t_s \geq t$): pop-off-stack($t_s, i_s, u_s$, $specified_s$)
2.  $V_{\text{update}} \Rightarrow C_{\text{total}}$
10. while (there are entries in stack):
11.     pop-off-stack($t_s, i_s, u_s$, $specified_s$)
12.     while ($t > t_s$):
13.         $\gamma C_{\text{total}} + c(i_s, u_s) \Rightarrow C_{\text{total}}$
14.         $t - T \Rightarrow t$
20.     if ($explore$ = 'none'):
21.         for (each TP action $u \in U(i_s)$ in state $i_s$):
22.             if ($u = u_s$):
23.                 $(1-\alpha)Q(i_s, u_s) + \alpha C_{\text{total}} \Rightarrow Q(i_s, u_s)$
24.                 if ($Q(i_s, u_s) < R(i_s, u_s)$):
25.                     $w(i_s, u_s) + 1 \Rightarrow w(i_s, u_s)$
26.                     if ($w(i_s, u_s) > w_{\max}$): $w_{\max} \Rightarrow w(i_s, u_s)$
27.                 otherwise:
28.                     $w(i_s, u_s) - 1 \Rightarrow w(i_s, u_s)$
29.                     if ($w(i_s, u_s) = 0$): remove the TP
30.             if [(state $i_s$ has only one TP) and ($specified_s$ = 'false')] or
                [($u \neq u_s$) and (another TP at state $i_s$ specifies $u_s$)]:
31.                 $(1-\alpha)R(i_s, u) + \alpha C_{\text{total}} \Rightarrow R(i_s, u)$
32.         for (each TP action $u \in U(i_s)$ in state $i_s$):
33.             if [($w(i_s, \mu(i_s)) < w_{\text{thr}}$) and ($w(i_s, u) > w(i_s, \mu(i_s))$)] or
                [($w(i_s, \mu(i_s)) \geq w_{\text{thr}}$) and ($w(i_s, u) \geq w_{\text{thr}}$) and
                ($Q(i_s, u) < Q(i_s, \mu(i_s))$)]: $u \Rightarrow \mu(i_s)$
40.     if (state $i_s$ has TPs) and (memory can be allocated) and
        ($explore$ = 'internal') and ($C_{\text{total}} < Q(i_s, \mu(i_s))$):
41.         allocate a new TP at state $i_s$ with action $u_s$
42.         $C_{\text{total}} \Rightarrow Q(i_s, u_s)$, $C_{\text{total}} \Rightarrow R(i_s, u_s)$
43.         $w_{\text{initial}} \Rightarrow w(i_s, u_s)$
44.         if ($w(i_s, u_s) > w(i_s, \mu(i_s))$) or $w(i_s, u_s) \geq w_{\text{thr}}$): $u_s \Rightarrow \mu(i_s)$
50.     if (state $i_s$ has no TPs) and (memory can be allocated) and
        [(($explore$ = 'internal') and $V_{\text{update}} < V_{\text{expected}}$)) or ($explore$ = 'external')]:
51.         allocate a new TP at state $i_s$ with action $u_s$
52.         $C_{\text{total}} \Rightarrow Q(i_s, u_s)$, $C_{\text{total}} \Rightarrow R(i_s, u_s)$
53.         $w_{\text{initial}} \Rightarrow w(i_s, u_s)$
54.         $u_s \Rightarrow \mu(i_s)$
```

Figure 4.4: The Stack Update Procedure

4.2.5 Delayed Updating

It was found, during experimental work with Practical TPDP, that updating the stack as soon as a new TP was encountered (see Equations 3.18 and 3.27) resulted in poor performance. This was because the TP states were frequently close together, and one was often encountered soon after another had been left. This resulted in the specified action being changed after only brief intervals, which prevented observation of the costs which each action specification would incur if they had time to truly affect the system. As explained in Section 3.1.4, actions specified for short durations cannot really affect systems that have inertia. In response to this, *delayed updating* was incorporated into Practical TPDP. Delayed updating prevents updating for a reasonable period of time, facilitating Q-value and R-value updating with longer term cost consequences.

Delayed updating is facilitated with the use of an *update time* t_{update} (see Figure 4.3). Instead of simply applying a TP action at a given TP state and updating the Q-values, R-values and weights of that TP as soon as another TP state is encountered, t_{update} is used to maintain the action specified by the TP and delay updating. The delay may continue as state transitions are made through many TP states. This is in fact the primary reason why a stack is employed in Practical TPDP[5]. The stack records the information related to all of the TPs that are encountered in the period before the update time t_{update} has been reached.

The update time t_{update} is set (see Appendix Section B.2, Lines 20 to 34) by adding a random positive *delay parameter* σ_{delay} to the current time step t. Section 4.2.6 describes how the random distribution of σ_{delay} is determined. Once t_{update} is set, stack updating and changes in the action specification are prevented until t exceeds t_{update} (unless internal or external exploration is initiated).

[5]The stack is also used to record all of the action specifications made during internal exploration.

Update times are not used during internal and external exploration because all exploration is terminated as soon as a TP state is encountered. This is done because internal and external exploration are not performed to assess existing TPs. Exploration is performed to determine if new TPs should be allocated for future assessment, and this determination can only be made based on the evaluation function values of the TP states at the start and end of the exploratory transition sequences through the state space (see Appendix Section B.3). In other words, all exploration takes place during transition sequences between TP states.

If a given TP is the only TP associated with a state, and updating is not occasionally delayed when the system enters that state, then the action associated with that TP will always be chosen for specification when that state is entered. This will preclude all updating of the R-value of that TP (see Appendix Section B.3). This is another reason why delayed updating is necessary.

4.2.6 The Practical TPDP Exploration Parameters

Of the various parameters in the Practical TPDP Algorithm that must be adjusted to ensure effective algorithm performance, the exploration parameters were found during experimental work to be the most important (see Chapter 5). This Section will describe these parameters, and present guidelines for their determination.

The Delay Parameter

As explained in Section 4.2.5, σ_{delay} is a random positive delay parameter added to t to determine the next update time t_{update}. Basically, t_{update} determines how often updates should occur. The distribution of σ_{delay} can be uniform or Gaussian[6], with the mean $\bar{\sigma}_{\text{delay}}$ entirely depending on the system being considered. The value of $\bar{\sigma}_{\text{delay}}$ must be set

[6]With negative values being made equal to zero.

high enough to ensure that the long term consequences of the action specified by each TP can be observed in the system under consideration (see Section 4.2.5). It must not be set so high however that the action specified by each TP is frequently maintained well beyond the boundaries of the uniform region (see Section 3.1.1) that it is best suited for. This would result in the costs experienced after actions are specified being unnecessarily high.

Generally, $\bar{\sigma}_{\text{delay}}$ is related to the natural modes of the system being controlled. If the action specification must change after an average interval of $\bar{T}_{\text{uniform}}$ for a given system to be optimally controlled (that is, it takes an average time of $\bar{T}_{\text{uniform}}$ to cross each uniform region), then Practical TPDP is most effective when:

$$\bar{\sigma}_{\text{delay}} = \bar{T}_{\text{uniform}} \tag{4.42}$$

A rough value for $\bar{T}_{\text{uniform}}$ can be determined by considering the general response characteristics of the system concerned. A rough value is sufficient, as Practical TPDP is not overly sensitive to the setting of this parameter (see Section 5.5).

The Swap Parameter

The random *swap parameter* $\sigma_{\text{swap-TP}}$ (see Appendix Section B.2), which has some fixed probability $\Pr(\sigma_{\text{swap-TP}} > 0)$ of being greater than zero each time it is considered, determines whether or not internal exploration is initiated at TP states. This probability must be set low enough to ensure frequent updates of the TPs already associated with each state, and high enough to facilitate thorough exploration of alternative TP possibilities at each state. Experimental work with Practical TPDP (see Chapter 5) indicated that a 0.2 probability of $\sigma_{\text{swap-TP}}$ being greater than zero generally produced good results – that is, internal exploration should be initiated once every five times that a TP is encountered.

The Initiate Parameter

The random *initiate parameter* $\sigma_{\text{add-TP}}$ (see Appendix Section B.2), which has some fixed probability $\Pr(\sigma_{\text{add-TP}} > 0)$ of being greater than zero each time it is considered, determines whether or not exploration (internal or external) is initiated at non-TP states. This probability must be set low enough to ensure that the updating of existing TPs is not constantly interrupted, and high enough to facilitate thorough exploration of new TP addition possibilities. As $\sigma_{\text{add-TP}}$ is considered every time step during which the system is not in a TP state, the cumulative probability of exploration (internal or external) being initiated between TP states must be adjusted to be appropriate for the system under consideration.

Experimental work with Practical TPDP (see Chapter 5) indicated that the best results were obtained when the cumulative probability of exploration was 0.5 or higher when the system passed through the first non-TP state encountered after leaving a TP state. This facilitated the addition of new TPs that could subtly refine the boundary state locations (see Section 3.1.1). By employing d_{quick}, the number of time steps required to pass through a single state at the highest system velocity, a formula for $\Pr(\sigma_{\text{add-TP}} > 0)$ can thus be determined:

$$\begin{aligned} 0.5 &= 1 - [1 - \Pr(\sigma_{\text{add-TP}} > 0)]^{d_{\text{quick}}} \\ \Pr(\sigma_{\text{add-TP}} > 0) &= 1 - .5^{1/d_{\text{quick}}} \qquad (4.43) \end{aligned}$$

Using Equation 4.43 to determine a value for $\Pr(\sigma_{\text{add-TP}} > 0)$ can result in the updating of existing TPs constantly being interrupted early in the learning process when the TPs are well spaced out. At this time refinement of the boundary locations is unnecessary and undesirable. A method of varying $\Pr(\sigma_{\text{add-TP}} > 0)$ as learning progresses would therefore be valuable. Such a method was not developed during this work however.

The External Parameter

Once internal or external exploration is initiated at non-TP states (based on the random parameter $\sigma_{\text{add-TP}}$), the probability $\Pr(\sigma_{\text{external}} > 0)$ of the random *external parameter* σ_{external} being greater than zero determines whether or not external exploration is initiated instead of internal exploration (see Appendix Section B.2). Experimental work with Practical TPDP (see Chapter 5) indicated that a 0.5 probability of σ_{external} being greater than zero generally produced good results – that is, when exploration is initiated at non-TP states, internal and external exploration should be chosen with equal likelihood.

The Route Change Parameter

During internal and external exploration, the probability $\Pr(\sigma_{\text{change}} > 0)$ of the random *route change parameter* σ_{change} being greater than zero determines whether or not a new experimental action is specified during each time step (see Appendix Section B.2). As σ_{change} is considered every time step during exploration, the cumulative probability of an action change must be adjusted so that action changes are made at intervals that reflect the ability of the system concerned to respond.

Like the delay parameter $\bar{\sigma}_{\text{delay}}$, $\Pr(\sigma_{\text{change}} > 0)$ is related to the natural modes of the system being controlled. Experimental work with Practical TPDP (see Chapter 5) indicated that the best results were obtained when there was a 0.5 cumulative probability of an action change occurring after the same action had been specified for one fifth of $\bar{T}_{\text{uniform}}$, the average time required to cross each uniform region:

$$\begin{aligned} 0.5 &= 1 - [1 - \Pr(\sigma_{\text{change}} > 0)]^{\bar{T}_{\text{uniform}}/(5T)} \\ \Pr(\sigma_{\text{change}} > 0) &= 1 - .5^{5T/\bar{T}_{\text{uniform}}} \end{aligned} \tag{4.44}$$

4.2.7 The Other Practical TPDP Parameters

The Learning Rate

The learning rate α should be set high enough to facilitate some learning, but not so high that the results from each learning experience can drastically alter what has already been learned. Learning rate values are commonly set around 0.3, and this value was found to produce good results during experimental work with Practical TPDP (see Chapter 5). There is no reason why this value should not produce equally good results in any application of Practical TPDP.

The Weight Parameters

The weight parameters must be set so that when a new TP is added to the assessment group with an initial weight of w_{initial}, that TP will be quickly removed if its R-value is thereafter less than its Q-value (see Section 4.2.1). The threshold value w_{thr} above which a TP can be considered as a potential policy TP for a given state (see Section 4.2.2) must be set high enough so that a newly added TP must be updated a number of times before it can reach that threshold. The maximum weight value $w_{\max}$ must be set so that the weight of a TP can be quickly reduced below w_{thr} if a system change reduces its merit (see Section 4.2.1). Experimental work with Practical TPDP (see Chapter 5) indicated that the following weight parameter values produced good results:

$$\begin{aligned} w_{\text{initial}} &= 2 \\ w_{\text{thr}} &= 10 \\ w_{\max} &= 20 \end{aligned}$$

There is no reason why these values should not produce equally good results in any application of Practical TPDP.

The Same State Limit Parameter

The parameter $T_{\mathbf{stuck}}$ is used to initiate exploration when the system has been "stuck" in the same state for too long (see Appendix Section B.2). It should be set so that $T_{\mathbf{stuck}}$ is a time period longer than that for which the system under consideration would normally be expected to stop.

Chapter 5

Application of Practical TPDP

This Chapter will describe the application of Practical TPDP to two different control tasks. These applications allow the characteristics of Practical TPDP to be analyzed and described.

5.1 The Race Track Problem

5.1.1 Description of the Problem

Gardner (1973) described a problem called Race Track which Barto *et al.* (1991) later modified for the application of DP controllers. As described by Barto *et al.* (1991), Race Track consists of a "track" like the one shown in Figure 5.5. A single "car" starts at the left of the track shown, accelerates down the length of it, and finishes after making a left turn. Each square in this track represents a position that the car can move to. The four positions where the car can start are the starting states S_S (with all velocities zero). The absorbing states S_A include all four finishing positions, with different velocities possible in each position. When one of the finishing positions is reached the problem is terminated.

Collisions with the walls are handled by leaving the car at the point of collision and setting its velocity to zero[1]. As explained by Barto *et al.* (1991), this implies that the car cannot leave the track except by reaching the finishing positions.

As the car moves down the track its horizontal and vertical accelerations can be

[1]Litigation is not a concern in this problem.

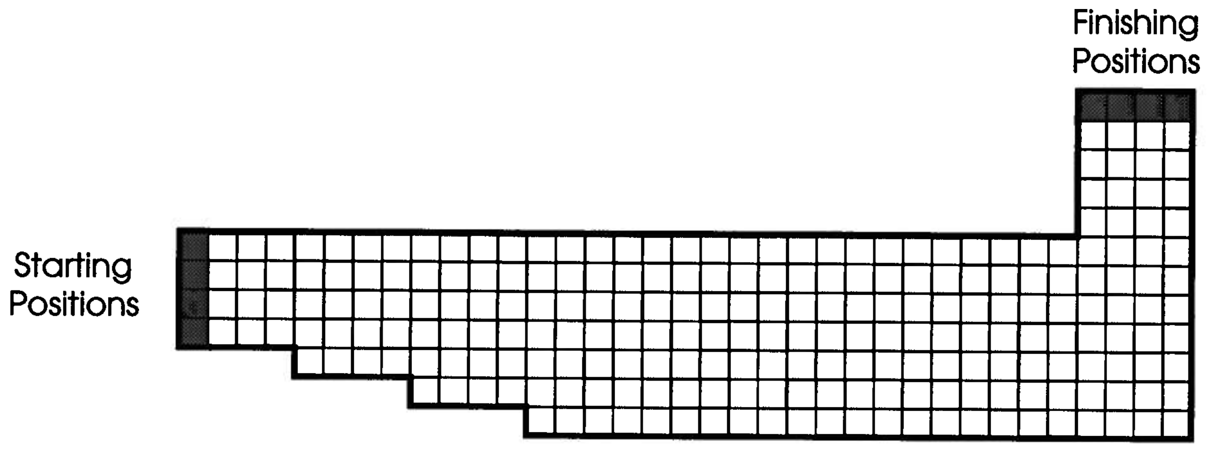


Figure 5.5: A Race Track

controlled, with the velocity of the car in both directions being limited. This limitation ensures that the state space S of the problem is finite (Barto *et al.*, 1991), although wall collisions will inherently limit the velocities possible on the track shown in Figure 5.5. Any acceleration specification has a probability $p_{\text{ineffective}}$ of having no effect on the car during the time step in which it is applied. This makes the problem stochastic.

The control task is to control the horizontal and vertical accelerations of the car so that it runs the length of the track in the minimum possible time. This optimality criteria is established by imposing the same immediate cost $c(i, u) = 1$ for all actions $u \in U(i)$ taken in each state i, with the exception of actions taken in the absorbing states S_A. All actions taken after reaching the absorbing states S_A, for all future time, result in immediate costs of $c(i, u) = 0$. The discount rate γ is set to 1. As a result, the lowest possible costs are incurred when the car runs the length of the track in the minimum possible time.

5.1.2 A Discrete-time Stochastic Dynamic System Formulation

Following Barto *et al.* (1991), the Race Track problem can be described as having a state space S consisting of states $s_t = [x_t, y_t, \dot{x}_t, \dot{y}_t]$, where x_t and y_t are the horizontal and vertical position of the car respectively at time step t. The possible actions $U(i)$ are the same for each state i and consist of $u = [u_x, u_y]$, where both u_x and u_y are in the set $\{-a, 0, a\}$, and where a is a constant. With velocity limitations of $\dot{x}_L$ and $\dot{y}_L$, the following state transition will occur at time step t with probability $1 - p_{\text{ineffective}}$:

$$\begin{aligned}
\dot{x}_{t+T} &= \max(-\dot{x}_L, \min(\dot{x}_L, (\dot{x}_t + u_{x_t}T))) \\
\dot{y}_{t+T} &= \max(-\dot{y}_L, \min(\dot{y}_L, (\dot{y}_t + u_{y_t}T))) \\
x_{t+T} &= x_t + \dot{x}_{t+T}T \\
y_{t+T} &= y_t + \dot{y}_{t+T}T
\end{aligned} \tag{5.45}$$

If the acceleration specifications have been ineffective (probability $p_{\text{ineffective}}$), the following state transition will occur at time step t:

$$\begin{aligned}
\dot{x}_{t+T} &= \dot{x}_t \\
\dot{y}_{t+T} &= \dot{y}_t \\
x_{t+T} &= x_t + \dot{x}_{t+T}T \\
y_{t+T} &= y_t + \dot{y}_{t+T}T
\end{aligned} \tag{5.46}$$

Given the state transitions described by Equations 5.45 and 5.46, if a, $\dot{x}_L$, $\dot{y}_L$, T and the initial state values $[x_0, y_0, \dot{x}_0, \dot{y}_0]$ are all integers, the state values will always remain integers.

5.1.3 A Continuous Version of the Problem

As explained in Section 3.1.4, TPDP performs best in continuous state spaces. Therefore, to fully demonstrate Practical TPDP, it was applied to a continuous version of the Race

Track problem. To make the problem continuous, the time step interval T in Equations 5.45 and 5.46 is made infinitesimally small.

When this is done there is no guarantee that the state values will remain integers. As a result, in order to control the car with a finite number of DP elements (see Section 2.3.2), the state values must be quantized into discrete intervals. For this application of Practical TPDP the state values were simply rounded-off to the nearest integer.

Making the state space continuous changes the total infinite-horizon discounted costs experienced as described in Section 3.4.5. Equations 3.37 and 3.38 were used to compensate for these effects during Practical TPDP solution of the Race Track problem.

By making the time step interval T small, and thereby making the Race Track problem continuous, the system state is prevented from drastically changing between time steps. In the discrete-time ($T = 1$) version of the problem considered by Barto *et al.* (1991) the car velocity could instantaneously change, and the car could move from one position to another one many squares away without passing through the positions in between. If the time step interval T is made small, state space transitions are restricted to occur only between neighboring states, and both of these behaviors are eliminated. As explained in Section 3.1.4, this is necessary for the effective operation of TPDP. If drastic state changes are possible, uniform region boundaries cannot be established.

5.1.4 Problem Specifics Used During Practical TPDP Application

The track used for Practical TPDP application was the one used by Barto *et al.* (1991). It is the track shown in Figure 5.5. Continuous movement was approximated by setting $T = 0.05$. This setting was small enough to ensure that state transitions could only be made between neighboring states, but large enough to ensure that the problem simulation time did not become exorbitant.

With T set to 0.05 on this track the system can enter roughly 15440 discretized states.

The car was started in one of the four starting positions with zero velocity, each of which was chosen with equal probability.

When Practical TPDP was applied to the Race Track problem, the various problem parameters were generally set to the values used by Barto *et al.* (1991):

$$\begin{aligned}
\gamma &= 1 \quad \text{(this value is unaffected by the } \gamma \text{ compensation Equation 3.38)} \\
c(i,u) &= \begin{cases} 0 & \text{for all actions taken in the absorbing states } S_A \\ 3 & \text{for any actions that lead to a wall collision} \\ 1\frac{0.05}{1} = 0.05 & \text{for all other actions (based on Equation 3.37)} \end{cases} \\
p_{\text{ineffective}} &= 0.1 \\
a &= 1 \\
\dot{x}_L &= 6 \\
\dot{y}_L &= 6
\end{aligned}$$

As indicated, the immediate cost $c(i,u)$ incurred when an action lead to a wall collision was set to 3. In the discrete-time ($T = 1$) version of this problem considered by Barto *et al.* (1991), no extra cost was incurred other than that which resulted from the additional time required to accelerate the car again. When the problem was made continuous, the collision cost of 3 was used because the ability of the car to accelerate was changed. This change lead to the uninteresting result that the total cost incurred was roughly the same whether the car swept smoothly around the left turn or whether it accelerated fully into the right wall, collided with it, and then accelerated again to reach the finishing positions. The collision cost of 3 made the latter strategy much less attractive.

5.1.5 The Practical TPDP Algorithm Parameters Used

During solution of the Race Track problem, the Practical TPDP Algorithm parameters (see Sections 4.2.1, 4.2.5, 4.2.6, 4.2.7, and Figures 4.3 and 4.4) were set to:

$$\begin{aligned}
\bar{\sigma}_{\text{delay}} &= 2.5 \\
\Pr(\sigma_{\text{swap-TP}} > 0) &= 0.2 \\
\Pr(\sigma_{\text{add-TP}} > 0) &= 0.25 \\
\Pr(\sigma_{\text{external}} > 0) &= 0.5 \\
\Pr(\sigma_{\text{change}} > 0) &= 0.1 \\
\alpha &= 0.3 \qquad \forall\ t, i, u \\
w_{\text{initial}} &= 2 \\
w_{\text{thr}} &= 10 \\
w_{\text{max}} &= 20 \\
T_{\text{stuck}} &= 1
\end{aligned}$$

The random parameter σ_{delay} had a uniform distribution from 0 to 5 (see Section 4.2.6). The values for $\bar{\sigma}_{\text{delay}}$ and $\Pr(\sigma_{\text{change}} > 0)$ were based on a $\bar{T}_{\text{uniform}}$ value of 2.5 being used in Equations 4.42 and 4.44 respectively. The value for $\Pr(\sigma_{\text{add-TP}} > 0)$ was based on a d_{quick} value of 3 being used in Equation 4.43. The rest of the parameter values were chosen based on preliminary experimentation.

5.1.6 Evaluation of Performance on the Problem

It was not expected that Practical TPDP would learn absolutely optimal policies for the Race Track problem – this would have taken far too long. Instead near optimal policies were sought. To evaluate the performance of Practical TPDP as it learned such near optimal policies, successive trials were simulated that included one run of the car from the starting positions to the finishing positions. After every set of 20 such learning trials, 500 test trials were simulated where Practical TPDP learning was suspended and the policy learned up to that point was employed to control the car. The average track time of the car down the track during the 500 test trials was used as an indication of how well the policy had been learned. The combination of 20 learning trials followed by 500 test trials was called an *epoch*, and it was also used by Barto *et al.* (1991).

All test trials were terminated in which a total cost of more than 500 was experienced before the car arrived at the finishing positions. Further, during testing, only actions specified by policy TPs (see Section 4.1.2) with weights $w(i, u)$ greater than w_{initial} were applied to the system.

5.2 Performance on the Race Track Problem

5.2.1 Comparing Practical TPDP and Conventional Q-learning

Figure 5.6 shows the average track time results (see Section 5.1.6) when both Practical TPDP and conventional Q-learning were applied to a continuous version (see Section 5.1.3) of the Race Track problem. As explained in Section 5.1.4, the track shown in Figure 5.5 was used, as well as the problem parameters presented in that Section. The Practical TPDP Algorithm parameters used were those presented in Section 5.1.5. The parameters and exploration method used for conventional Q-learning were those employed by Barto *et al.* (1991) on this same track during their discrete-time ($T = 1$) experiments.

Figure 5.6 shows that Practical TPDP learned a near optimal policy much faster than conventional Q-learning. This comparison is somewhat inequitable however, because the conventional Q-learning approach used (Barto *et al.*, 1991) is best suited for discrete-time ($T = 1$) applications. When the time step interval T is made small enough to make the problem approximately continuous (see Section 3.4.5), state transitions are made only between neighboring states. As a result, Q-value updating is performed by conventional Q-learning using only the immediate costs experienced after short periods of time. This makes learning a near optimal policy very difficult (see Section 4.2.5). This problem was solved in Practical TPDP by making use of delayed updating and its randomly chosen update time $t_{\mathbf{update}}$ (see Section 4.2.5). In order to make the comparison between Practical TPDP and conventional Q-learning more equitable, this delayed updating approach was added to conventional Q-learning.

The average track time results after delayed updating was incorporated into conventional Q-learning are presented in Figure 5.7. Figure 5.7 shows that the performance of conventional Q-learning on the continuous Race Track problem was indeed improved by adding delayed updating to it, but that Practical TPDP still learned a near optimal policy considerably faster. The reason for this, as explained in Section 3.1.3, is that conventional Q-learning associates a DP element with every possible state and action combination. Practical TPDP only associates DP elements (in the form of TPs) with a fraction of this total, and thus enjoys the benefit of not having to make DP computations for all of them.

Considering total computational effort, to complete 1500 epochs (with no test trials) on the Race Track problem Q-learning without delayed updating required 983 CPU seconds on a Sun SPARCstation 10, Q-learning with delayed updating required 229 CPU seconds, and Practical TPDP required 528 CPU seconds. As explained, Q-learning without delayed updating is not well suited to application in such continuous state spaces, and the protracted learning times which result necessitate more computational effort. More computational effort was required for Practical TPDP than for Q-learning with delayed updating because its computations are more complex (see Chapter 4). In real-time control applications however, the learning time, not the total computational effort,

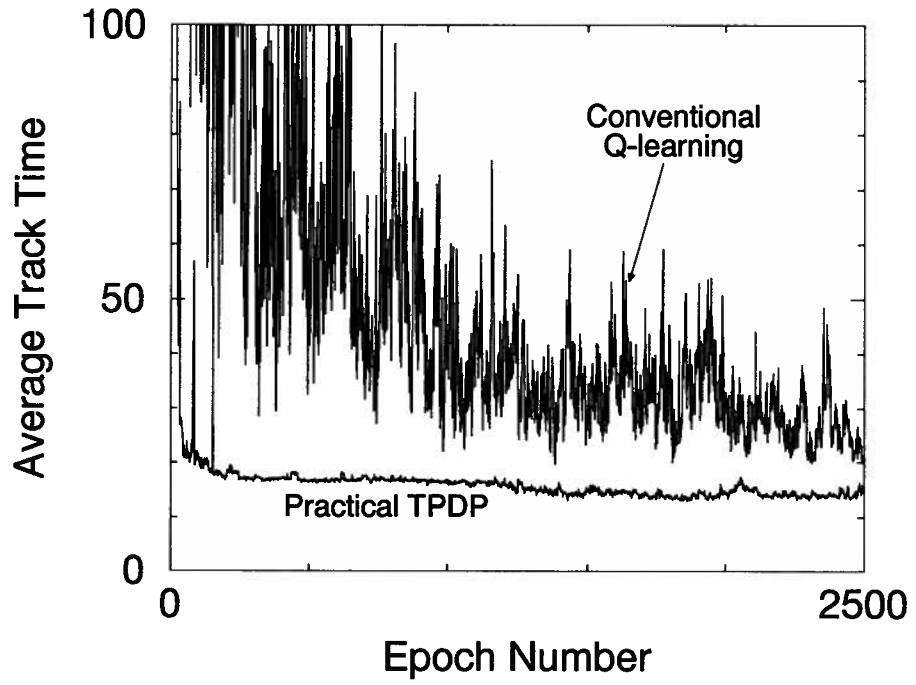


Figure 5.6: Performance of Practical TPDP and Conventional Q-learning

is likely to be the larger concern.

Finally, as a matter of interest, Figure 5.8 shows the average track time results when both Practical TPDP and conventional Q-learning were applied to the discrete-time version of the Race Track problem considered by Barto *et al.* (1991)[2]. The average track time performance of conventional Q-learning in this case was, not surprisingly, very similar to that obtained by Barto *et al.* (1991). As shown in Figure 5.8, Practical TPDP again learned a near optimal policy much faster than conventional Q-learning. One fact is not clear in Figure 5.8 however. This is that, even though Practical TPDP settled on a policy much sooner than conventional Q-learning, the average track time that resulted from that policy remained roughly constant, and it was slightly higher than the average track time that conventional Q-learning was able to achieve after 2000 epochs.

[2]The immediate cost schedule used by Barto *et al.* (1991) on this problem was used as well.

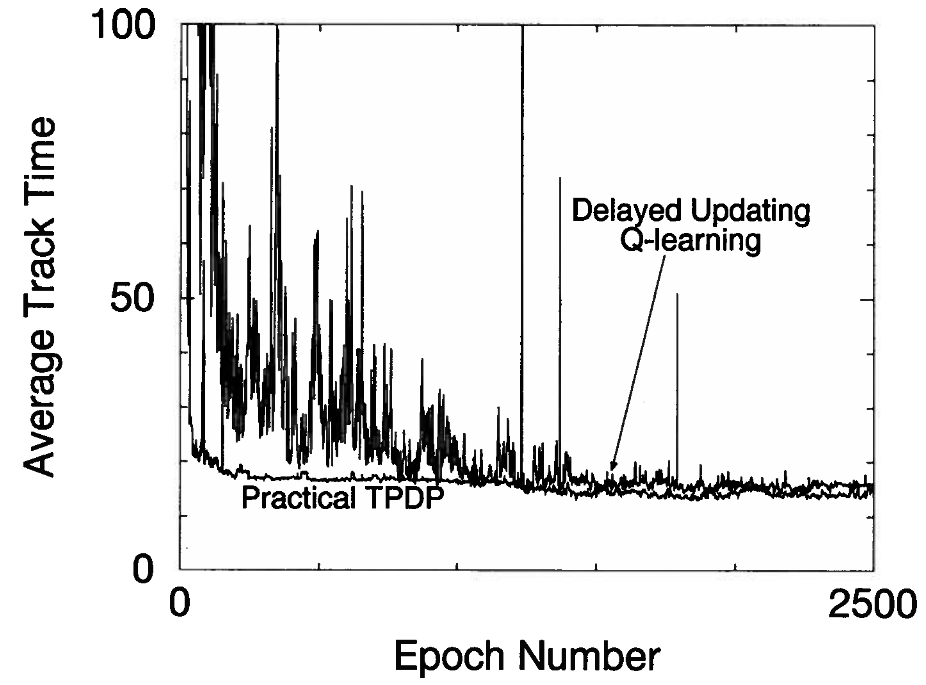


Figure 5.7: Performance of Practical TPDP and Delayed Updating Q-learning

Undoubtedly this reflects an inability of Practical TPDP to learn an absolutely optimal policy in this discrete-time application. This is not a concern however, as Practical TPDP was not developed to operate in such discrete-time applications (see Section 3.1.4).

5.2.2 Near Optimal Performance of Practical TPDP

Figure 5.9 again shows the average track time results when Practical TPDP was applied to the Race Track problem. This Figure indicates that after Practical TPDP settled on an initial average track time of around 17, and maintained that track time for roughly 600 epochs, a *learning transition* occurred that reduced the average track time to around 14. To illustrate why this distinct learning transition occurred, Figures 5.10, 5.11, 5.12 and 5.13 show five typical car paths down the length of the track after 300, 800, 1300 and 1800 epochs respectively of Practical TPDP learning.

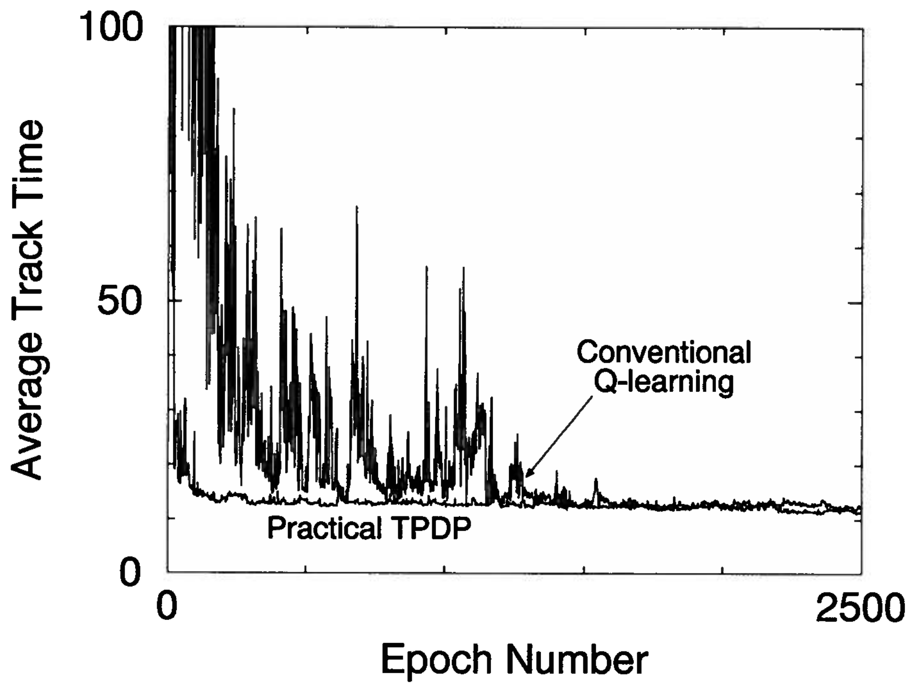


Figure 5.8: Performance on a Discrete Version of the Problem

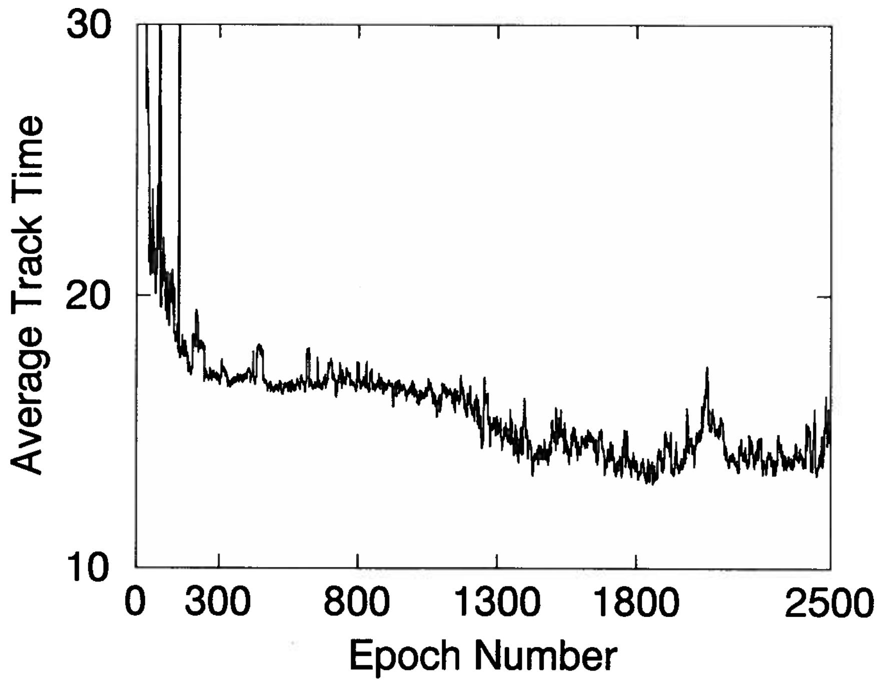


Figure 5.9: Performance of Practical TPDP

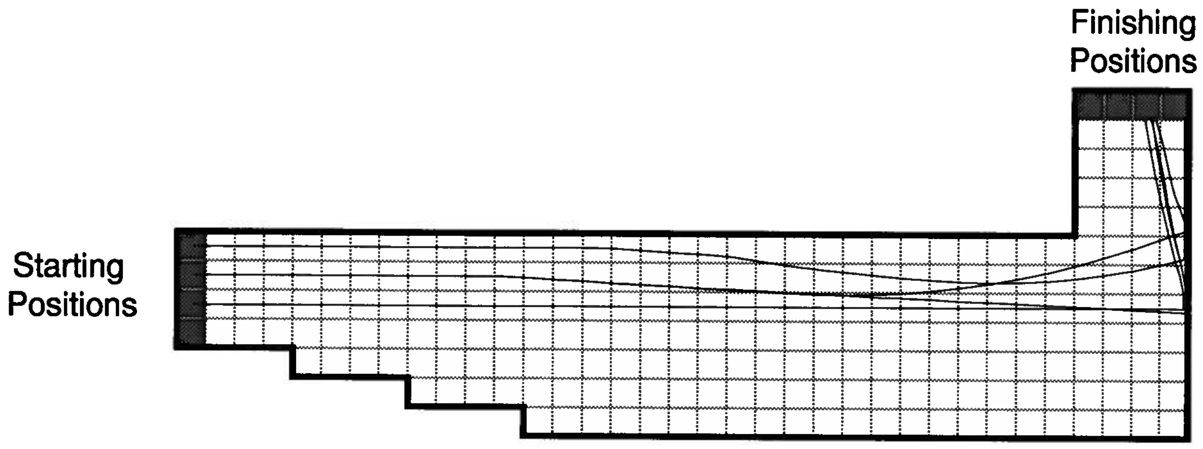


Figure 5.10: Five Typical Race Track Routes After 300 Epochs

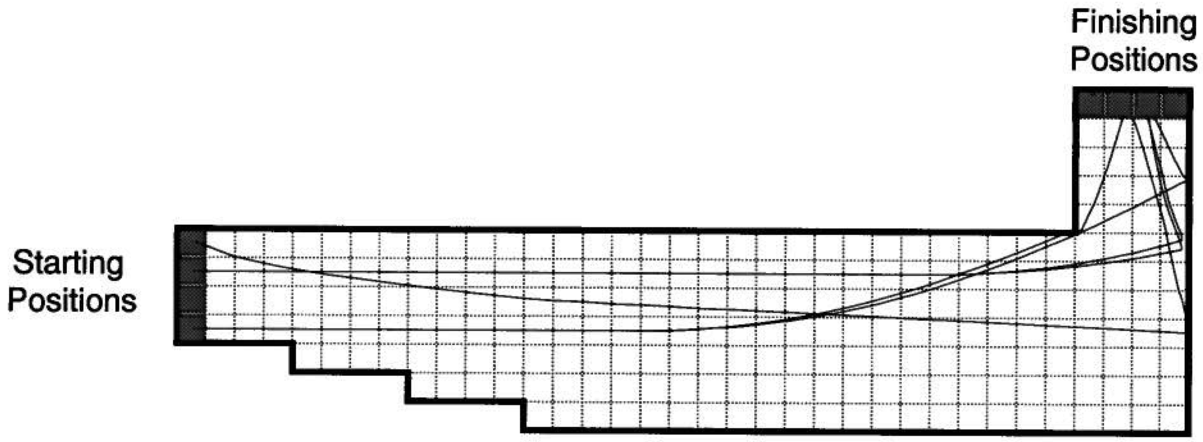


Figure 5.11: Five Typical Race Track Routes After 800 Epochs

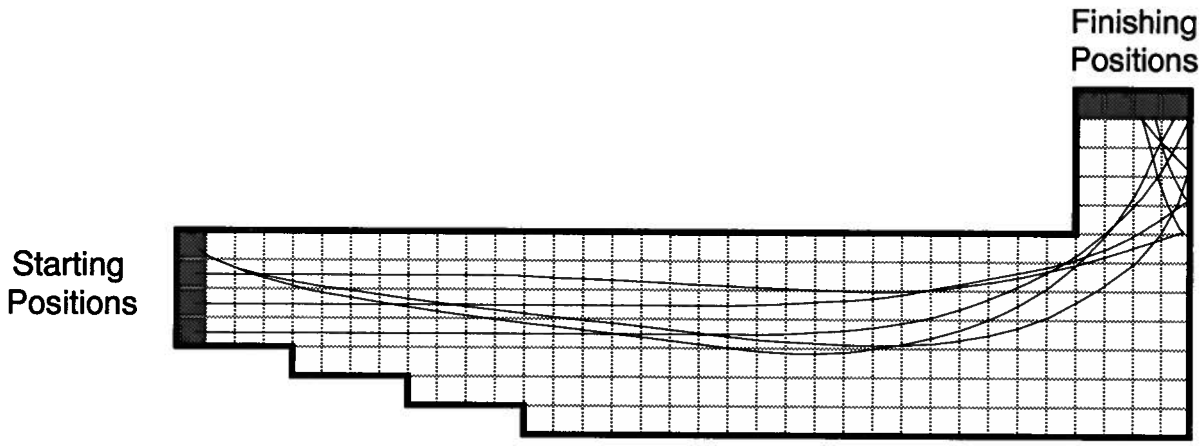


Figure 5.12: Five Typical Race Track Routes After 1300 Epochs

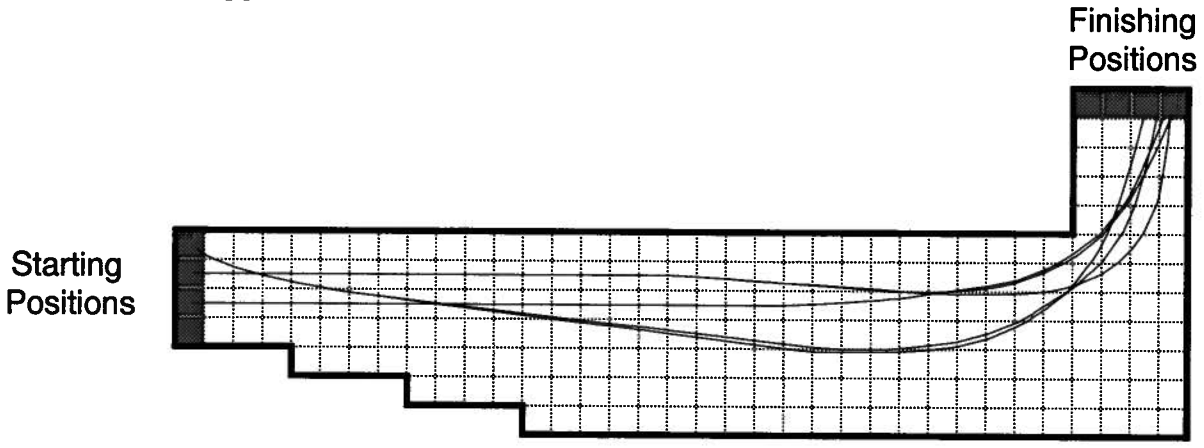


Figure 5.13: Five Typical Race Track Routes After 1800 Epochs

After 300 epochs (Figure 5.10) Practical TPDP had learned the policy of fully accelerating the car down the track, colliding with the right wall, and then accelerating the car towards the finishing positions. After 800 epochs (Figure 5.11) Practical TPDP was still employing this same basic policy, but it had started learning to curve the car path upward. This reduced the time required to reach the finishing positions once the right wall had been struck. Figure 5.11 also shows the path taken by a car when it recovered from a collision with the top wall after attempting to curve upward too much.

After 800 epochs the learning transition began. The learning transition consisted of Practical TPDP learning that if the car was not accelerated horizontally to full speed (full speed is 6, see Section 5.1.4) while moving towards the right wall, it could be accelerated upward beginning at roughly the track midpoint to produce a smooth left turn into the finishing positions. By making such a turn the car could avoid collision with any walls, and reach the finishing positions in the shortest possible time. Figure 5.12 shows five typical car paths taken after 1300 epochs of learning, in the midst of the learning transition. Some paths included smooth turns; others were like the collision paths taken after 800 epochs. After 1800 epochs the learning transition was complete. Figure 5.13 shows that smooth turns were consistently made at this stage of Practical TPDP learning.

The learning transition described illustrates fully the exploration issues discussed in Section 4.2.3. Basically what occurred during the Practical TPDP learning process was that an initial low cost policy was learned quickly (that of fully accelerating, colliding with the right wall, and moving towards the finishing positions – at 300 epochs, see Figure 5.10). The TPs specifying this low cost policy defined a closed state space S_C. Once this policy had been learned, internal exploration within S_C lead to reductions in the costs experienced when it was followed (800 epochs, see Figure 5.11). As this occurred, external exploration resulted in the placement of ineffectual TPs that could guide the car along an even lower cost path (smoothly turning – 1800 epochs, see Figure 5.13). At around 1300 epochs enough such TPs were in place for a learning transition to be made.

Figure 5.9 shows that the near optimal policy was not consistently followed after about 2000 epochs. The average track time increased sporadically after 2000 epochs. This was due to the continued allocation of superfluous TPs, a problem which will be discussed in Section 5.4.2.

5.3 Practical TPDP and Generalization

5.3.1 Generalization by Copying

Section 2.4.4 described how the ability to generalize is often incorporated into DP controllers by parametrically approximating the evaluation function. It was further explained that such parametric approximation typically requires extensive computational effort for the approximation to be learned, and that parametric approximation is particularly difficult to incorporate into Q-learning controllers. This is because Q-values are a function of action as well as state, and an action dimension must also be included in any approximation that represents Q-values. As a result of all this, parametric approximation was never investigated in the development of Practical TPDP.

Another type of generalization is possible with Practical TPDP however. This is to bias the selection of the random actions attempted during exploration (see Section 4.2.4 and Appendix Section B.2) towards actions already specified by TPs in the state space vicinity around the state where the random action is being attempted. This type of generalization is called *generalization by copying.* Generalization by copying is highly localized in terms of the state space regions over which generalizations are made, but it can be extremely effective when used in Practical TPDP. This is because, when Practical TPDP is applied to the continuous state space systems to which it is best suited (see Section 3.1.4), it is likely that a TP that has been found to be worthwhile at one state will also be worthwhile if duplicated in neighboring states.

5.3.2 Generalization by Copying in the Race Track Problem

Generalization by copying was attempted during Practical TPDP solution of the Race Track problem. Whenever an action had to be randomly chosen during internal or external exploration, the states adjacent to the one where the random action was being attempted were inspected. If any of them had TPs, and the random *generalization parameter* $\sigma_{\mathbf{generalize}}$ was greater than zero, generalization by copying was performed. Specifically, an action was chosen, with equal probability for each choice, from all the actions specified by TPs in the adjacent states. Generalization by copying was only performed on random occasions (according to the random value of $\sigma_{\mathbf{generalize}}$) to prevent the internal and external exploration processes from losing too much vigor. That is, to ensure that they were not biased too frequently.

Figure 5.14 illustrates the results of attempting generalization by copying during Practical TPDP solution of the Race Track problem, with $\Pr(\sigma_{\mathbf{generalize}} > 0) = 0.2$. Figure 5.14 shows that generalization by copying can increase the rate at which Practical TPDP learns a near optimal policy.

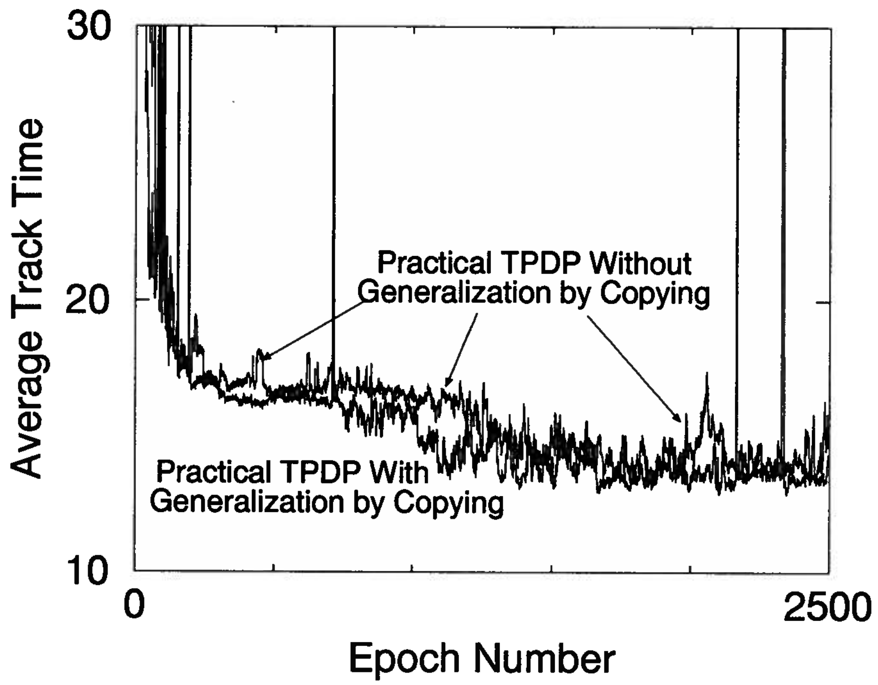


Figure 5.14: Performance of Practical TPDP With and Without Generalization

5.3.3 Practical TPDP Glitches on the Race Track Problem

Inspection of Figure 5.14 reveals that there are short duration glitches in the average track time at epochs 709, 2164 and 2328. While these glitches were the result of attempting generalization by copying during Practical TPDP, they appeared during many other experiments in which Practical TPDP was applied to the Race Track problem. They were caused by the removal of TPs associated with zero velocity wall states (states that had zero velocity and a position immediately next to a wall). Such TPs were crucial because they specified actions that started the car moving again once it had collided with a wall. When they were removed during Practical TPDP learning, and the car entered the states they were associated with during test trials, the car would remain in those states until the test trial was aborted. This lack of movement resulted in large costs being incurred, which produced the glitches seen in Figure 5.14.

Glitches did not result when TPs were removed from other states because the car would continue to move through those states whether TPs specified new actions in them or not. Thus, it was the definition of the Race Track problem that lead to the occurrence of the glitches. If the cars had bounced off the walls for example, instead of stopping completely, the glitches would not have occurred. As they were a result of the Race Track problem definition, glitches also occurred when conventional Q-learning was applied to the Race Track problem. In the case of conventional Q-learning, they resulted when the policy action at the zero velocity wall states did not accelerate the car away from the wall. Figures 5.6 and 5.8 show many glitches in the average track time resulting from conventional Q-learning. In fact, it was the rapid convergence of Practical TPDP to a near optimal policy that makes the Practical TPDP glitches appear so onerous. The glitches rise up in stark contrast to the generally settled average track time curve.

Any number of techniques could be employed to prevent glitches in the Practical TPDP solution of the Race Track problem. TP removal could be prevented in zero velocity states for example. No such techniques were investigated however, as the problem was not considered to be particularly serious. It was mainly a result of the peculiarities of the Race Track problem definition, not a result of some fundamental failing of Practical TPDP. Further, the rapid recovery of Practical TPDP when glitches occurred was seen as further evidence of the viability of the TPDP approach.

5.3.4 A Performance Metric

To determine the level of generalization by copying ($\Pr(\sigma_{\text{generalize}} > 0)$) that produced the best results during Practical TPDP solution of the Race Track problem, a *performance metric* was developed that facilitated quantitative comparison between the results obtained when different versions of Practical TPDP were employed. This performance metric was designed to indicate when the learning transition had been made (see Section

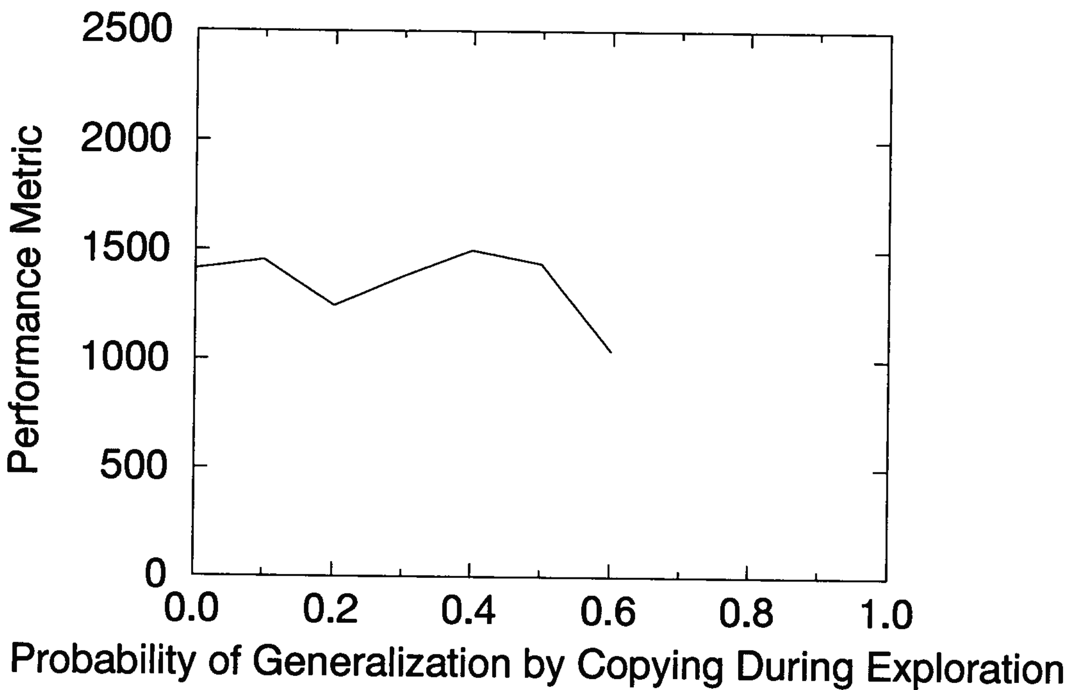


Figure 5.15: The Effect of Changing $\Pr(\sigma_{\text{generalize}} > 0)$

5.2.2). That is, when Practical TPDP had learned to smoothly turn the car, as opposed to colliding with the right wall. The performance metric was based on the first 100 test trials in which the average track time was less than 14. The average number of epochs required for the first 100 such track times was used as the performance metric.

5.3.5 Comparing Generalization Levels With the Performance Metric

Using the performance metric described in Section 5.3.4, Figure 5.15 was generated to illustrate the performance results from different levels of generalization by copying. That is, Figure 5.15 illustrates the results of varying the probability that the random parameter $\sigma_{\text{generalize}}$ was greater than zero. The portions of the curve which are missing from Figure 5.15 indicate Practical TPDP solutions of the Race Track problem where the average track time was not found to be less than 14 at least 100 times within 2500 epochs.

It was found that generalization by copying had the general effect of increasing the rate at which an existing policy was refined. That is, generalization by copying aided the internal exploration process in determining the lowest cost TPs for the closed state space S_C (see Section 4.2.3). It did so by focusing the internal exploration on action specifications that were already known to have merit. However, by restricting exploration to the actions already being specified in the various regions of S_C, generalization by copying inhibited external exploration and the discovery of alternative low cost routes through the state space (see Section 4.2.3). External exploration inherently requires more vigorous experimentation. As a consequence of this, Figure 5.15 indicates that a generalization by copying level of 0.2 produced the best results during Practical TPDP solution of the Race Track problem.

The 0.2 level of generalization by copying did not result in the lowest overall performance metric value however. The 0.6 level did. But the low performance metric value resulting from the 0.6 level was the spurious result of a near optimal policy being discovered exceptionally and fortuitously early in the Practical TPDP learning process. As Figure 5.15 indicates, at a generalization by copying level of 0.7 a near optimal policy was not found at all (no performance metric value was determined within 2500 epochs). The low performance metric value resulting from the 0.6 level is therefore not an indication that such a high level of generalization by copying is likely to produce superior results.

5.4 Practical TPDP and TP Allocation

5.4.1 TP Allocation in the Race Track Problem

Figure 5.16 shows the percentage of TPs allocated as Practical TPDP learned a near optimal policy for the Race Track problem. This percentage was based on the fact that roughly 15440 states could be entered during movement of the car around the track

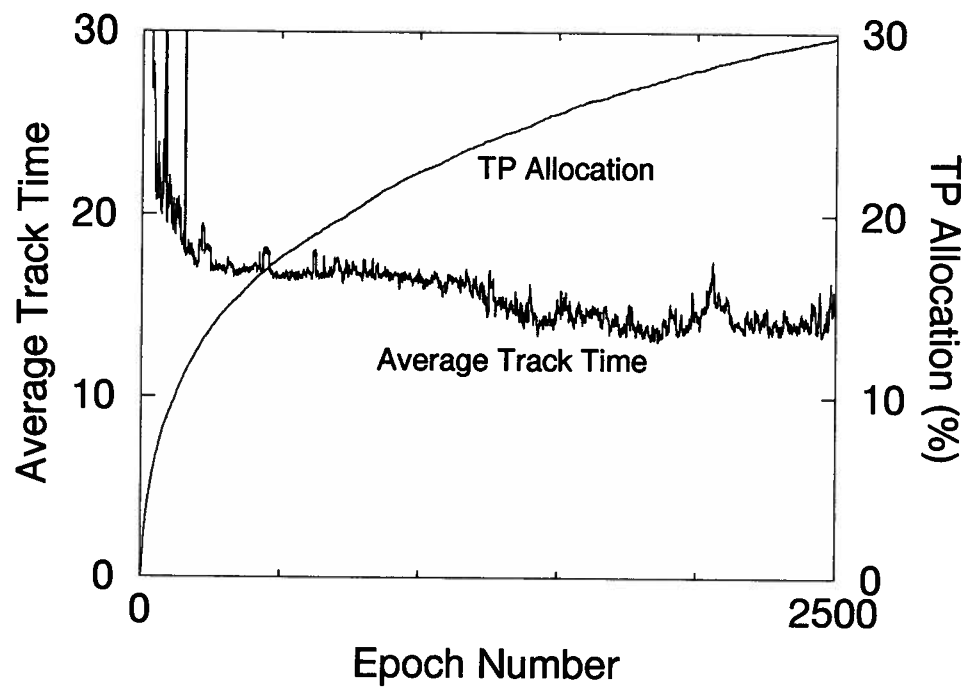


Figure 5.16: TP Allocation as Practical TPDP Learning Progressed

considered, and that 9 actions were possible in each state. As a result, it was possible for 138960 TPs to be allocated.

5.4.2 Superfluous TPs

Figure 5.16 shows that the percentage of TPs allocated during Practical TPDP solution of the Race Track problem continued to increase after epoch 1500, when the learning transition had certainly occurred (see Section 5.2.2) and a near optimal policy had basically been learned. In any application of Practical TPDP, TP allocation will continue (likely with a decreasing rate of allocation, see Figure 5.16) until every state in the system concerned has at least one TP associated with it. There are two reasons for this.

The first reason for continued TP allocation is that the R-value of any TP specifying an optimal action will always be the same or higher than the Q-value of that TP (theoretically, see Theorem 3.6). If it is higher, then that TP will rightfully never be removed (see Figure 4.4). If it is the same, then that TP is unnecessary (see Sections 3.2.5 and 3.3.9). But any small system disturbance (noise, or a new suboptimal TP for example) will make it higher. As a result, TPs specifying optimal actions at dormant states (see Section 3.1.1) are rarely removed, and as new TPs specifying optimal actions are continually added to the closed state space S_C (as a result of internal exploration, see Section 4.2.3), the number of TPs in S_C will continue to grow.

The second reason for the continued allocation of TPs is that one TP is allocated every time external exploration occurs (see Section 4.2.3). As Practical TPDP learning progresses, this will result in TPs being allocated at states more and more remote from the closed state space S_C.

Continued TP allocation is undesirable because it results in Practical TPDP using more memory than is necessary, and that additional memory usage does not typically facilitate any real improvement in system performance. In fact, continued TP allocation can negatively affect any near optimal policies that have been learned. This can be seen in Figure 5.16, where the average track time sporadically increased after a near optimal policy had been learned (around 1800 epochs). Such distortion of the policy results when new TPs, specifying suboptimal actions, are allocated at states between those where optimal actions are already being specified by TPs. Eventually these new TPs will be removed, or replaced by TPs specifying optimal actions, but until that happens they perturb the policy that has been learned, increasing the costs experienced by the system.

Continued TP allocation can also prevent the removal of TPs at dormant states (see Sections 3.2.5 and 3.3.9) specifying optimal actions. As described previously in this Section, the removal of such TPs is prevented when system disturbances increase their R-values. The allocation of new TPs specifying suboptimal actions creates such disturbances, so the continued allocation of new TPs can deter the removal of older, unnecessary TPs.

In general, once a near optimal policy has basically been learned, there is little point in Practical TPDP allocating any more TPs. Continued allocation will have the undesirable effects described, and while there is some possibility that each new TP can increase the optimality of the existing policy, this is not a significant concern. This is because the action specified by each new TP, even if it is optimal (and many will not be), can at best slightly improve the overall policy. If the system being controlled has any inertia, and TPs have already been allocated that result in a near optimal policy, then new TPs placed between the existing TPs can only specify short duration control adjustments that will have very limited effect (see Section 3.1.4). As the addition of such TPs is not really worthwhile, continued TP allocation should be prevented in some way.

Sections 5.4.3 through 5.4.5 present a number of ways in which continued TP allocation can be prevented in Practical TPDP.

5.4.3 Stopping Learning

The simplest way to prevent continued TP allocation in Practical TPDP is to arbitrarily stop the Practical TPDP learning process at some point and retain the policy which has been learned up to that point. Such an approach requires the ability to judge the optimality of the existing policy so that the stopping decision can be made. Such an approach also rules out adaptive control as it cannot be applied to a system which is changing. Both of these requirements can be met in many potential applications of Practical TPDP however.

A variation to stopping learning is to steadily decrease the level of internal and external exploration by varying the random exploration parameters appropriately (see Section 4.2.6). This approach requires some knowledge of an appropriate rate of exploration reduction.

5.4.4 Arbitrarily Limiting the Number of TPs

Another approach to preventing continued TP allocation is to arbitrarily limit the number of TPs which can be allocated. Considering Figure 5.16, it would seem reasonable to limit the percentage of TPs allocated during solution of the Race Track problem to the percentage at which the learning transition to a near optimal policy occurred (see Section 5.2.2). The TP allocation at that point was around 24%. When arbitrary TP limitation was attempted during Practical TPDP solution of the Race Track problem, allocation limits between 20% and 30% produced good results. At an allocation limit of 20% the learning transition occurred sooner, and the sporadic increases in the average track time after the learning transition occurred were reduced. The average track time curve at the 20% allocation limit was generally lower and flatter. This was due to the prevention of superfluous TPs, as described in Section 5.4.2. As the allocation limit was increased from 20% to 30%, where the allocation of new TPs was not limited much at all (see the percentage of allocation curve in Figure 5.16), these beneficial effects subsided.

Reducing the allocation limit below 20% resulted in increased glitching (see Section 5.3.3) and, eventually, severe disruptions in the learning of a near optimal policy. With an allocation limit of 10% the learning transition (see Section 5.2.2) did not occur within 2500 epochs. Figure 5.17 shows the results of a 15% TP allocation limit. This 15% allocation limit resulted in the learning transition being made at roughly the same point at which it was made when no allocation limit was applied (a point later than that which resulted from a 20% allocation limit). The average track time curve was lower and flatter than that which resulted when there was no allocation limit however.

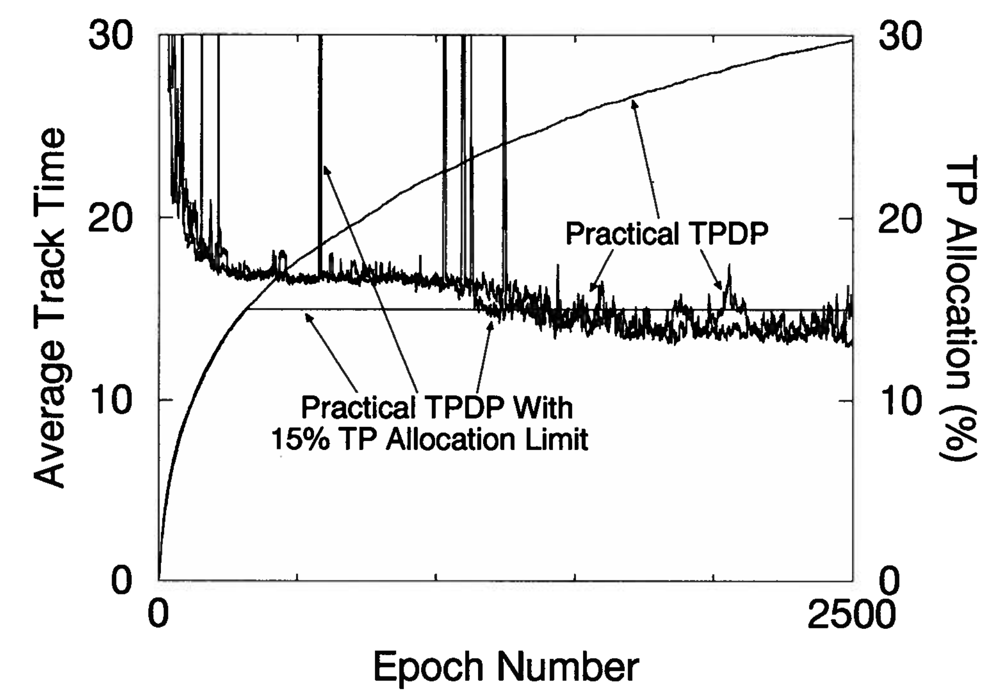


Figure 5.17: The Effect of Limiting TP Allocation

5.4.5 Placing a Price on TPs

The rate of TP allocation can be reduced by placing a cost on the allocation and preservation of each TP. This is done by adding an *allocation cost* $c_{\text{allocation}}$ to the following values upon which decisions are based in the Stack Updating Procedure (see Appendix Section B.3, Figure 4.4):

1. Line 24: $(Q(i_s, u_s) < R(i_s, u_s))$ becomes $(Q(i_s, u_s) + c_{\text{allocation}} < R(i_s, u_s))$
2. Line 40: $(C_{\text{total}} < Q(i_s, \mu(i_s)))$ becomes $(C_{\text{total}} + c_{\text{allocation}} < Q(i_s, \mu(i_s)))$
3. Line 50: $(V_{\text{update}} < V_{\text{expected}})$ becomes

$$V_{\text{update}} + c_{\text{allocation}} < V_{\text{expected}})$$

Change 1 results in TPs being removed if their Q-value is not at least $c_{\text{allocation}}$ lower than their R-value. Change 1 thereby increases the likelihood that TPs specifying optimal actions at dormant states will be removed (see Section 5.4.2). Changes 2 and 3 prevent the allocation of new TPs as a result of internal exploration unless the costs experienced during the exploratory state space transitions were $c_{\mathbf{allocation}}$ lower than the costs which would normally have been experienced.

Figure 5.18 shows the result of employing an allocation cost $c_{\mathbf{allocation}}$ of 0.1 during Practical TPDP solution of the Race Track problem. Doing so had the effect of making the learning transition (see Section 5.2.2) occur sooner than when there was no allocation cost, as well as reducing the sporadic increases in average track time after the learning transition had occurred. The average track time curve when $c_{\mathbf{allocation}}$ was set to 0.1 was generally lower and flatter, although it did include many more glitches (see Section 5.3.3). The increased number of glitches was due to the allocation cost increasing the likelihood that TPs at zero velocity wall states would be removed (see Section 5.3.3). All of these results were due to the prevention of superfluous TPs, as described in Section 5.4.2. Figure 5.18 indicates that the expected reduction in TP allocation did in fact result from the use of the allocation cost. Overall, employing an allocation cost of 0.1 during Practical TPDP solution of the Race Track problem proved to be very beneficial.

Allocation costs above 0.1 resulted in the learning transition (see Section 5.2.2) not being made before 2500 epochs had passed. This occurred because the allocation cost deterred TP allocations that were necessary for the learning of a near optimal policy. Extensive glitching also occurred as increasing allocation costs increased the likelihood that TPs at zero velocity wall states would be removed (see Section 5.3.3).

The allocation cost approach to reducing TP allocation results in suboptimal polices being learned because it effectively makes reduction of the number of TPs part of the optimality criteria. This prevents Practical TPDP from learning optimal policies that are full in accordance with the optimality criteria. This effect, making TP allocation part of the optimality criteria, may be exactly what is desired in some applications of

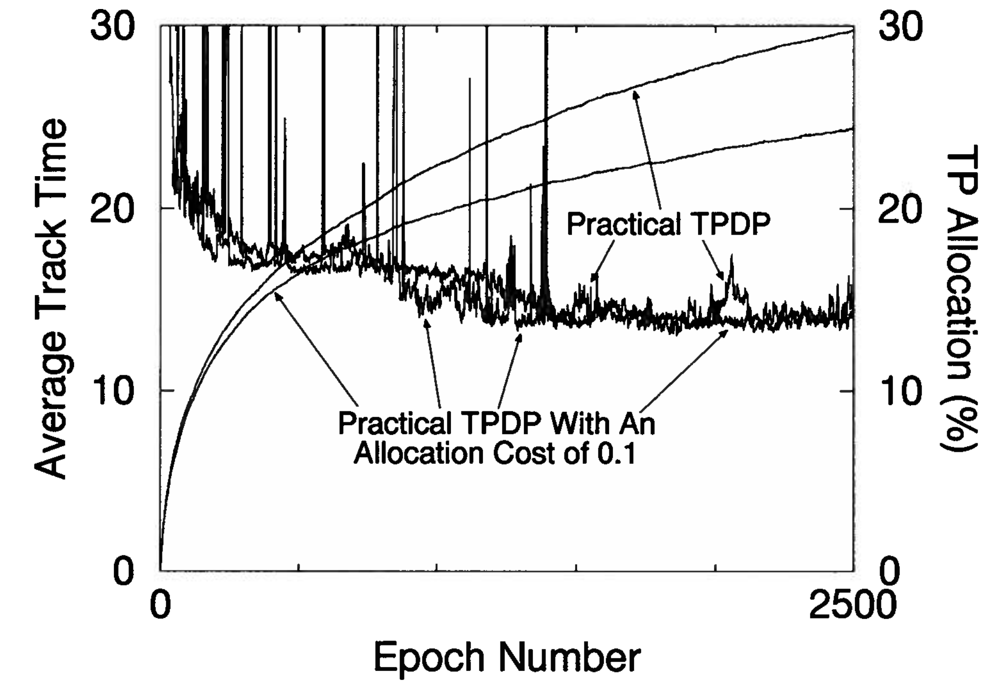


Figure 5.18: The Effect of Incorporating a TP Allocation Cost

Practical TPDP however.

5.4.6 Eliminating Suboptimal TPs

As explained in Section 4.1.2, Practical TPDP can result in more than one TP being simultaneously associated with a state. As learning progresses in Practical TPDP, and a near optimal policy is learned, a policy TP will be selected for each state that specifies an optimal action (see Section 4.2.2). At this point the actions specified by other TPs associated with the same states no longer need to be considered. As a result, the TPs specifying suboptimal actions can be eliminated. Eliminating such suboptimal TPs will reduce the amount of memory used by Practical TPDP, and have few other effects.

There are innumerable ways in which suboptimal TPs can be identified and eliminated. One approach that was attempted during Practical TPDP solution of the Race

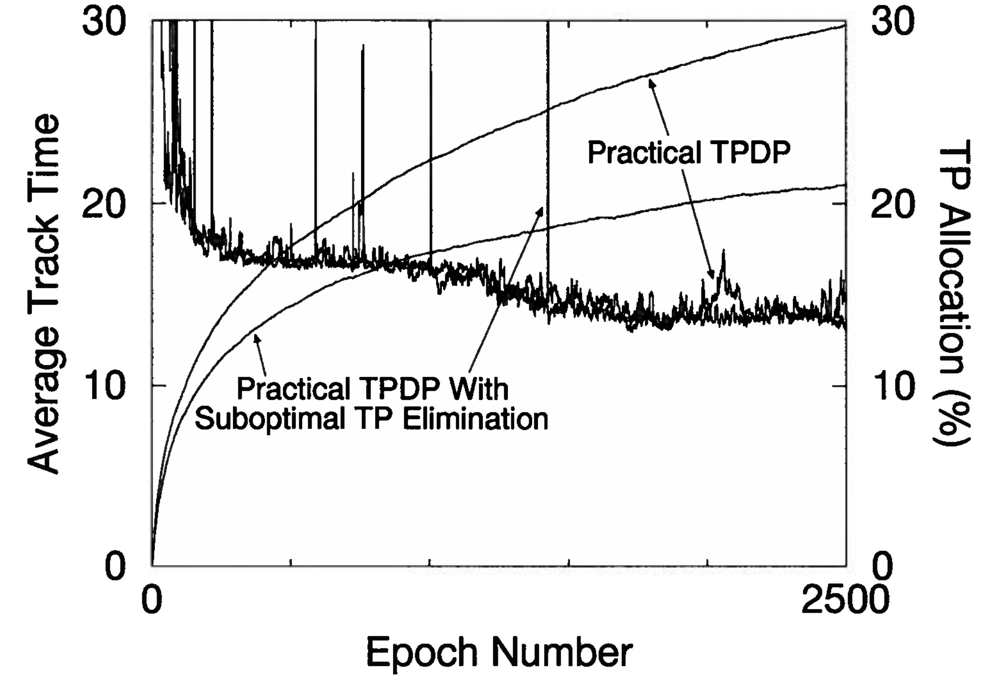


Figure 5.19: The Effect of Eliminating Suboptimal TPs

Track problem was to randomly eliminate TPs whose Q-value was greater than the Q-value of the policy TP. Such elimination was done randomly in order to ensure that Practical TPDP was provided with some time (stochastically determined) in which to fully learn the Q-values of each TP. Figure 5.19 shows the results of eliminating suboptimal TPs in this manner, with each suboptimal TP having a 0.25 probability of being eliminated each time its Q-value was updated. Figure 5.19 indicates that eliminating the suboptimal TPs, aside from reducing the sporadic increases in the average track time after the learning transition had occurred (see Section 5.2.2), had almost no effect on the learning of a near optimal policy – as compared to when elimination was not performed. The allocation of TPs was significantly reduced however.

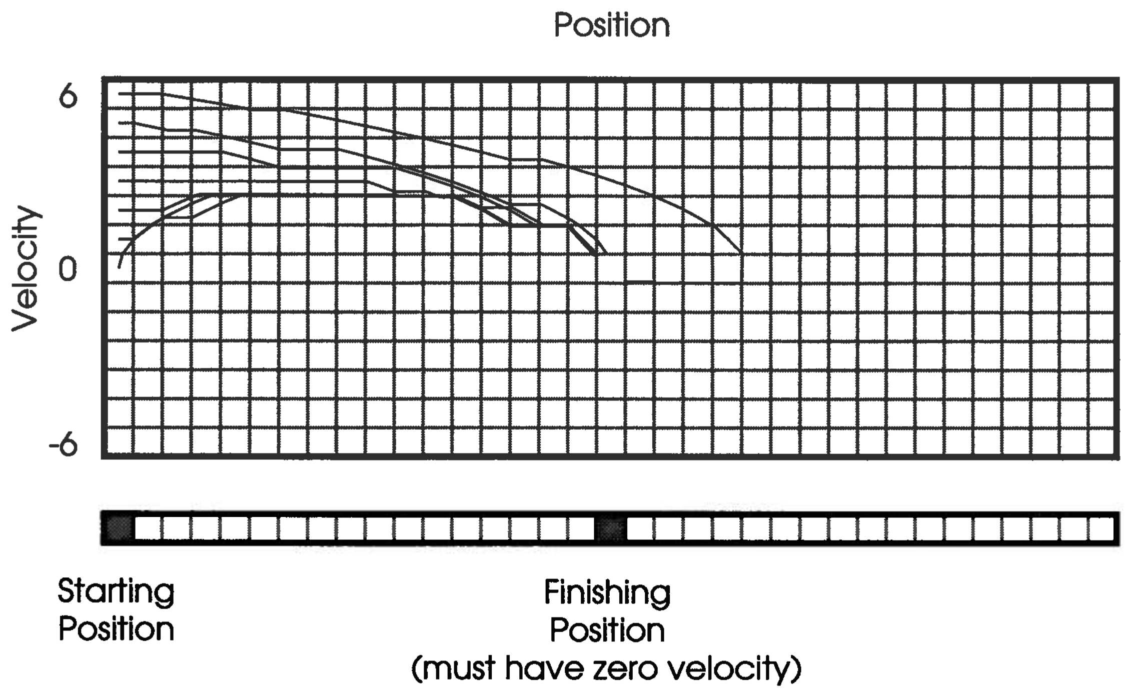


Figure 5.20: The One-Dimensional Race Track and Its Phase Plane

5.4.7 TP Allocation in a One-Dimensional Race Track Problem

In order to observe the actual allocation of TPs in Practical TPDP, it was applied to a one-dimensional Race Track problem. This track is shown at the bottom of Figure 5.20. For this problem the car was started at the starting position with a randomly chosen integer velocity from 0 to 6 (motion towards the right had a positive velocity). It then had to move horizontally until it arrived at the finishing position with zero velocity. Aside from the different track, all of the Race Track problem and Practical TPDP parameters were kept the same as those used on the more complex track described in Sections 5.1.1 to 5.1.6. As a result, the car could enter 420 states, and with 3 actions possible in each state, it was possible for 1260 TPs to be allocated.

Above the track shown in Figure 5.20 is a *phase plane* that indicates the horizontal position and velocity of the car as it moves about the track. The state of the car on the one-dimensional track is fully described with these two values, thus facilitating the two-dimensional representation of the system state space in the form of the phase plane.

Figure 5.20 also shows the phase plane traces resulting from typical runs of the car down the track after 200 epochs of Practical TPDP learning of the optimal policy. The trace produced when the initial velocity was set to 6 (the highest initial velocity possible, and the one with the top trace in Figure 5.20) indicates that the car moved past the finishing position and then returned to it. This was due to the fact that, with that initial velocity, the car simply could not decelerate in time to avoid passing the finishing position. The optimal policy in that case was to return to the finishing position as quickly as possible.

Figure 5.21 shows the phase plane traces resulting from typical runs of the car after Practical TPDP with a 25% TP allocation limit (see Section 5.4.4) had been applied to the one-dimensional Race Track problem for 200 epochs of learning. Comparing this figure to Figure 5.20, it is clear that the 25% TP allocation limit inhibited the learning of a near optimal policy. Figure 5.22 shows the average track time and TP allocation results when Practical TPDP was applied to the problem both with and without a 25% TP allocation limit. This figure indicates that the average track time sporadically increased once a near optimal policy had been learned with the 25% TP allocation limit. TP allocation limits higher than 25% resulted in performance comparable to that which resulted when there was no allocation limit.

Both the TP allocation percentage curves (Figures 5.16 and 5.22) and the lowest feasible TP allocation limit were higher when Practical TPDP was applied to the one-dimensional track than they were when it was applied to the more complex track (see Sections 5.4.1 and 5.4.4). This was mainly due to the fact that on the one-dimensional track, with only three possible actions in each state, the 25% TP allocation limit resulted in an average of 0.75 TPs being allocated per state. In contrast, the more complex track, with nine possible actions in each state, had an average of 2.25 TPs per state (at the

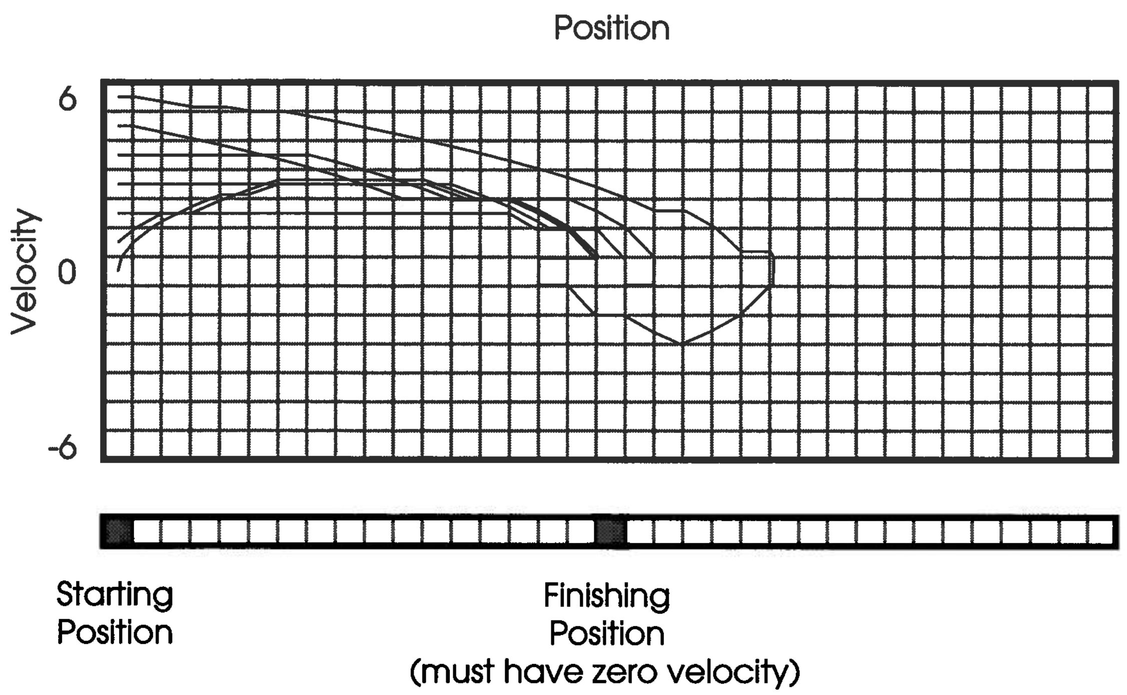


Figure 5.21: Limiting TP Allocation on the One-Dimensional Race Track

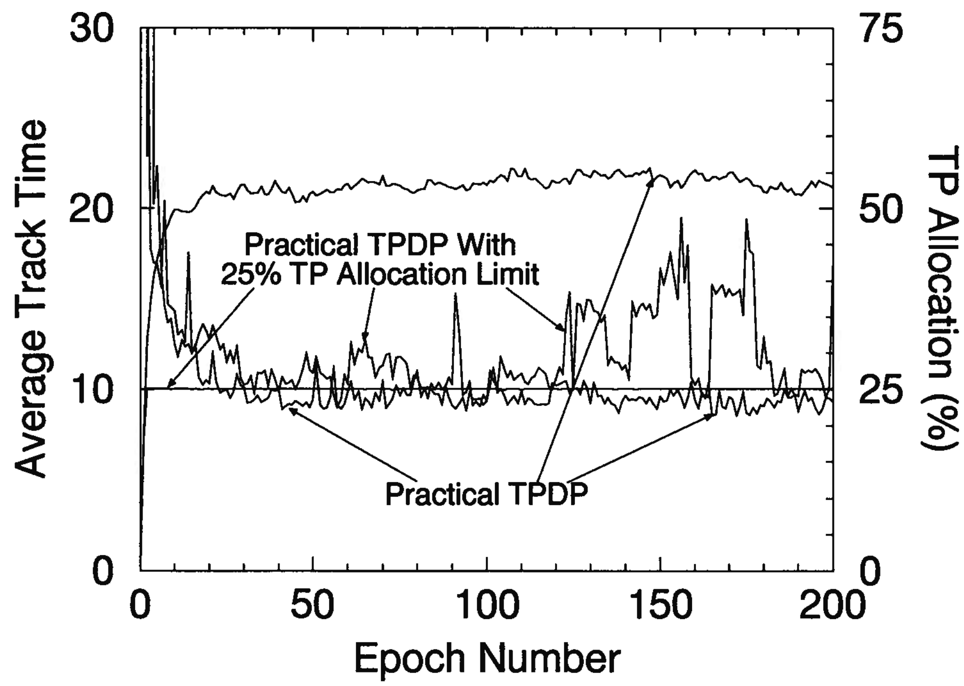


Figure 5.22: Performance of Practical TPDP on the One-Dimensional Race Track

25% TP allocation limit). The difference in the feasible TP limit was also due to the fact that there were fewer external states that did not require TPs (see Section 3.2.4), relative to the total number of states, in the one-dimensional Race Track problem.

Finally, Figures 5.23 and 5.24 show the location of the policy TPs in the phase plane after 200 epochs of Practical TPDP learning both with and without a 25% TP allocation limit. The tails extending from each TP in these figures indicate the phase plane trace that resulted when the system was started at the center of each TP state, the action specified by the policy TP was applied to it, and state transitions were allowed to continue (without the stochastic interference of noise) until another TP state was encountered. Both figures show that the policy TPs generally specified actions that directed the car to the finishing position. Both figures also show that many policy TPs in the upper-right and lower-left corners of the phase planes did not specify such actions. In some cases the policy TPs in these regions actually specified actions that accelerated the car away from the finishing position. This was because, when the car was in those states, it could not decelerate in time to prevent collision with the ends of the track. As a result, the optimal policy was to accelerate, causing the collision to occur sooner, so that movement towards the finishing position could begin as soon as possible.

Figure 5.23 indicates that a TP was allocated for almost every state when there was no TP allocation limit. Figure 5.24 shows the more interesting TP positioning that resulted when TP allocation was limited.

5.5 Varying the Practical TPDP Algorithm Parameters

Figures 5.25 to 5.30 indicate the results, in terms of performance metric values (see Section 5.3.4), of varying the Practical TPDP Algorithm parameters during different solutions of the complex Race Track problem (see Sections 5.1.1 to 5.1.6). While each

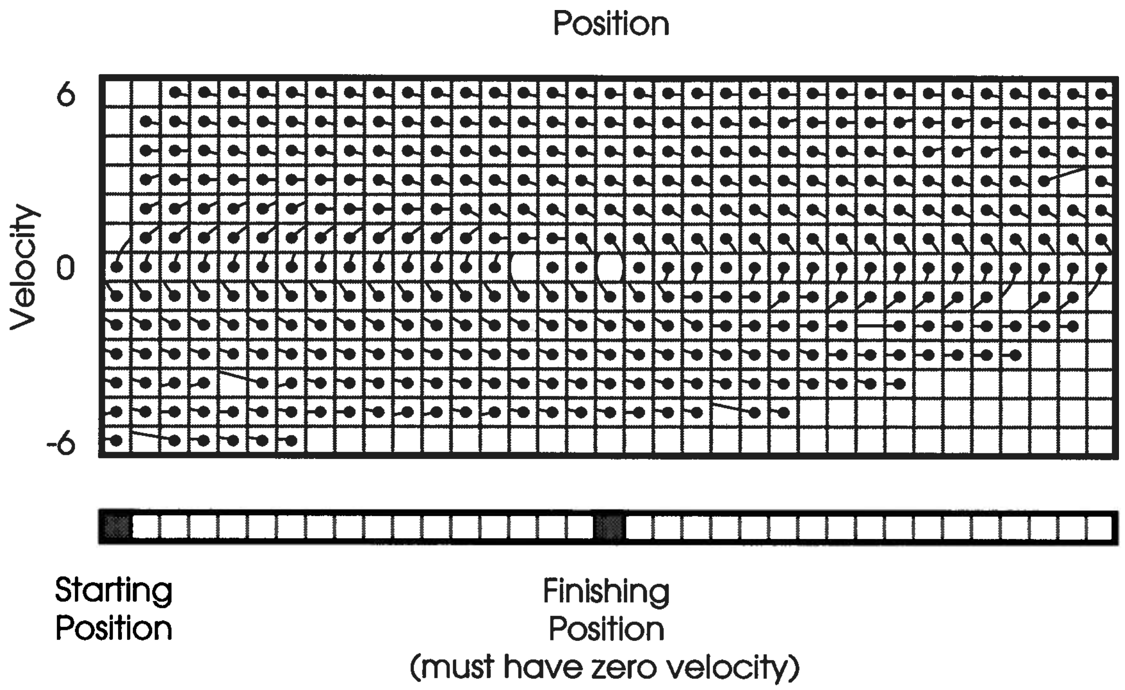


Figure 5.23: TP Positioning in the One-Dimensional Race Track

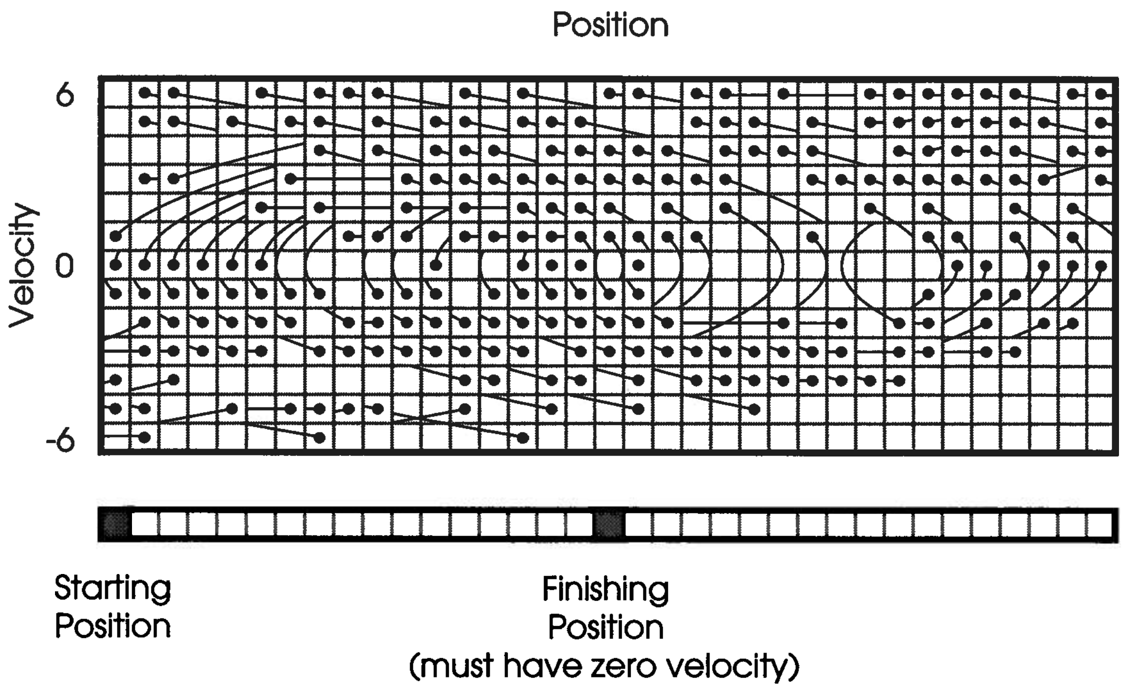


Figure 5.24: TP Positioning with a 25% TP Allocation Limit

parameter was varied the others were maintained at the settings specified in Section 5.1.5. The parameters w_{initial}, w_{thr} and w_{max} were not analyzed in this way because these parameters, as long as they were set to reasonable values (see Sections 4.2.1, 4.2.2 and 4.2.7), had little effect on the learning performance of Practical TPDP.

Missing sections of the curves shown in Figures 5.25 to 5.30 indicate parameter settings for which a performance metric value (see Section 5.3.4) was not determined within 2500 epochs. This means that the average track time was not determined to be less than 14 more than 100 times within 2500 epochs, and that the learning transition (see Section 5.2.2) had thus not occurred.

As explained in Section 4.2.6, for each application of Practical TPDP $\sigma_{\mathrm{add\text{-}TP}}$ must be adjusted to match the time required to pass through the high velocity states in the system under consideration, and σ_{change} must be adjusted so that the action changes specified during internal and external exploration are made at intervals that reflect the ability of the system concerned to respond. As a result, both of these parameters must be tuned to suit the problems to which Practical TPDP is applied. As can be seen in Figures 5.27 and 5.29 there was a fairly wide range of settings over which both parameters produced good performance on the complex Race Track problem.

The setting of the parameter σ_{delay} also depends on the dynamic characteristics of the system to which Practical TPDP is being applied (see Section 4.2.6). Figure 5.25 indicates however that the performance of Practical TPDP was fairly independent of the setting of this parameter during solution of the complex Race Track problem.

Of the various parameters, Practical TPDP was found to be most sensitive to the setting of $\sigma_{\mathrm{swap\text{-}TP}}$ (see Figure 5.26). Normally, the main concern with such sensitivity would be that a suitable parameter setting will be difficult to determine when Practical TPDP is applied to different problems. This is not a concern with $\sigma_{\mathrm{swap\text{-}TP}}$ however, because $\sigma_{\mathrm{swap\text{-}TP}}$ does not have to be adjusted to match the system to which Practical

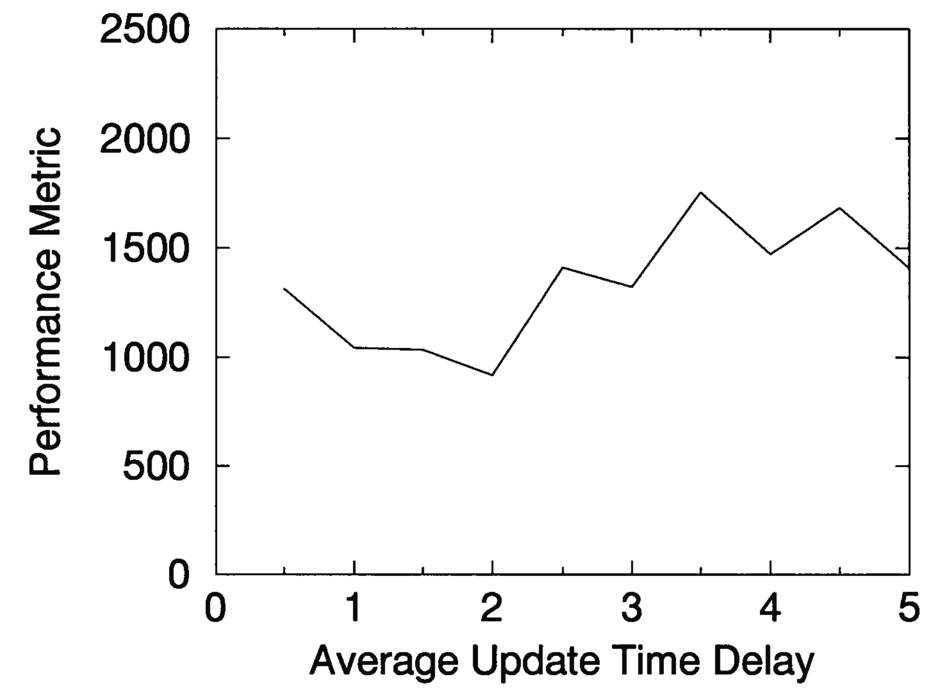


Figure 5.25: The Effect of Changing $\bar{\sigma}_{\text{delay}}$

TPDP is being applied. It simply sets the balance between the evaluation of TPs already associated with each state, and the internal exploration of other TP possibilities for those states (see Section 4.2.6 and Appendix Section B.2). Setting $\Pr(\sigma_{\text{swap-TP}} > 0)$ to 0.2 produced good results during Practical TPDP solution of both Race Track problems, and the 0.2 setting should result in reasonable performance in most applications.

Figure 5.28 indicates, uninterestingly, that the setting of $\sigma_{\text{add-TP}}$ does not have to be precise. Figure 5.30 indicates, unsurprisingly, that the learning rate α should be set high enough to facilitate some learning, but not so high that the results from each learning experience can drastically alter what has already been learned.

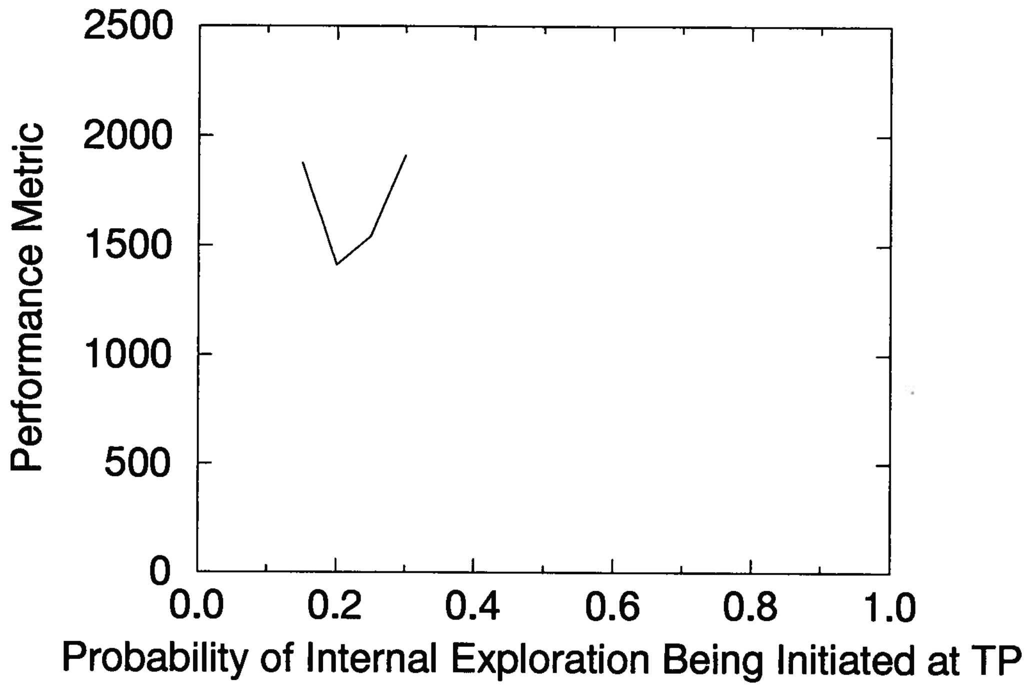


Figure 5.26: The Effect of Changing $\Pr(\sigma_{\text{swap-TP}} > 0)$

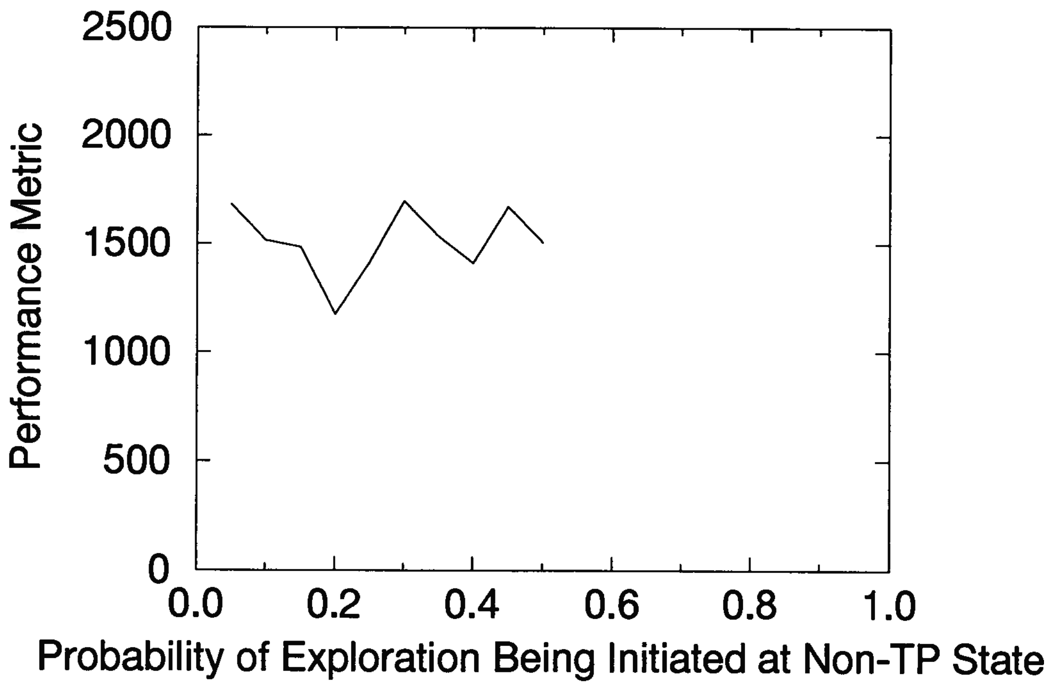


Figure 5.27: The Effect of Changing $\Pr(\sigma_{\text{add-TP}} > 0)$

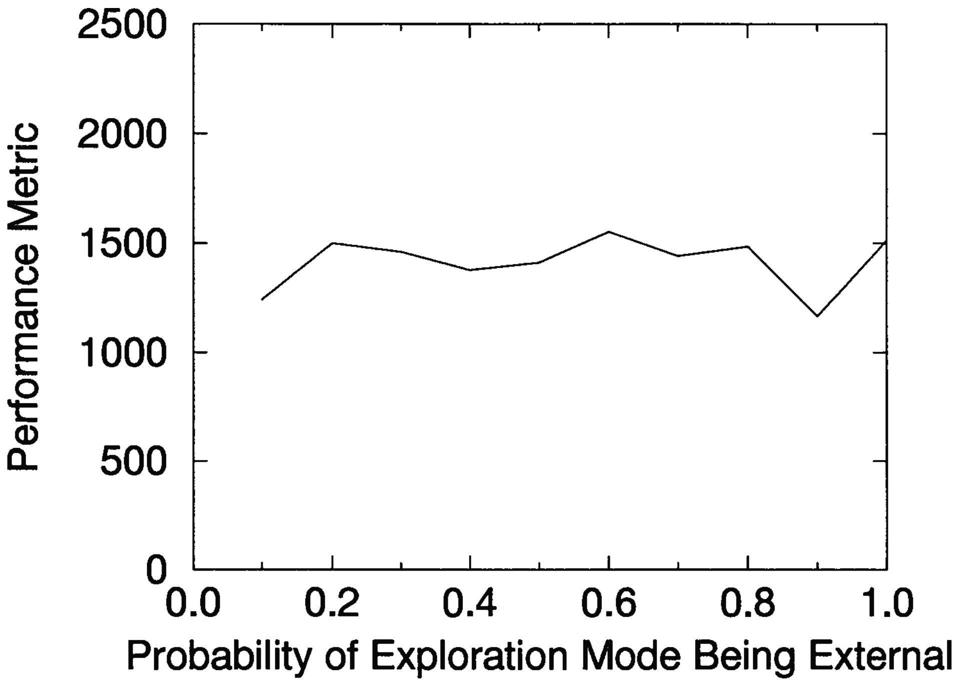


Figure 5.28: The Effect of Changing $\Pr(\sigma_{\text{external}} > 0)$

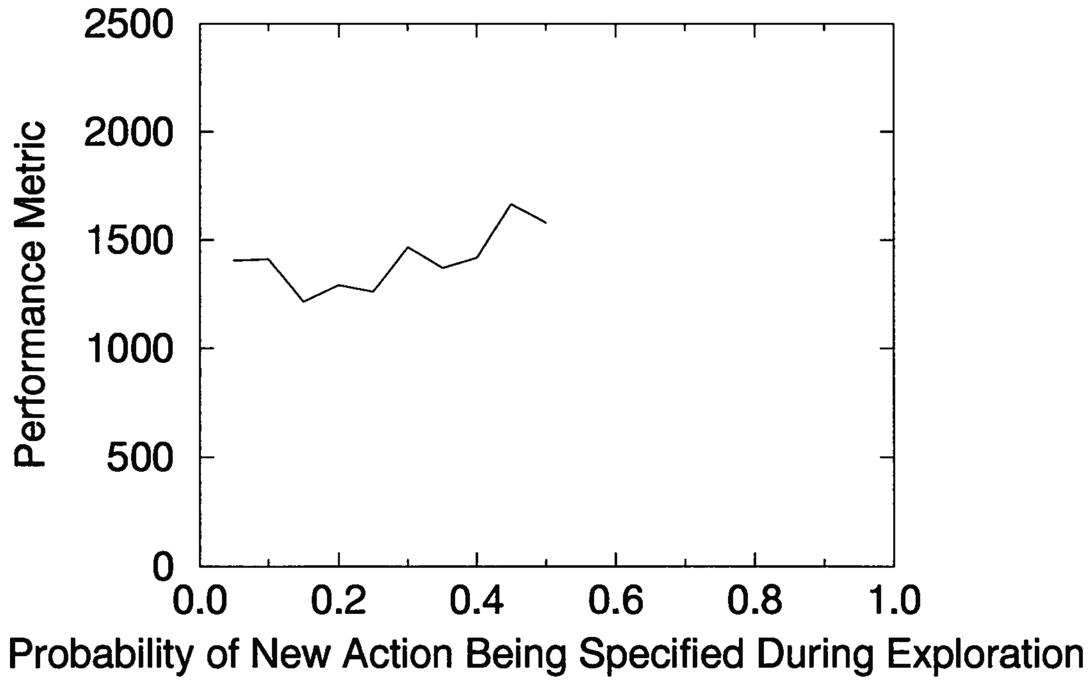


Figure 5.29: The Effect of Changing $\Pr(\sigma_{\text{change}} > 0)$

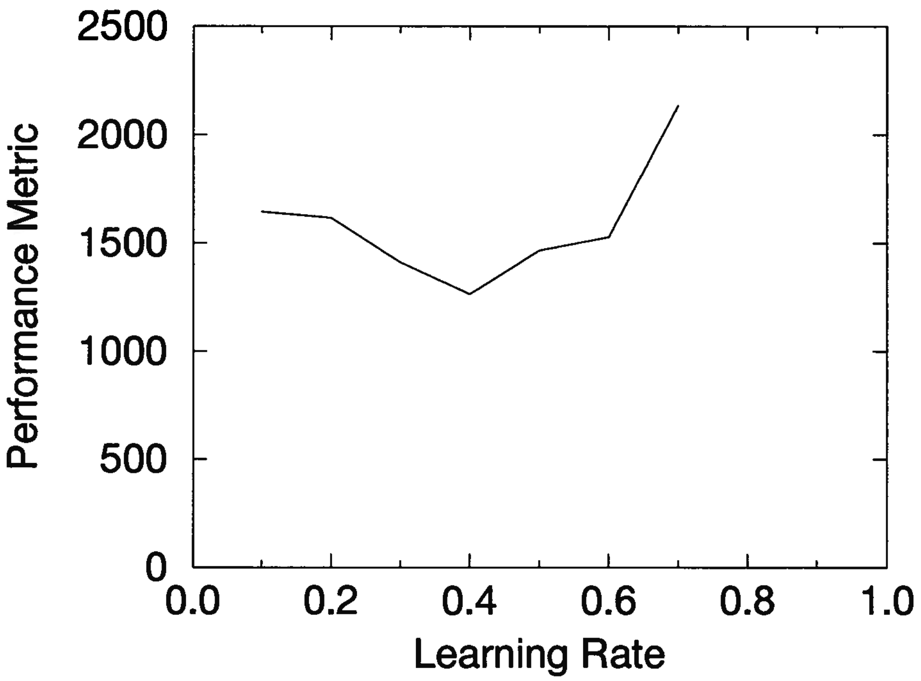


Figure 5.30: The Effect of Changing α

Chapter 6

Neural TPDP

TPDP was developed in the context of neural network control. This Chapter will describe how Practical TPDP can be implemented with a neural network. Some characteristics of this neural implementation, as well as its biological plausibility, will be discussed.

6.1 A Neural Network Design for Direct DP

6.1.1 Neural Networks and Evaluation Function Approximation

As explained in Section 2.4.4, neural networks have been used extensively to parametrically approximate the evaluation function of DP controllers. Such approximation facilitates generalization of the evaluation function and combats the "curse of dimensionality" (see Section 2.4.1) by eliminating the need for the evaluation function to be represented by an individual value $V_\mu(i)$ at each state i (see Section 2.4.4).

As further explained in Section 2.4.4, the main drawback to evaluation function approximation is that the approximating mechanisms themselves typically require extensive computational effort to develop a reasonably accurate approximation. If parametric approximation is being done with neural networks, the computational effort of approximation is that required for network training (for example Rumelhart *et al.*, 1986).

Evaluation function approximation is more difficult in Q-learning controllers, including TPDP controllers, because Q-values are a function of action as well as state, and an additional action dimension must be included in any approximation that represents

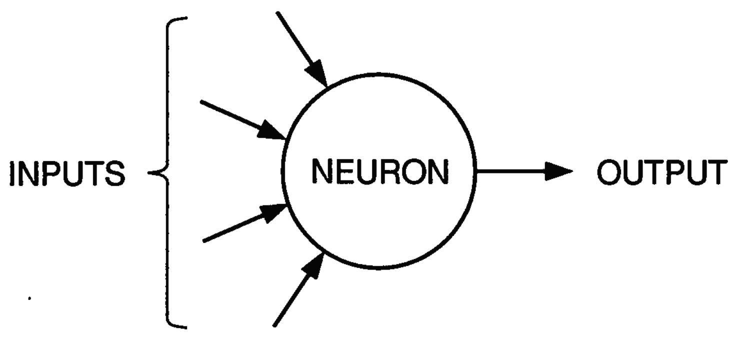


Figure 6.31: A Generic Neuron Model

Q-values (see Section 2.4.4). In the case of TPDP controllers, evaluation function approximation is inherently more difficult because TPDP focuses on the effects of actions instead of on the determination of an evaluation function (see Section 3.4.3).

As a result of all this, evaluation function approximation was not attempted during this research, neither with neural networks or with any other approach. A neural network design that could directly facilitate Practical TPDP control was developed however, and it will be discussed in Sections 6.1.2 to 6.1.8. In fact, as mentioned in Section 3.1.1, this neural network design actually led to the development of TPDP. TPDP evolved out of attempts to modify neural connections so that groups of neighboring states with similar control requirements could be amalgamated and represented with one neuron (Buckland *et al.*, 1993). The general neural network design used for the attempts at such amalgamation is what will be presented.

6.1.2 A Neural Network Design for State Space Control

A generic neuron model is presented in Figure 6.31. In general terms, a neuron is a device which accepts a number of inputs and passes these inputs through some function to produce an output. A generic neuron model of this type can be used as the constituent element of a controller as shown in Figure 6.32.

The *state lines* shown in Figure 6.32 carry binary signals, and each represents a

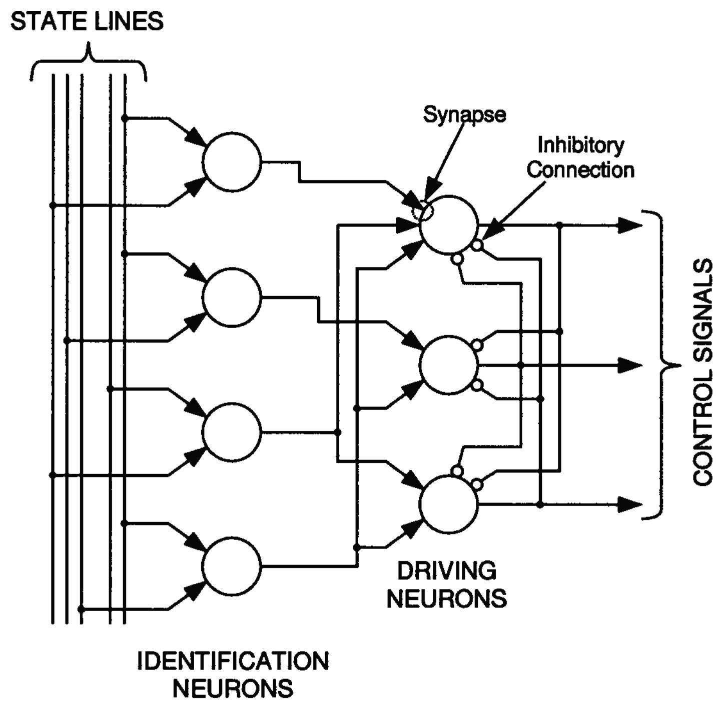


Figure 6.32: A Neural Network Design for State Space Control

quantized interval of one dimension of the state space. As a result, each state space dimension is represented by a group of such state lines, one for each interval of the dimension. One state line in such a group will always be *high* (active) while all of the others are *low* (inactive), indicating which interval of the dimension the system is in. A single high state line in every group of state lines fully defines the state of the system.

As indicated in Figure 6.32, *identification neurons* inspect the state lines. Each of them is responsible for identifying a particular state i. They output a high signal when the system enters the state they are responsible for. The function of the identification neurons is to act as *and-gates*. They *and* together a single state line from each group (each state space dimension) to identify when the system is in the state i uniquely identified by that set of state lines. Only one identification neuron outputs a high signal at any given time because each one identifies a separate state, and the system can only be in one state at a time.

The *driving neurons* specify the actions that control the system. When appropriate, each one signals with a high binary signal that a specific action should be performed. The action specified by each driving neuron u is always the same, and it is different from those specified by the other neurons. To prevent more than one driving neuron from specifying an action at the same time, they are connected to mutually inhibit each other (see Figure 6.32, Feldman *et al.*, 1982). The inhibiting signals are randomly perturbed slightly to resolve deadlocks between driving neurons that are equally active[1].

The outputs of the identification neurons are connected as inputs to the driving neurons, and the point of connection is called a *synapse*[2]. Whenever one of the inputs to a driving neuron is high, that neuron is active and is capable of outputting a high signal to specify an action. If it is the victor of a mutual inhibition contest with any other driving neurons that may be active, it is able to do so. The driving neurons thus act as (mutually inhibiting) *or-gates*, allowing a number of different identification neurons (only one of which outputs a high signal at any given time) to specify the same action.

The connections between the identification neurons and the driving neurons – the synapses – define how the system is controlled. When a system state is identified, a high signal is passed from the corresponding identification neuron, through one or more synapses, to a number of driving neurons. The driving neurons that are activated as a result engage in a mutual inhibition contest to determine which action will actually be specified. If each identification neuron is connected to only one driving neuron, the resulting network operates in exactly the manner of the control paradigm described in Section 2.1.1.

[1]All active driving neurons are equally active in the neural network design described up to this point. This will not remain true as the design is more fully developed however.

[2]The input connection points on the identification neurons are *not*, by this definition, *synapses*. The reason for this is presented in Section 6.1.5.

Overall the neural network performs an *and-or* function. Similar network designs have been investigated by others – Minsky (1985) and Ayestaran *et al.* (1993) are two examples. Further, the mutual inhibition between the driving neurons results in a form of competitive learning (Rumelhart *et al.*, 1985).

6.1.3 ACAM Operation of the Neural Network Design

A controller implemented with a neural network in the manner being described is inherently a memory-based controller (see Section 2.3.2). Specifically, it operates as an ACAM (see Section 2.4.2). The identification neurons act as self-activating table entries, specifying through the driving neurons[3] the action that is appropriate for the state to which they attend. Because they are self-activating, the number of identification neurons allocated need only be the number required to specify an action in each state that the system actually enters. As a result, the total amount of memory required for control can be reduced below what would be necessary if a full table were employed. This is one of the main rationales for implementing a controller with this type of neural network.

6.1.4 Implementing Mutual Inhibition

There are a number of ways in which signal level determination can be made in a mutually inhibiting neural network controller. This Section will describe a simple and effective heuristic approach (developed during this research, although similar to work done by others – for example Feldman *et al.*, 1982) that facilitates parallel computation, in each of the driving neurons, of the mutual inhibition contest victor. It is based on the following update equation for the *driving neuron output* $m_t(u)$ of each driving neuron u at time

[3]The number of driving neurons will always be fixed – there must be one for each possible action.

step t:

$$m_{t+1}(u) = \min(1, \max(\kappa a_t(u), m_t(u) + \psi \left[\frac{1}{2} + m_t(u) - \sum_{v \in U} m_t(v)\right])) \tag{6.47}$$

Where $a_t(u)$ is the *activity level* of driving neuron u at time step t, κ is the *low-grade output factor*, ψ is the updating rate, and U is the set of actions specified by all of the driving neurons. Normally κ should be set to around 0.1 to keep the low-grade neuron output low and unintrusive.

The activity level $a_t(u)$ is the result of *or-ing* together the inputs to each driving neuron u (see Section 6.1.2). It is the signal level that each driving neuron would output if it was not being mutually inhibited by the other driving neurons. The activity level $a_t(u)$ is multiplied by the low-grade output factor κ to determine a weak *low-grade output level* (see Equation 6.47) that each driving neuron should output to indicate that it is active (see Section 6.1.2).

When all of the driving neuron outputs $m_t(u)$ are at a low level, updating Equation 6.47 will increase all of them. The output level of each will continue increasing until, for each neuron, $\left[\frac{1}{2} + m_t(u) - \sum_{v \in U} m_t(v)\right] < 0$. When each driving neuron reaches this *balancing point* its output $m_t(u)$ will begin decreasing. The output level of each driving neuron at the beginning of the mutual inhibition contest determines how soon each neuron reaches its balancing point. The lower the initial output level of each neuron, the sooner it will reach its balancing point.

As the mutual inhibition contest continues, all but one of the driving neurons will reach a balancing point. This neuron will be the mutual inhibition contest victor, and its output level will continue increasing until it reaches 1. Equation 6.47 ensures that this high level in the victorious driving neuron results in all of the other driving neuron outputs being reduced to their low-grade output level $\kappa a_t(u)$.

For Equation 6.47 to produce these results, it must be ensured that the combined low-grade output levels of all the inhibited driving neurons are not enough to affect the neuron output $m_t(u_{\text{victor}})$ of the mutual inhibition contest victor. This is ensured when (from Equation 6.47):

$$\begin{aligned} \frac{1}{2} + m_t(u_{\text{victor}}) - \left[\sum_{v \in U} m_t(v)|_{\text{worst-case}} \right] &\geq 0 \qquad (6.48) \\ \frac{1}{2} + 1 - [(|U| - 1)\kappa + 1] &\geq 0 \qquad (6.49) \\ \frac{1}{2(|U| - 1)} &\geq \kappa \qquad (6.50) \end{aligned}$$

Where $|U|$ is the number of driving neurons.

The driving neuron which emerges victorious from the mutual inhibition contest will always be the one that had the highest output level $m_t(u)$ at the start of the contest. As the starting level equals the low-grade output level $\kappa a_t(u)$, and κ is a constant, the victor of the mutual inhibition contest will always be the driving neuron u with the highest activity level $a_t(u)$. Since all active driving neurons will have the same nominal activity level of 1, $a_t(u)$ is actually determined by multiplying the output from the driving neuron *or-ing* function by a uniformly distributed random value ranging from 0 to 1. Randomly perturbing the activity levels in this manner ensures that the mutual inhibition contest victor will be chosen randomly from amongst the active driving neurons.

6.1.5 Synapses as TPs

An optimal policy (see Section 2.1.2) is manifested in the neural network controller being described (see Figure 6.32) by making the right connections between the identification neurons and the driving neurons. That is, the right set of synapses must be determined for the network. Practical TPDP can be used to determine what this set should be.

To facilitate Practical TPDP learning of optimal policies in the controller being described, each synapse is treated like a TP. Q-values $Q(i, u)$, R-values $R(i, u)$ and weights

$w(i,u)$ are associated with each synapse and are used together to determine the merit of the single state/action combination that the synapse defines. For a given synapse/TP, the state i associated with that TP is the state attended to by the identification neuron outputting to the synapse, and the action u that the TP specifies is the action specified by the driving neuron to which that synapse is attached. Because synapses are essentially TPs in this context, the term "synapse" will henceforth refer to TPs[4].

The Q-value and R-value of each synapse reflect the costs experienced each time a high signal passes through that synapse (Section 6.1.6 will describe this more fully). The weight value $w(i,u)$ of each synapse is modified based on the Q-value and R-value of that synapse (see Section 4.2.1), and indicates the integrity of the synapse. A low weight indicates a new, untested synapse, or a synapse that has been found to lack merit. A high weight indicates a synapse that has been found to be worthwhile. When the weight of a synapse is decremented to zero that synapse is removed. The synapse weights are not used to modulate the signals going through the synapses in any way.

6.1.6 The Full Implementation of Neural TPDP

Section 6.1.5 presented the idea that Practical TPDP can be used to learn optimal policies in the neural network controller being described if synapses are treated like TPs. This neural implementation of Practical TPDP is called *Neural TPDP.* Neural networks can be used to implement Practical TPDP without the basic functionality of the Practical TPDP Algorithm being affected in any way (see Sections 4.2.4, Appendix Sections B.2 and B.3, and Figures 4.3 and 4.4). That is, a Neural TPDP controller will function exactly like a Practical TPDP controller, with the same Q-value, R-value and weight updates being made in what is effectively just a different processing environment. This

[4]This is why the term "synapse" is not applied to identification neuron input connections – "synapse" has a special meaning related to TPs that does not apply to such connections.

Section will explain in detail how such is done.

Basically, all states identified by identification neurons are TP states (see Section 3.2.1). The identification neurons that identify such TP states always output to at least one synapse. To ensure that action specifications are maintained as the system moves through non-TP states between TP states, which is a fundamental requirement of TPDP (see Section 3.1.1), the driving neurons must have some positive feedback. Such positive feedback ensures that, once a driving neuron has won a mutual inhibition contest (see Section 6.1.4), it will continue to specify the same action until another mutual inhibition contest occurs. If Equation 6.47 is used to determine the mutual inhibition contest winner, the required positive feedback occurs.

Movement of the system through the state space occurs as a result of successive mutual inhibition contests between the driving neurons (see Section 6.1.2). Each such contest is initiated when the current mutual inhibition contest victor ***flash reduces*** its output from 1 to its low-grade output level $\kappa a_t(u)$ (see Equation 6.47). Mutual inhibition contests are initiated when the system is in a TP state, but not at every TP state. As explained in Section 4.2.5, the random parameter $\sigma_{\mathbf{delay}}$ is used to maintain the same action specification while the system passes through a number of TP states. This is done so that the costs resulting from each action specification can be observed over an interval of reasonable length. Delayed updating is facilitated in Neural TPDP by flash reducing the output of the mutual inhibition contest victor (see Section 6.1.6) only when the system is in a TP state and $t > t_{\mathbf{update}}$ (see Section 4.2.5). The former condition is indicated to the mutual inhibition contest victor by the presense of at least one non-zero low-grade signal on its mutual inhibition inputs (see Section 6.1.4).

The Q-value of each synapse is updated (using Equation 3.18) when that synapse becomes active and the driving neuron that it is connected to is the mutual inhibition contest victor. When the Q-value of a synapse is updated, the weight of that synapse is updated as well (according to the rules of Section 4.2.1). The R-value of each synapse is updated (using Equation 3.27) when that synapse becomes active and the driving neuron that it is connected to is a mutual inhibition contest loser. In this way the evaluation of each synapse is performed according to the Practical TPDP Algorithm (see Appendix Section B.2).

Internal and external exploration (see Section 4.2.3) are facilitated by using the random exploration parameters $\sigma_{\text{swap-TP}}$ and $\sigma_{\text{add-TP}}$ to determine when exploration should occur (Section 4.2.6). To initiate exploration the activity level $a_t(u)$ of each driving neuron u is set to a uniformly distributed random value between 0 and 1. This ensures that each action specification will be chosen with equal probability during exploration. When this is done the output of the mutual inhibition contest winner is also flash reduced. Action specification changes during exploration are facilitated by using the random exploration parameter σ_{change} to determine when new mutual inhibition contests should be initiated in the same manner.

The random exploration parameter σ_{external} is used to select between internal and external exploration when exploration is initiated in non-TP states (see Appendix Section B.2, Lines 50 to 55, and Section 4.2.6).

If the results of internal or external exploration indicate that new TPs should be allocated (see Section 4.2.3), such is performed by adding synapses that connect the identification neurons with the driving neurons.

6.1.7 Allocating Identification Neurons

As well as being able to add synapses that correspond to the allocation of new TPs, Neural TPDP must be able to allocate new identification neurons. This is necessary when the first TP associated with a state is allocated (see Figure 6.32). Similarly, Neural TPDP must be able to deallocate identification neurons when the last TP associated with a state is removed. Driving neurons need not be allocated or deallocated because each of them permanently specifies a different control action.

An interesting alternative Neural TPDP design that does not require identification neuron allocation involves making the driving neurons more complex so that they can perform the state identification function. The driving neurons can be modified to operate as *and-or* gates, first *and-ing* together state lines to identify individual states, and then *or-ing* together the results to combine the identification signals in the usual manner (see Section 6.1.2). Similar neuron designs have been investigated by others for entirely different applications (for example Feldman *et al.*, 1982). If driving neurons of this type are employed, identification neurons are not required at all. The price paid for this simplification is that the mechanisms required to identify each state must be replicated at every synapse related to that state.

6.1.8 Parallel Processing and Neural TPDP

Aside from the fact that Neural TPDP operates like an ACAM (see Section 6.1.3), an important rationale for using Neural TPDP is that such an approach facilitates the parallel processing possible with neural networks. To do so a universal "stack" is not used, as described in Section 4.2.4, to store the information necessary for Q-value and R-value updating. Instead, the relevant information is stored locally by each neuron so that updating can occur in a parallel fashion. Some information still has to be distributed to all of the neurons however. This information includes the immediate costs experienced, the mode of exploration, the fact that an update should be made, and the evaluation function value $V_\mu(i)$ of the state i that the system is in.

6.2 Analysis of Neural TPDP

6.2.1 The Localized Operation of Neural TPDP

As described in Section 6.1.2, Neural TPDP fulfills its function as a controller through the connections between the identification neurons and the driving neurons (see Figure 6.32). It is thus the connectivity of the Neural TPDP network that matters the most. This flexible connectivity is altered based on the synapse weights $w(i, u)$, which indicate the integrity of each connection. This type of operation is very different from conventional neural networks (the popular error propagation method for example, Rumelhart *et al.*, 1986, Fahlman *et al.*, 1987), which are normally fully interconnected with fixed connections, and which use their weights to modulate the signals passing through each connection. The neurons in such networks work together to process the signals input to the network. In contrast, only one identification neuron and one driving neuron in Neural TPDP output a high signal at any one time. Neural TPDP thus functions using *localized operation* (Baker *et al.*, 1992, Schmidhuber, 1990*b*). Other neural network designs that have localized operation include CMAC (Albus, 1975*a*, 1975*b*, Kraft *et al.*, 1992) and radial basis function networks (Poggio *et al.*, 1990, Girosi *et al.*, 1989).

Related to the localized operation of Neural TPDP is the fact that identification neurons and driving neurons output binary signal levels. This is different from the continuous signal levels output by most neural networks.

The main disadvantage of localized neural network designs is that such networks cannot readily make generalizations (Jacobs *et al.*, 1991). Conventional neural networks in contrast excel at making generalizations (Baker *et al.*, 1992). The localized operation of Neural TPDP precludes the formation of generalizations between states and the actions specified in those states[5]. Section 6.2.2 will present a way in which generalization by

[5]This is different from the evaluation function approximation that Practical TPDP also precludes (or

copying (see Section 5.3.1) can be facilitated with Neural TPDP however.

The main advantages of localized neural network designs are that repeated training sessions are not required to facilitate generalization (Baker *et al.*, 1992), and that learning in one region of the input to output mapping does not affect other parts of the mapping (Baker *et al.*, 1992, Ayestaran *et al.*, 1993). In conventional neural network designs, generalization results from the network learning, through repeated presentation of training vectors, how it should adjust its weights to produce the desired input to output mapping for all of the training vectors.

6.2.2 Generalization by Copying in Neural TPDP

Section 5.3.1 described how generalization by copying can be facilitated in Practical TPDP by biasing the selection of the random actions attempted during exploration (see Section 4.2.4 and Appendix Section B.2, Lines 20 to 34, 40 to 43, and 50 to 55) towards actions already specified by TPs in the state space vicinity around the current state. This type of generalization is facilitated in Neural TPDP by modifying the identification neurons so that they operate as enhanced *and-gates*. The goal of this modification is to have each identification neuron output a low-grade signal when the system is in a state close to that which it identifies. Such low-grade signals are passed through synapses to the driving neurons connected to each partially activated identification neuron, and indicate to those driving neurons that they should modify their activity levels $a_t(u)$ to bias the random action selection (see Section 6.1.6).

There are many ways in which low-grade signals can be produced in the identification neurons. One way is to define the identification neuron function as follows:

$$o_t(i) = \prod_{l \in L(i)} [(1-\delta)z_t(l) + \delta] \tag{6.51}$$

at least makes very difficult – see Sections 2.4.4 and 6.1.1).

Where $o_t(i)$ is an *identification neuron output* at time step t, $L(i)$ is the set of state lines connected as inputs to identification neuron i, $z_t(l)$ is the signal level of each state line $l \in L(i)$ at time step t, and δ is the desired low-grade signal level.

Equation 6.51 results in the output of the identification neurons being $\delta^{n_{\text{low}}}$, where n_{low} is the number of input state lines $l \in L(i)$ which have low signal levels. If δ is set to 0.1, and the driving neurons bias their activity levels $a_t(u)$ only when they receive low-grade input signals of at least 0.1, generalization by copying will result between all immediately adjacent states.

Once it has been determined that a driving neuron should bias its activity level during random action selection (see Section 6.1.6), a biasing technique must be employed. Normally when mutual inhibition contests are used to randomly select actions during exploration the activity levels $a_t(u)$ of each driving neuron are set to a random value between 0 and 1 (see Section 6.1.6). To bias the random action selection, n_{bias} random values between 0 and 1 are generated for each driving neuron that has a sufficient low-grade signal, and the highest one is used as the activity level $a_t(u)$ for that neuron. This increases n_{bias}-fold the likelihood of each driving neuron biased in this manner being the victor of the mutual inhibition contest.

6.2.3 Elemental and Composite Actions

The neural network implementation of Practical TPDP brings to light one important consideration in TPDP control, and in control in general. In the Race Track problem presented in Sections 5.1.1 through 5.1.3 the possible actions $U(i)$ in each state i were $u = [u_x, u_y]$, where both u_x and u_y were in the set $\{-a, 0, a\}$, and where a was a constant. Each action was thereby a *composite action* constructed from simpler *elemental actions.* That is, each action u consisted of an elemental action u_x in the x direction, and an elemental action u_y in the y direction. In real controllers there is often a limit to the number of composite actions that can be constructed out of elemental actions. When that limit is reached the action specification in each state must be made using a number of elemental actions or less complex composite actions.

In conventional Q-learning controllers (see Chapter 2) the amount of memory available limits the complexity of the composite actions. The amount of memory required for control is proportional to the number of composite actions that can be constructed from the elemental actions, and this number grows exponentially with the number of elemental action dimensions. Basically, a Q-value must be stored for each composite action in each state. As a result, the amount of memory available determines how complex the composite actions can be.

Other types of memory-based controllers (see Sections 2.3.2 and 2.4.2), ones which do not allocate a memory entry for each state/composite action combination, do not suffer from exponentially increasing memory requirements as the number of elemental action dimensions grows. An example of such memory-based controllers would be a Practical TPDP controller operating as an ACAM (see Section 2.4.2). Such a controller would only allocate memory entries to associate a small number of composite actions with each state (often just one), and it may not associate any actions with some states. In the case of Neural TPDP, each memory entry would correspond to a synapse.

As an ACAM control approach (see Section 6.1.3), Neural TPDP does not experience exponentially increasing memory requirements as the number of elemental action dimensions grows. But the number of composite actions possible with Neural TPDP is limited for another reason. This reason is that Neural TPDP uses one driving neuron for each composite action, and having too many such driving neurons would make implementation of the neural network impractical. The same problem does not occur when Practical TPDP is implemented using tables (see Section 2.3.2), because each table entry can consist of an arbitrary list of elemental action specifications defining a composite action. As a result, the number of possible composite actions is limited only to the number of TPs in Practical TPDP.

Aside from memory usage concerns, complex composite actions can make the learning of an optimal policy more difficult in any controller. If, for example, a TPDP controller is presented with the task of learning an optimal snowboarding policy, the composite actions available could include u_{hl}, a hip elemental action combined with a left thumb elemental action, and u_{hr}, the same hip elemental action combined with a right thumb elemental action. The TPDP controller would then have to determine the Q-values, R-values and weights for composite actions u_{hl} and u_{hr}. Clearly it would be more effective to learn these values for only the hip elemental action u_h.

Conversely, separating complex composite actions into elemental actions (or into smaller composites of elemental actions) results in credit assignment (see Section 2.3.4) being made more difficult. If an undesirable snowboarding outcome occurs, it must be determined whether the hip elemental action u_h or the torso elemental action u_c was responsible for this result. This determination is difficult, and it is normally made using extensive observation of the outcomes when various combinations of elemental actions are attempted. It may be that TPDP, with it Q-values and R-values, is very well suited to credit assignment determination. This is because these values indicate not only the results incurred when an elemental action is taken, but the results incurred when it is not taken. This extra information, which few learning controllers make use of, may be of great use in assigning credit properly. This issue requires further research.

6.3 Biological Plausibility

6.3.1 A Possible Model of Biological Movement Control

Neural TPDP is, arguably and speculatively, a plausible model of biological movement control and refinement. Specifically, Neural TPDP can be viewed as a model of how muscle control signals are learned and generated at the lowest levels of control in the brain. Neural TPDP functions by making synaptic connections between identification neurons that identify states and driving neurons that output action signals. Each such synapse is either maintained or discarded based on evaluations of whether or not the state/action association it makes improves the control of the system. These evaluations are made in a way that solves the credit assignment problem (see Section 2.3.4) – a necessity in biological control. Further, the action specified by each synapse is maintained until another state is encountered in which an action is specified. As a result, a limited number of synapses can define entirely how the system is controlled, and these synapses can be modified based on observations of the system response. This is a very simple but effective approach to movement control and refinement, and as such it may be a plausible model of biological control.

Beyond the intuitive appeal of Neural TPDP as a model of biological movement control, Neural TPDP has the following specific attributes which add to its plausibility as a biological model. Many of these attributes were discussed in relation to Q-learning and the theoretical form of TPDP in Sections 2.3.1, 2.3.2, 2.3.3, 2.3.5, 3.4.1 and 3.4.2.

Minimal Use of Memory

As a direct DP control approach, Neural TPDP avoids the extensive use of memory that is required for indirect DP control (see Sections 2.2.2 and 2.3.1). The memory savings inherent in Neural TPDP go much further however. Because Neural TPDP operates as a

ACAM (see Sections 2.4.2 and 6.1.3), it only allocates memory at those states which are actually entered during control of the system. It does not require a full, multi-dimensional table to produce the non-linear control advantages that result from memory-based control (see Section 2.3.2). Further, because of the way in which actions are specified and then maintained in TPDP (see Section 3.1.1), Neural TPDP can achieve optimal control by allocating memory at only a limited number of uniform region boundary states (see Section 3.4.2).

Reduced memory usage in Neural TPDP means that the number of synapses required for control are reduced. By making a limited number of connections between identification neurons and driving neurons, Neural TPDP can achieve optimal control. Further, because reduced memory usage increases the rate of DP learning (see Section 2.4.1), reduced numbers of synapses increase the rate of learning in Neural TPDP.

Rapid Learning of Preliminary Policies

Section 5.2.2 illustrated that one of the main attributes of Practical TPDP is that it learns effective preliminary policies very rapidly. Neural TPDP, as a way of implementing Practical TPDP, shares this attribute, and it is an important one in terms of making Neural TPDP a plausible model of biological movement control. This is because the early learning of reasonably effective policies has obvious survival value for biological systems.

In many control applications, biological and otherwise, the ability to rapidly learn effective preliminary policies may be the most important attribute of a controller. Once such a preliminary policy is learned, it can be refined and optimized as more learning effort is exerted. As an example, consider a casual and a professional snowboarder. The former wishes to rapidly acquire a level of competence in the sport by learning an effective but suboptimal policy. The latter wishes to highly refine his skills in the sport, and is willing to go through considerable training effort to do so. In the context of Neural TPDP, the professional performs much more extensive exploration, and allocates a larger number of synapses as he seeks to learn an absolutely optimal policy.

Localized Operation and Reinforcement Learning

As explained in Section 6.2.1, the neurons in Neural TPDP operate in a localized way. Further, Neural TPDP, as a form of Q-learning, is a reinforcement learning approach (see Section 2.3.3). These two characteristics of Neural TPDP mean that it can learn optimal policies with each neuron operating in parallel, and with only limited amounts of information being passed between the neurons (other than the direct identification neuron to driving neuron synaptic signaling). Such localized operation and limited information passing are necessary for any plausible model of biological movement control (Alkon, 1989, Alkon *et al.*, 1990, Schmidhuber, 1990*a*). Neural network designs which do not operate in a localized way, like the ubiquitous error propagation method (Rumelhart *et al.*, 1986), require learning supervisors and complex information passing routes that are unlikely to exist in biological movement controllers.

The only information that the neurons in Neural TPDP require from external sources is the state information and a universally distributed reinforcement signal. The reinforcement signal indicates the immediate costs $c(i, u)$ incurred as the system moves about the state space.

The internal information that Neural TPDP passes includes the mode of exploration, the evaluation function value $V_\mu(i)$ of the state i that the system is in, and the fact that an update of Q-values, R-values and weights should be made. The last item of information can be inferred indirectly by each driving neuron if they observe the action specification changes indicated by their mutual inhibition inputs (see Figure 6.32), and combine that information with their knowledge of the exploration mode (see Appendix

Section B.2 and Figure 4.3).

In summary, aside from the explicit network connections shown in Figure 6.32, Neural TPDP operates with only the following information being passed between the neurons: the immediate costs $c(i,u)$, the mode of exploration, and the evaluation function value $V_\mu(i)$ of the state i that the system is in. All three of these items of information are universally distributed, and can thus be equated to hormone levels in a biological movement controller. As a result, the neurons in Neural TPDP operate in a highly independent manner. In contrast, most other neural network designs involve complex supervisory systems and information flows that do not seem biologically plausible[6].

Continuous State Space Control

As explained in Section 3.1.4, TPDP operates best in continuous state space control applications. As biological movement controllers operate entirely in continuous state spaces, this attribute of Neural TPDP is an important part of its plausibility as a model of biological control.

Adaptive Control

As a form of Q-learning, Neural TPDP is an adaptive control approach (see Section 2.3.5). This attribute is an essential one in any model of biological movement control because biological controllers are clearly adaptive.

Composite Action Learning

Any plausible model of biological movement control must have the ability to learn optimal policies using elemental actions, or partial composites of elemental actions (see

[6]Fidelity to biological systems is admittedly *not* the goal of most artificial neural models however. Practical application on non-biological problems is instead the concern.

Section 6.2.3). This is because biological systems, with their enormous number of elemental actions, could not possibly operate using composite actions constructed from every combination of elemental actions. Unfortunately, although Neural TPDP seems well suited to learning with elemental actions, this capability has not yet been thoroughly investigated.

6.3.2 Increasing the Localized Operation of Neural TPDP

One aspect of Neural TPDP which has not been discussed up to this point is how the *policy synapse* (the policy TP) $\mu(i)$ is determined for each state i. There are a number of ways in which policy synapses can be determined, but all are rather involved. One method of determination will be presented in this Section, and then a simplifying approach to Neural TPDP which does not require policy synapse determination will be described. This simplification will be combined with an additional simplification to increase the independent operation of the Neural TPDP neurons. Such increased independent operation makes Neural TPDP even more biologically plausible.

One way to determine the policy synapse is to have each identification neuron i occasionally output some sort of special signal (a bursty one perhaps) when the system is in the state i identified by that neuron. This special signal indicates to all of the driving neurons activated by that identification neuron that they should modulate their activity levels $a_t(u)$ not with a random value (see Section 6.1.4), but with $1 - \frac{Q(i,u)}{Q_{\textbf{max-possible}}}$. This modulation will result in the mutual inhibition contest victor being the driving neuron u whose synapse has the lowest Q-value $Q(i,u)$. This result, which indicates which synapse is the policy synapse $\mu(i)$ in state i, can then be noted at all network locations where this information is of consequence.

To avoid the complexity of policy synapse determination, Neural TPDP can be modified to operate without policy synapses $\mu(i)$, or the evaluation function values $V_\mu(i)$

determined from them; $V_\mu(i) = Q(i, \mu(i))$. This is done, at each state i where an update is performed, by updating not with $V_\mu(i)$ but with the Q-value $Q(i, u)$ of the action u that is actually specified at that state (see Appendix Sections B.2 and B.3).

A Neural TPDP implementation difficulty similar to that of policy synapse determination is the problem of informing each synapse as to whether or not other synapses are associated with the same state that it is. This *synapse allocation knowledge* can be acquired by each synapse through inspection of the mutual inhibition inputs to the driving neuron to which they are attached. But this determination is again rather involved. To avoid this complexity, the use of synapse allocation knowledge can also be eliminated from Neural TPDP.

Eliminating both the use of policy synapses and the use of synapse allocation knowledge requires changes to the Practical TPDP Algorithm, as well as to the Stack Updating Procedure it calls (see Section 4.2.4, and Appendix B). The necessary changes are presented in Figures 6.33 and 6.34.

Regarding the changes, when internal exploration is initiated at a TP state, a policy action is first selected, and then a random action is reselected (Lines 22 and 25 of the modified Practical TPDP Algorithm, see Figure 4.3). This is done so that stack updating can be performed using a Q-value of an existing TP (Line 23). If this were not done, the randomly chosen exploration action might not have a TP associated with it, and the stack could not be updated. The other algorithm modifications will not be explained in detail as they are fairly straight-forward.

Figure 6.35 presents the results of using the modified Practical TPDP Algorithm to solve the Race Track problem described in Sections 5.1.1 to 5.1.3 (using the parameters defined in Sections 5.1.4 and 5.1.5). One further modification had to be made to the Practical TPDP Algorithm to obtain these results. The random selection of TP actions had to be biased towards TPs with high weight values $w(i, u)$ when no exploration was

```
1.  randomly choose starting state $s_0 \in S_S$
2.  choose a starting action $u_0 \in U(s_0)$
3.  if (state $s_0$ has a TP): 'none' $\Rightarrow$ $explore$
4.  otherwise: 'external' $\Rightarrow$ $explore$
5.  $0 \Rightarrow t$, $0 \Rightarrow t_{\text{update}}$
10. while $s_t$ is not an absorbing state $S_A$:
11.     if (state $s_t \neq s_{t-T}$) or (state $s_t = s_{t-T}$ for time $T_{\text{stuck}}$):
20.         if (state $s_t$ has a TP):
21.             if ($\sigma_{\text{swap-TP}} > 0$):
22.                 randomly choose action $u_t$ from TP actions in $U(s_t)$
23.                 update-stack($explore$, $t$, $Q(s_t, u_t)$, $V_{\text{expected}}$)
24.                 $Q(s_t, u_t) \Rightarrow V_{\text{expected}}$
25.                 randomly re-choose action $u_t \in U(s_t)$
26.                 'internal' $\Rightarrow$ $explore$
27.                 push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
28.             otherwise if ($explore \neq$ 'none') or ($t > t_{\text{update}}$):
29.                 randomly choose action $u_t$ from TP actions in $U(s_t)$
30.                 update-stack($explore$, $t$, $Q(s_t, u_t)$, $V_{\text{expected}}$)
31.                 $Q(s_t, u_t) \Rightarrow V_{\text{expected}}$
32.                 'none' $\Rightarrow$ $explore$
33.                 $t + \sigma_{\text{delay}} \Rightarrow t_{\text{update}}$
34.                 push-on-stack($t$, $s_t$, $u_t$, 'true')
35.             otherwise: push-on-stack($t$, $s_t$, $u_t$, 'false')
40.         if (state $s_t$ has no TP) and ($explore \neq$ 'none') and ($\sigma_{\text{change}} > 0$):
41.             if ($explore =$ 'external'): flush-stack
42.             randomly choose action $u_t \in U(s_t)$
43.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
50.         if (state $s_t$ has no TP) and ($explore =$ 'none') and ($\sigma_{\text{add-TP}} > 0$):
51.             flush-stack
52.             if ($\sigma_{\text{external}} > 0$): 'external' $\Rightarrow$ $explore$
53.             otherwise: 'internal' $\Rightarrow$ $explore$
54.             randomly choose action $u_t \in U(s_t)$
55.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
60.     $t + T \Rightarrow t$
61.     $\frac{1}{\gamma}(V_{\text{expected}} - c(s_t, u_t)) \Rightarrow V_{\text{expected}}$
62.     observe system for new state $s_t$
63. update-stack($explore$, $t$, $s_t$, $V_{\text{expected}}$)
```

Figure 6.33: The Localized Operation Practical TPDP Algorithm

```
                [Parameters passed: $explore$, $t$, $V_{\text{update}}$, $V_{\text{expected}}$]
1.  while (time at top of stack $t_s \geq t$): pop-off-stack($t_s, i_s, u_s$, $specified_s$)
2.  $V_{\text{update}} \Rightarrow C_{\text{total}}$
10. while (there are entries in stack):
11.     pop-off-stack($t_s, i_s, u_s$, $specified_s$)
12.     while ($t > t_s$):
13.         $\gamma C_{\text{total}} + c(i_s, u_s) \Rightarrow C_{\text{total}}$
14.         $t - T \Rightarrow t$
20.     if ($explore$ = 'none'):
21.         for (each TP action $u \in U(i_s)$ in state $i_s$):
22.             if ($u = u_s$):
23.                 $(1-\alpha)Q(i_s, u_s) + \alpha C_{\text{total}} \Rightarrow Q(i_s, u_s)$
24.                 if ($Q(i_s, u_s) > R(i_s, u_s)$):
25.                     $w(i_s, u_s) + 1 \Rightarrow w(i_s, u_s)$
26.                     if ($w(i_s, u_s) > w_{\max}$): $w_{\max} \Rightarrow w(i_s, u_s)$
27.                 otherwise:
28.                     $w(i_s, u_s) - 1 \Rightarrow w(i_s, u_s)$
29.                     if ($w(i_s, u_s) = 0$): remove the TP
30.             if ($specified_s$ = 'false') or ($u \neq u_s$):
31.                 $(1-\alpha)R(i_s, u) + \alpha C_{\text{total}} \Rightarrow R(i_s, u)$
40.     if (memory can be allocated) and
        [(($explore$ = 'internal') and $V_{\text{update}} < V_{\text{expected}}$)) or ($explore$ = 'external')]:
41.         allocate a new TP at state $i_s$ with action $u_s$
42.         $C_{\text{total}} \Rightarrow Q(i_s, u_s) \Rightarrow R(i_s, u_s)$
43.         $w_{\text{initial}} \Rightarrow w(i_s, u_s)$
```

Figure 6.34: The Localized Operation Stack Update Procedure

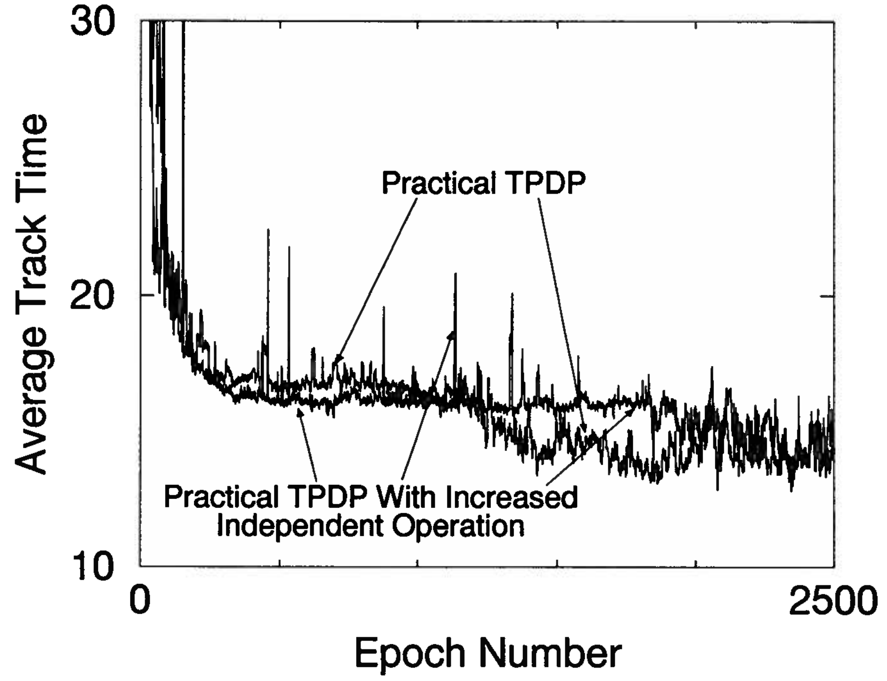


Figure 6.35: Performance of Practical TPDP with Increased Localized Operation

occurring. This was necessary to ensure that the Q-values used for stack updating were predominantly ones whose high weight values indicated that they were associated with near optimal actions. In effect, this biasing resulted in evaluation function values being used for the updating.

To bias the random action selection, $w(i,u)^2$ random values from 0 to 1 were generated for each driving neuron u, where $w(i,u)$ was the weight of the active synapse attached to each driving neuron. The highest of these values was then used to modulate the activity level $a_t(u)$ of that driving neuron. This resulted in the probability of each action being selected being proportional to the square of $w(i,u)$ (see Section 6.1.6).

Figure 6.35 indicates that the modified Practical TPDP Algorithm performed very well. The learning transition (see Section 5.2.2) occurred later than it did with conventional Practical TPDP, but the average track time curve was generally flatter. Considering how little information was passed between neurons in the modified algorithm, it is surprising that it performed as well as it did.

In Section 6.1.7 it was suggested that Neural TPDP could be made simpler if identification neurons were not used at all, and the driving neurons were instead designed to operate as *and-or* gates. The driving neurons could then perform the state identification. As described in Section 6.1.7, this design change would eliminate the need to determine when identification neurons should be allocated and deallocated. This change would also increase the independence of the neurons in Neural TPDP because any identification neuron allocation mechanism would require synapse allocation knowledge. Using a network design that does not employ identification neurons would eliminate the need for this knowledge, as well as the information routes that pass it.

Chapter 7

Practical TPDP as Part of a Complete Hierarchical Controller

This Chapter will describe how Practical TPDP can be incorporated into a complete hierarchical controller. The attributes and capabilities of such a controller will also be presented.

7.1 Practical TPDP Facilitates Lower Movement Control

It was speculated in Section 6.3.1 that Neural TPDP may be a plausible model of lower movement control in biological systems. If such is the case, the question that arises is how Neural TPDP would function as part of a complete hierarchical controller. This Chapter will discuss this issue, and while it will do so frequently in the context of biological controllers, the arguments presented could be equally well applied to the incorporation of Practical TPDP in any hierarchical controller. The terms "Practical TPDP" and "Neural TPDP" will therefore be used interchangeably in this Chapter, with the latter indicating a biological controller context.

7.1.1 Context Lines

Inherently, to function as a low level movement controller within a hierarchical controller (Albus, 1983, 1988, 1991), Neural TPDP must be able to perform more than one control task on the same system. If it cannot, the hierarchical control layers above it, which decide which specific low level tasks should be performed, have no purpose.

An effective way to facilitate the performance of more than one task with the same

Neural TPDP controller is to have *context lines* (similar to input from the "plan units" in Massone *et al.*, 1989) descending from higher levels of control. Each context line is associated with a single task, and when higher levels of control have decided that a particular task should be performed, the appropriate context line is made high. As only one task can be performed at a time, only one context line can be high at a time. Context lines are used with state lines (Singh made a similar combination, 1991*b*) as input to the identification neurons of a Neural TPDP controller (see Section 6.1.2 and Figure 6.32). A single context line is used as an input to each identification neuron, and because of the *and-ing* function of these neurons, it will act as an identification "enabler". By connecting the same context line to all of the identification neurons involved in each control task, the context lines can be made to act as task enablers, preparing complete sets of identification neurons to make the state/action associations (through their driving neuron synaptic connections – see Section 6.1.2) appropriate for each task.

As Neural TPDP learning progresses, the set of identification neurons enabled by each context line learns the optimal policy for the task associated with that context line. This learning is based on the immediate costs incurred whenever that context line is high. It is therefore assumed that there is some consistent relationship between when the higher levels of control activate each context line, and the situation that the system is in (as reflected in the immediate costs incurred). For example, it is assumed that the higher levels of control would not indicate a snowboarding task when the system is relaxing on the beach and requires a beer.

If Practical TPDP is implemented as a memory-based controller (see Section 2.3.2), the context line function can be included in the controller by making use of additional table dimensions.

7.1.2 A Sketch of the Complete Hierarchical Controller

The complete model of a controller incorporating Neural TPDP is that of a hierarchical controller where the abstract control decisions are made, based on sensor information, at higher levels of control. As a result of these abstract control decisions, a context line is activated that enables a set of identification neurons. These identification neurons are individually activated based on state line signals, and specify control actions through the driving neurons (see Section 6.1.2 and Figure 6.32). The set of identification neurons thus enabled is the set which makes the appropriate state/action associations for the control task indicated by the context line. Neural TPDP learns these associations by observing the costs incurred whenever each context line is high, and by modifying its synapses accordingly (see Sections 6.1.2 and 6.1.5).

7.2 Using Higher Level Knowledge

In a hierarchical controller incorporating Neural TPDP, the relationship between the levels of control extends beyond the high levels deciding which context lines should be activated (see Section 7.1.1). The high levels can also aid the lowest level (Neural TPDP) in learning optimal policies (the use of high level knowledge has been investigated by, among others, Lin, 1991*a*, 1991*b*, Utgoff *et al.*, 1991). Sections 7.2.1 through 7.2.5 will describe some ways in which this can occur.

7.2.1 Guided Learning

One way in which higher levels of control can aid the lowest level in learning optimal policies is by guiding the exploration that occurs in the lowest level. If the higher levels possess some knowledge[1] about what an optimal policy is, in general terms, this knowledge can be passed to the lowest of control to direct exploration. This is called *guided learning.*

As a general example of guided learning, consider a novice snowboarder. The task confronting that snowboarder is to learn an optimal snowboarding policy. If the snowboarder has abstract knowledge of generally how to position his body, that knowledge can be passed to his lower movement controller through conscious positioning and conscious exertions of force. Such conscious intervention biases learning exploration, and can greatly increase the rate of learning. If such were not the case, the jobs of many a snowboarding instructor (whose verbal messages guide conscious positioning) would be in jeopardy.

7.2.2 Implementing Guided Learning in Practical TPDP

In the case of Neural TPDP, guided learning is facilitated by having the higher levels of control transmit signals down binary *biasing lines* connected, one each, to the driving neurons. These lines bias the mutual inhibition contests that are an essential part of exploration (see Section 6.1.6) by increasing the activity level $a_t(u)$ of the driving neuron u to which they are connected (see Section 6.1.4). If higher levels of control possess knowledge[1] that a certain action will result in good performance if applied in a certain state, and the system enters that state, a high signal is transmitted down the biasing line connected to the driving neuron which specifies the appropriate action. This increases the activity level $a_t(u)$ of the driving neuron, which in turn increases the likelihood that the desired action will be specified as a result of a mutual inhibition contest at that state (see Section 6.1.2).

[1]How the higher levels of control obtain and store this knowledge is immaterial – this description is concerned only with how such knowledge can be passed down to the lowest level of control (Neural TPDP) and used effectively.

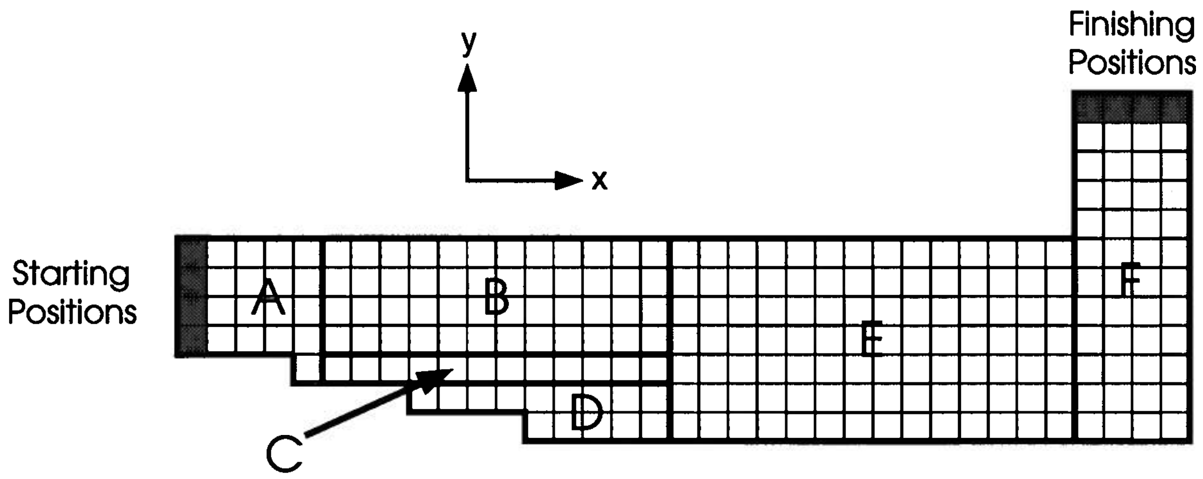
To control movement, Neural TPDP learns state/action associations which direct the controlled system through its state space. Initially, before optimal state/action associations have been learned, higher levels of control may have knowledge of the general trajectory that the system should follow. That knowledge can be utilized, through guided learning, to direct learning exploration along the general trajectory. As learning progresses along the general trajectory, Neural TPDP "fleshes out" the trajectory by adding and modifying synapses that define specific optimal state/action associations (see Sections 6.1.2 and 6.1.5) along it. Optimal control can be achieved by this means – without exhaustive exploration being necessary.

Figure 7.37 presents the results of applying Practical TPDP, with guided learning, to the Race Track problem described in Sections 5.1.1 to 5.1.3 (using the parameters defined in Sections 5.1.4 and 5.1.5). Guided learning was facilitated by biasing the random selection of actions during exploration so that the actions shown in Figure 7.36 were selected 50% of the time. The actions shown in Figure 7.36 were not optimal, they were simply intelligent guesses (made by this researcher) at the optimal actions. Figure 7.37 indicates that guided learning was indeed very effective in increasing the rate of learning.

7.2.3 Gross Inverse Models

As a direct DP control approach, Practical TPDP does not develop an explicit system model to facilitate control (see Sections 2.2.2 and 2.3.1). In Section 2.3.1 it was stated that an explicit model of a system is very useful if that system has to perform many tasks. This is because that model can be utilized for every task, and the modeling knowledge acquired learning the optimal policy for each task can be used by the others.

The problem with such a general purpose model is that, if it is the only model used in all tasks, it must be highly resolved. That is, it must have fine enough resolution



A:	u_{bias}	$=$	$[1,0]$	**E:**	u_{bias}	$=$	$[-1,1]$	if $\dot{x} > 2.5$
B:	u_{bias}	$=$	$[1,-1]$				$\emptyset$	otherwise
C:	u_{bias}	$=$	$[1,0]$	**F:**	u_{bias}	$=$	$[-1,1]$	if $\dot{y} < -2.5$
D:	u_{bias}	$=$	$[1,1]$				$\emptyset$	otherwise

Figure 7.36: The Biased Action Specifications Used for Guided Learning

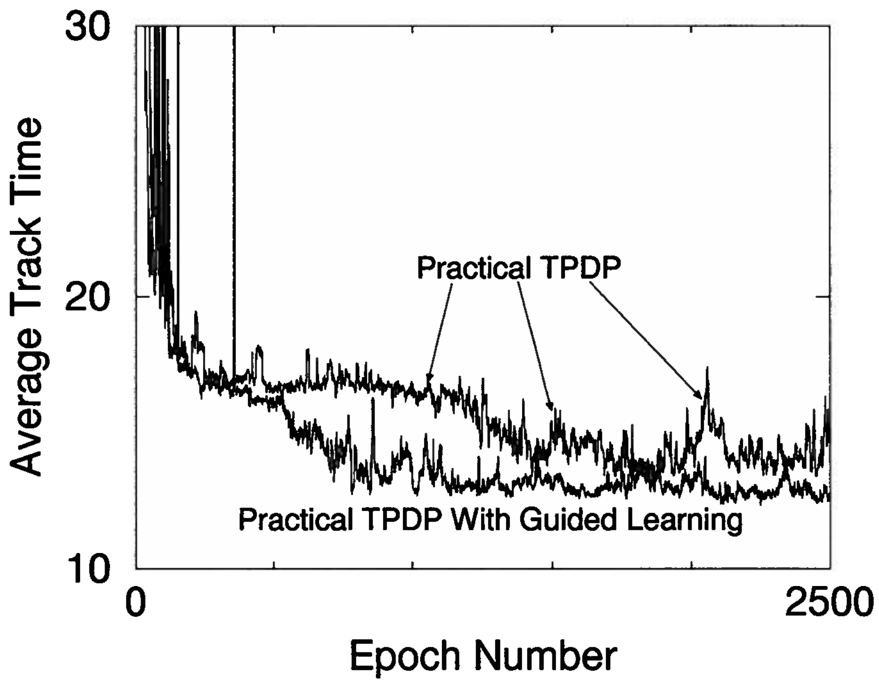


Figure 7.37: Performance of Practical TPDP During Guided Learning

to be of use in each and every task. This could result in a very prohibitive memory requirement. To reduce this memory requirement, a variable resolution model could be employed (Moore, 1991). Such a model would have increased resolution in the states associated with those tasks that the system must perform. But if it did, there is some question as to whether or not it is really a general purpose model. That is, few knowledge sharing benefits may result from combining the variable resolution models determined for each task.

Alternatively, a controller can be constructed that has two levels of modeling. In the context of Practical TPDP, such a controller would operate as follows. Higher levels of control would employ Practical TPDP, at the lowest level of control, to develop a *gross inverse model* (for further description of inverse models see Aboaf *et al.*, 1988, Moore, 1992). This model would be developed by having Practical TPDP learn policies that move the system to a general set of states or *gross positions* (by doing so gross inverse models are similar to the variable temporal resolution models investigated by Singh, 1992*a*). The movement to each such gross position would constitute a single control task, and would have a context line (see Section 7.1.1) associated with it – a *gross position context line.* The controller would learn a complete gross inverse model by learning the policy required to reach every interesting region of the state space. The resulting gross inverse model would provide the controller with general knowledge of how to control the system.

Once a gross inverse model has been developed, that model can be used by high levels of control to generally direct the system when policies for specific tasks must be learned (this is similar to the approaches taken by Singh, 1991*a*, 1991*b*, 1992*b*, Schmidhuber, 1990*b*). In other words, gross inverse models facilitate guided learning (see Sections 7.2.1 and 7.2.2).

In Section 7.2.2 the use of bias lines that activated driving neurons was described as a way to facilitate guided learning in Neural TPDP. An alternative approach is to modify the neural design so that multiple context lines (see Section 7.1.1) can be activated simultaneously. Specifically, modify the design so that one primary context line can be activated fully to indicate what specific task is being performed, and secondary context lines can be partially activated to bias the random action selections during exploration. If a gross inverse model exists, higher levels of control can partially activate, at the appropriate time, gross position context lines associated with the intermediate gross positions that the system should reach during completion of a specific task. This will partially activate one or more identification neurons, which will in turn partially activate a number of driving neurons. The end result will be the biasing of the mutual inhibition contest between the driving neurons (see Section 6.1.2), biasing the random action selection to actions that take the system to the intermediate gross positions.

7.2.4 Optimality Criteria and Higher Knowledge

Section 7.2.1 described how higher levels of control can aid the lowest level in learning optimal policies by using knowledge of optimal actions to bias the random action selection during exploration. This biasing approach was called guided learning. There is another way in which high level, abstract knowledge can aid low level learning. If the optimal actions are not known, but extensive general knowledge of the desired system behavior exists in higher levels of control[1], this knowledge can be used to increase the rate at which the optimal policy is learned (Barto, 1992).

Consider the Race Track problem described in Sections 5.1.1 to 5.1.3. The car in that problem experiences the same immediate cost every time step until it reaches the finishing positions. When it has reached those positions it experiences no further costs. As a result, the optimal behavior of the car is to reach the finishing positions as soon as possible. Practical TPDP and conventional Q-learning learn this policy by backing-up the costs (see Section 2.1.4) that will be experienced before the car reaches the finishing positions. Accurate estimates of these costs, in the form of evaluation function values (see Section 2.1.2), are initially learned in the states that the car enters immediately before encountering the finishing positions. These accurate evaluation function values are then used in the learning of other accurate evaluation function values in states progressively further away from the finishing positions. As accurate evaluation function values are learned for each state, optimal actions can be determined. The learning of the optimal policy thus backs-up from the finishing positions.

Because the learning of the optimal policy backs-up from the finishing positions, protracted periods of *back-up time* can pass before optimal actions can be determined for states distant from those positions. If higher levels of control have extensive general knowledge of the desired system behavior however, this back-up time can be reduced. For example, if it is known that the car in the Race Track problem should make a left turn (even though the actions facilitating that turn are not known), that knowledge can be used to modify the optimality criteria that the controller is presented with. Figure 7.38 indicates a roughly optimal path that the car should follow for the Race Track problem described in Sections 5.1.1 to 5.1.3 (using the parameters defined in Sections 5.1.4 and 5.1.5). Figure 7.39 shows the average track time results (see Section 5.1.6) when Practical TPDP was applied to this Race Track problem, and the immediate costs at car positions along the path shown in Figure 7.38 were reduced to one quarter their normal level (see Section 5.1.4).

Figure 7.39 indicates that the learning transition (see Section 5.2.2) occurred much sooner when lower immediate costs were incurred at car positions along the path shown in Figure 7.38. This is because the path provided additional optimality information, and it acted as a source from which accurate evaluation function values could be backed-up. As a source of accurate evaluation function values, it was much closer to a large number of states than the finishing positions were, so the backing-up time was reduced accordingly.

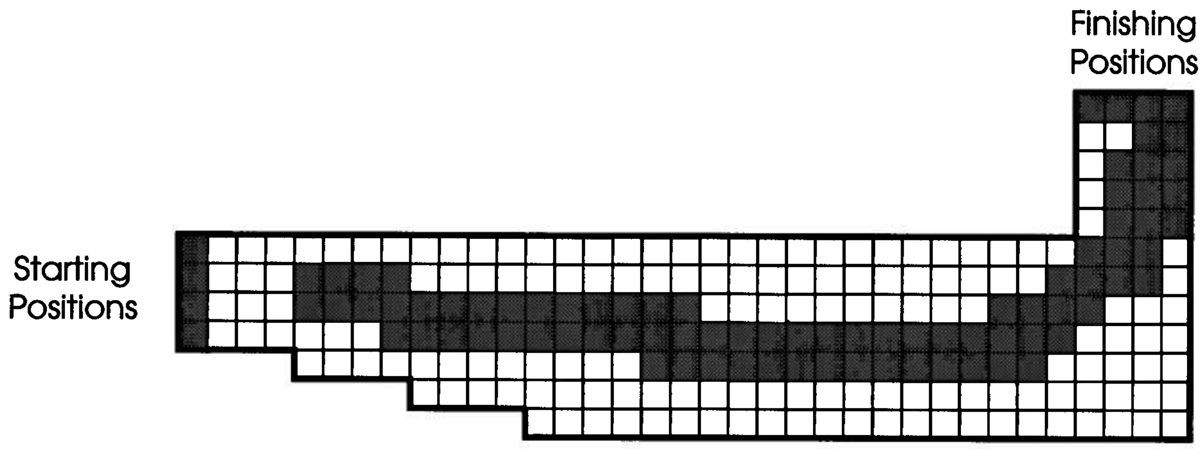


Figure 7.38: A Roughly Optimal Path on the Race Track

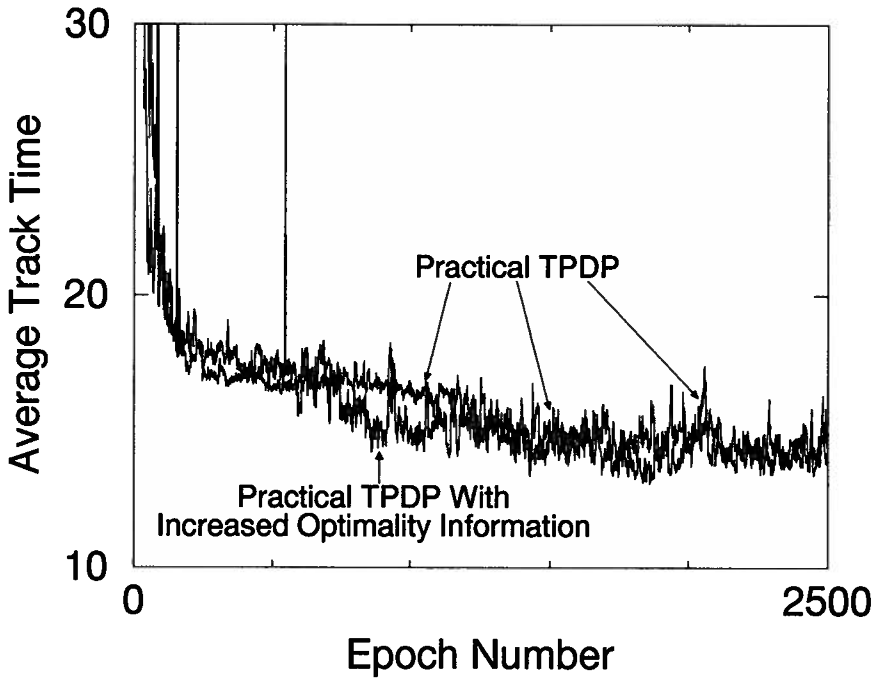


Figure 7.39: Performance of Practical TPDP with Increased Optimality Information

The use of additional optimality information in the Race Track problem illustrates how optimality information from higher levels of control can aid in the learning of optimal policies. If prudent (or optimal) state space trajectories which lead to the arrival at goal states are known, but the actions that will result in those trajectories are not known, those trajectories can be incorporated into the optimality criteria. This is done by reducing the immediate costs experienced by the lowest level of control when the system is following those trajectories.

7.2.5 Dynamic Optimality Criteria

As learning progresses in a controller which incorporates Practical TPDP, the controller can continually make abstract, high level observations (or predictions) about the optimal behavior of the system. These observations may include the determination or expansion of prudent state space trajectories (see Section 7.2.4), where expansion may involve identifying trajectories that lead to other trajectories. All of this knowledge can be utilized in the learning process by dynamically modifying the optimality criteria presented to the lowest level of control (Practical TPDP).

In fact, optimal control can be viewed as a race between higher levels of control and the lowest level of control. As higher levels continually develop more comprehensive and informative optimality criteria, and the lowest level strives to learn the optimal policy for the optimality criteria that the higher levels have developed at any one time.

In a complete hierarchical controller, the emphasis should be placed on the higher levels of control rapidly developing the optimality criteria. This is because even small enhancements in the optimality knowledge can drastically reduce the learning time required to learn optimal policies. The lowest level of control operates in highly resolved, continuous state spaces, and learning at this level is inherently laborious, requiring exhaustive experimentation and exploration. At higher levels however, where the optimality criteria is developed, the state space is abstracted into large, discrete, grossly defined blocks. High level control frequently consists of polar decisions. One such decision, "If I lift my snowboard nose higher, I will turn quicker", can be made without exhaustive experimentation, and when incorporated into the optimality criteria it can have enormous impact on how rapidly the optimal policy is learned at the lowest level.

It would seem that human intelligence is a fine example of the importance of emphasizing the development of optimality criteria (at the highest levels of control) over optimal policy determination (at the lowest level). Humans develop incredibly complex optimality criteria which facilitate the learning of policies for amazingly involved tasks. The lower movement controllers used to perform these tasks are little different from those found in many other mammals (especially other primates), but the tasks humans perform are far more involved. This difference is entirely a result of humans developing more complex optimality criteria.

Chapter 8

Conclusion

8.1 The Main Benefit of TPDP

The main benefit of TPDP is that it learns near optimal policies very rapidly. This was demonstrated throughout Chapter 5 in applications of Practical TPDP. As described in Section 5.2.1, Practical TPDP learned near optimal policies for the Race Track problem much more quickly than conventional Q-learning.

The main reason that Practical TPDP rapidly learns near optimal policies is that a few initial TP allocations (made during early learning exploration) can direct the system along state space transition routes which are generally good (see Section 5.2.2). After such routes have been discovered, further TP allocations can be made along them which "flesh out" the policy, making it optimal at all points. This sequence of events results in Practical TPDP very quickly directing its learning effort to regions of the state space where that effort will produce the best results.

The hazard of learning optimal policies in this manner is that the learning effort can sometimes be concentrated on suboptimal state space transition routes which were discovered early on, while undiscovered, truly optimal routes are ignored. The external exploration of Practical TPDP is a way of preventing this (see Section 4.2.3).

Aside from the way in which it rapid focuses learning effort, Practical TPDP also learns near optimal policies quickly because it makes evaluation function value back-ups (see Section 2.1.4) between states which are widely separated (see Chapter 3). This results in faster learning.

It was speculatively suggested in Section 6.3.1 that the rapid learning of good preliminary policies makes TPDP a plausible model of biological movement control and refinement.

8.2 The Main Disappointment of TPDP

The main disappointment of TPDP was that it did not, in practise, result in the significant memory usage reductions that were hoped for (see Chapter 5). The reason for this was that superfluous TPs were continually added to the state space after a near optimal policy had basically been learned – a result of the way in which Practical TPDP operates (see Section 5.4.2). In Sections 5.4.3 to 5.4.6 a number of improvements were suggested and demonstrated which combat this problem.

The absence of a significant reduction in memory usage in TPDP is exacerbated by the fact that, for each state/action association made by a TP, more memory is required for TPDP than for Q-learning. That is, for every TP, two floating point values (the Q-value and R-value) and one integer (the weight) must be stored. In contrast, Q-learning requires only one floating point value (the Q-value) for each state/action association. Further, to take advantage of the sparsity of TPs in the state space, Practical TPDP is best implemented with an ACAM (see Sections 2.4.2 and 3.4.2). This requires the additional storage of two integers (the discretized state and action) as ACAM address fields. Q-learning can also be implemented with an ACAM if at least one integer (the discretized state) is stored as an address field. Assuming that floating point values require twice the storage space of integers, and ignoring the many different ways in which both Practical TPDP and Q-learning can be implemented with ACAMs, Practical TPDP must allocate no more than 40% of the maximum number of TPs (one for each possible state/action combination) to require less memory in a given application than Q-learning. This was achieved in both of the applications of Practical TPDP described in Chapter 5.

8.3 Direct DP Optimal Control with TPDP is Practical

Adaptive optimal control is very desirable, but DP approaches to such control, while theoretically possible, are often impractical. The main reasons are that they require too much memory, too much learning time, or both. Direct DP approaches like Q-learning require less memory than indirect approaches (see Sections 2.2.1 and 2.2.2), but often require more learning time to compensate for the lack of an explicit system model. As described in this work however, TPDP is a direct DP approach that increases the rate of learning by focusing the learning effort and increasing the distance over which back-ups are made (see Section 8.1). This increase in the rate of learning could make TPDP a practical alternative in many control applications.

If Practical TPDP is incorporated into a complete hierarchical controller (see Chapter 7), the rate of learning can be increased still further by employing guided learning (see Section 7.2.1), and by passing high level optimality knowledge to the lowest level of control – the level at which Practical TPDP operates (see Section 7.2.4).

8.4 Contributions of this Work

The main contribution of this work is the development of TPDP. This development includes proof of the fact that TPDP is sure to achieve minimal TP optimal control (see Chapter 3), as well as formulation of the Practical TPDP Algorithm (see Chapter 4). No proof exists that Practical TPDP is sure to achieve minimal TP optimal control, but the Theorems of Chapter 3 were developed in such a way that they make the achievement of minimal TP optimal control seem highly likely when Practical TPDP is employed.

In addition to the development of TPDP and the Practical TPDP Algorithm, the following contributions were also made:

1. "Generalization by copying" was developed and demonstrated as a way of facilitating generalization in Practical TPDP (see Sections 5.3.1 and 5.3.2).

2. Neural TPDP was developed as a neural implementation of Practical TPDP (see Chapter 6). This development included the design of a neural model which could be used for control (see Section 6.1.2), a neural mechanism that facilitated mutual inhibition contests (see Section 6.1.4), and a neural mechanism that facilitated generalization by copying (see Section 6.2.2).

3. Incorporation of Practical TPDP into a complete hierarchical controller was investigated (see Chapter 7), and the ways in which high level control knowledge could be made use of in such a controller were developed and demonstrated. These included "guided learning" (see Section 7.2.1), "gross inverse models" (see Section 7.2.3) and the use of high level optimality knowledge (see Section 7.2.4).

4. A number of approaches were developed and demonstrated that combat the allocation of superfluous TPs in Practical TPDP (see Sections 5.4.2 to 5.4.6).

8.5 Future Work

There are a number of ways in which this work could be continued:

1. Complete proof that Practical TPDP achieves minimal TP optimal control could be developed – a daunting task.

2. Techniques other than generalization by copying (see Section 5.3.1) could be developed which further extend the ability of TPDP to make generalizations.

3. The problem of the allocation of superfluous TPs in Practical TPDP (see Section 5.4.2) could be more fully investigated, possibly leading to the development of a complete strategy to combat this effect.

4. The incorporation of Practical TPDP into a complete hierarchical controller could be more fully investigated (see Chapter 7), possibly leading to the use of TPDP ideas in higher levels of control. Specifically, TPs could be used to specify actions of varying abstractness at all levels of control. This is a particularly promising direction.

In general, the idea behind TPs is expressed as: "Try this for a while and see what happens". TPDP provides a framework in which controllers that operate using this principle can be developed.

Glossary

Addition Rule: a rule guiding the addition of new TPs (see Section 3.3.8).

DP: dynamic programming.

DP element: a memory entry associated with a single state for the purposes of dynamic programming control.

Neural TPDP: a neural network implementation of Practical TPDP (see Chapter 6).

Practical TPDP: a specific practical approach to TPDP (see Chapter 4).

Practical TPDP Algorithm: an algorithm that facilitates Practical TPDP control.

Q-learning: a direct DP method (see Chapter 2).

Q-value: $Q(i,u)$; the expected total infinite-horizon discounted cost if action u is taken in state i and the current policy is followed in all states thereafter.

R-value: $R(i)$; the expected total infinite-horizon discounted cost if no new action is specified in state i and the current policy is followed in all states thereafter.

Removal Rule: a rule guiding the removal of TPs (see Section 3.3.9).

Stack Updating Procedure: a procedure called by the Practical TPDP Algorithm.

Swapping Rule: a rule guiding the swapping of TPs (see Section 3.3.5).

TP: transition point – the association of an action u_{TP} with a state i.

TP action: u_{TP}; the action associated with a TP.

TP states: S_B; states associated with TPs.

TPDP: transition point dynamic programming.

aborted swap states: S_G; the set of TP states for which a TP swap was attempted but aborted.

absorbing states: S_A; states where system transitions are terminated.

accepted swap states: $S_{G'}$; the set of TP states for which a TP swap was attempted and accepted.

action: (control action) the action a controller specifies that a system should perform.

activity level: $a_t(u)$; the signal level that each driving neuron would output if it was not being mutually inhibited by the other driving neurons.

addition: adding a TP to a closed state space S_C.

allocation cost: $c_{\text{allocation}}$; a cost placed on the allocation and preservation of each TP.

anchored states: TP states which are always reached from the starting states S_S (considered in the proof of Theorem 3.3).

assessment group: the set of TPs being evaluated at any one time as Practical TPDP learning progresses.

back-up time: the time required for accurate cost estimates to be backed-up.

backing-up: the backwards movement of cost estimates that results from updating the cost estimates at each state using evaluation function values of later states.

balancing point: the point at which the output of a driving neuron stops increasing and starts decreasing during a mutual inhibition contest.

biasing lines: lines carrying binary signals, each one of which can increase the activity level of a single driving neuron in order to bias mutual inhibition contest outcomes.

boundary: a set of boundary states associated with the same uniform region.

boundary states: $S_{B^\diamond}$; states where TPs specify an action that will be used throughout a uniform state space region.

closed state space: S_C; the states which can be reached in a valid environment with a given set of TPs.

composite action: an action consisting of a number of elemental action specifications.

context lines: lines carrying binary signals, each one of which indicates if the system is to perform a specific task.

control task: a task that the system is to perform.

controller: a mechanism controlling a system or plant.

delay parameter: σ_{delay}; a random positive parameter added to time step t to determine the update time t_{update}.

delayed updating: the delaying of Q-value, R-value and weight updating until longer term costs can be observed.

direct controllers: controllers which use implicit system models.

discount factor: γ; the attenuation factor used for future immediate costs.

dormant states: $S_{D^\diamond}$; states within a uniform region that are not boundary states – no actions need be specified at them.

driving neuron output: $m_t(u)$; the output level of driving neuron u at time step t.

driving neurons: neurons that each specify a separate action – they *or* together outputs from identification neurons attending to states for which their action is appropriate.

elemental action: actions representing the finest control action resolution possible.

entry action probability: $\rho(i,u)$; the probability that action u led to the arrival of the system at state i (in a valid environment, with the existing set of TPs).

entry actions: $U_e(i)$; all the actions that can result in a transition to state i from some other state (in a valid environment, with the existing set of TPs).

environment: a set of conditions under which TPDP operates (see Section 3.2.2).

epoch: a set of 20 learning and 500 testing trials.

evaluation function: V_μ; the set of evaluation function values $V_\mu(i)$ over the entire state space S.

evaluation function value: $V_\mu(i)$; the expected total infinite-horizon discounted cost if the existing policy is applied from state i onward.

excluded cost: $C_\mu(i,J)$; the expected cost of all state space transitions from state i that can occur with policy μ – *excluding* those after all states j in the arbitrarily chosen set of states J have been encountered.

expected evaluation function value: V_{expected}; an estimate of what the evaluation function value will be at the next TP state.

experienced cost: C_{total}; the total infinite-horizon discounted cost experienced after the system leaves each of the states recorded in the stack.

external exploration: learning exploration intended to discover low cost state transition routes through the external states S_E.

external parameter: σ_{external}; a random parameter used to determine whether or not external exploration is initiated instead of internal exploration.

external states: S_E; states outside the closed state space S_C.

flash reducing: instantaneous reduction of the output of a driving neuron from 1 to its low-grade output level – ends inhibition of other driving neurons by that neuron.

generalization by copying: generalization by attempting actions, during learning exploration, which have been found effective in nearby states.

generalization parameter: $\sigma_{\text{generalize}}$; a random parameter used to determine whether or not generalization by copying should be performed.

gross inverse model: a general system model consisting of knowledge of the actions necessary to move the system to a number of gross positions.

gross position: a loosely defined set of neighboring states.

gross position context lines: context lines that specify that the control task is to reach a gross position.

guided learning: prudent exploration choices made during learning based on higher level control knowledge.

high: an active binary signal.

identification neuron output: $o_t(i)$; the output level of each identification neuron i at time step t.

identification neurons: neurons that inspect binary state lines, each identifying a separate state.

immediate cost: $c(i, u)$; the cost incurred when action u is applied in state i.

indirect controllers: controllers which use explicit system models.

ineffectual TPs: TPs associated with external states S_E.

initiate parameter: $\sigma_{\text{add-TP}}$; a random parameter used to determine whether or not exploration is initiated at non-TP states.

internal exploration: learning exploration intended to discover cost-reducing TPs that should be added to the closed state space S_C.

learning transition: a distinct change in the learning of the optimal policy which occurred during application of Practical TPDP (see Section 5.2.2).

localized operation: the independent operation of neurons in certain types of neural network designs.

low: an inactive binary signal.

low-grade output factor: κ; the parameter used to adjust the low-grade output level.

low-grade output level: the output signal of driving neurons which are active but inhibited by another driving neuron.

memory-based controllers: controllers which use look-up tables that associate actions with states.

minimal TP optimal control: optimal control with TPs when all unnecessary TPs have been removed.

non-TP states: S_D; states *not* associated with TPs.

off-line: calculations which a controller makes when it is not busy controlling the system.

optimal Q-value: $Q^*(i,u)$; the Q-value when the policy is optimal.

optimal TP states: S_{B^*}; states associated with TPs when the policy is optimal.

optimal action: $\mu^*(i)$; the policy action at state i when the policy is optimal.

optimal closed state space: S_{C^*}; closed state space when the policy is optimal.

optimal control: control of a system when that control is optimal with regard to some optimality criteria.

optimal evaluation function: V_{μ^*}; the evaluation function when the policy is optimal.

optimal evaluation function value: $V_{\mu^*}(i)$; the evaluation function value of state i when the policy is optimal.

optimal external states: S_{E^*}; the external states when the policy is optimal.

optimal non-TP states: S_{D^*}; the non-TP states when the policy is optimal.

optimal policy: μ^*; a policy that provides optimal control.

optimality criteria: cost criteria which defines when optimal control has been achieved.

participation factor: $\eta_\mu(i,j)$; a factor which indicates how much of the evaluation function value $V_\mu(i)$ of state i depends on the evaluation function value $V_\mu(j)$ of another state j with policy μ.

performance metric: used during the performance evaluation of Practical TPDP (see Section 5.3.4).

phase plane: a two-dimensional diagram showing the position and velocity of a system.

policy: μ; the set of policy actions $\mu(i)$, over the entire state space, that the controller specifies in each state i.

policy TP: the one TP at each TP state which specifies the policy action.

policy action: $\mu(i)$; the action that the controller specifies in state i (when learning exploration is not occurring).

policy synapse: a policy TP in Neural TPDP.

possible actions: $U(i)$; the set of actions that can be performed in each state i.

removal: removing a TP from a closed state space S_C.

route change parameter: σ_{change}; a random parameter used to determine whether or not a new experimental action is specified during exploration.

starting states: S_S; states in which a system is started.

state lines: lines carrying binary signals, each one of which indicates if the system is in a specific quantized interval of one dimension of the state space.

state space: S; the set of all states that the system can enter.

state transition probability: $p_j(i,u)$; the probability that a transition will occur to state j if action u is applied in state i.

swap parameter: $\sigma_{\text{swap-TP}}$; a random parameter used to determine whether or not internal exploration is initiated at TP states.

swapping: exchanging one TP for another in a closed state space S_C.

synapse: a connection between the output of an identification neuron and the input of a driving neuron – acts as a TP, associating an action with a state.

synapse allocation knowledge: the knowledge each synapse requires as to whether or not other synapses are associated with the same state.

task: (control task) a task that the system is to perform.

time step interval: T; the time between two control time steps.

transition point: (TP) the association of an action u_{TP} with a state i.

trial: a complete set of system state transitions from starting states to absorbing states.

uniform region: a region of neighboring states where the same action is optimal.

update rate: $\alpha_t(i, u)$; the rate at which a Q-value or R-value is updated in Q-learning.

update time: $t_{\mathbf{update}}$; the time step when the next Q-value, R-value and weight update should occur.

value iteration: a successive approximation approach for determining the evaluation function (see Section 2.1.4).

weight: $w(i, u)$; the merit of the TP associating action u with state i.

List of Variables

$\alpha_t(i,u)$	**update rate:** the rate at which a Q-value or R-value is updated in Q-learning.
γ	**discount factor:** the attenuation factor used for future immediate costs.
δ	the low-grade signal level which is sought from an identification neuron.
$\eta_\mu(i,j)$	**participation factor:** a factor which indicates how much of the evaluation function value $V_\mu(i)$ of state i depends on the evaluation function value $V_\mu(j)$ of another state j with policy μ.
κ	**low-grade output factor:** the parameter used to adjust the low-grade output level.
μ	**policy:** the set of policy actions $\mu(i)$, over the entire state space, that the controller specifies in each state i.
μ^*	**optimal policy:** a policy that provides optimal control.
$\mu(i)$	**policy action:** the action that the controller specifies in state i (when learning exploration is not occurring).
$\mu^*(i)$	**optimal action:** the policy action at state i when the policy is optimal.
$\rho(i,u)$	**entry action probability:** the probability that action u led to the arrival of the system at state i (in a valid environment, with the existing set of TPs).

$\sigma_{\text{add-TP}}$	**initiate parameter:** a random parameter used to determine whether or not exploration is initiated at non-TP states.
σ_{change}	**route change parameter:** a random parameter used to determine whether or not a new experimental action is specified during exploration.
σ_{delay}	**delay parameter:** a random positive parameter added to time step t to determine the update time t_{update}.
σ_{external}	**external parameter:** a random parameter used to determine whether or not external exploration is initiated instead of internal exploration.
$\sigma_{\text{generalize}}$	**generalization parameter:** a random parameter used to determine whether or not generalization by copying should be performed.
$\sigma_{\text{swap-TP}}$	**swap parameter:** a random parameter used to determine whether or not internal exploration is initiated at TP states.
$C_{\mu}(i, J)$	**excluded cost:** the expected cost of all state space transitions from state i that can occur with policy μ – *excluding* those after all states j in the arbitrarily chosen set of states J have been encountered.
C_{total}	**experienced cost:** the total infinite-horizon discounted cost experienced after the system leaves each of the states recorded in the stack.
$L(i)$	the set of state lines connected as inputs to identification neuron i.
$Q(i, u)$	**Q-value:** the expected total infinite-horizon discounted cost if action u is taken in state i and the current policy is followed in all states thereafter.
$Q^*(i, u)$	**optimal Q-value:** the Q-value when the policy is optimal.

$R(i)$	**R-value:** the expected total infinite-horizon discounted cost if no new action is specified in state i and the current policy is followed in all states thereafter.
S	**state space:** the set of all states that the system can enter.
S_A	**absorbing states:** states where system transitions are terminated.
S_B	**TP states:** states associated with TPs.
S_{B^*}	**optimal TP states:** states associated with TPs when the policy is optimal.
$S_{B^\diamond}$	**boundary states:** states where TPs specify an action that will be used throughout a uniform state space region.
S_C	**closed state space:** the states which can be reached in a valid environment with a given set of TPs.
S_{C^*}	**optimal closed state space:** closed state space when the policy is optimal.
S_D	**non-TP states:** states *not* associated with TPs.
S_{D^*}	**optimal non-TP states:** the non-TP states when the policy is optimal.
$S_{D^\diamond}$	**dormant states:** states within a uniform region that are not boundary states – no actions need be specified at them.
S_E	**external states:** states outside the closed state space S_C.
S_{E^*}	**optimal external states:** the external states when the policy is optimal.
S_G	**aborted swap states:** the set of TP states for which a TP swap was attempted but aborted.

$S_{G'}$	**accepted swap states:** the set of TP states for which a TP swap was attempted and accepted.
S_S	**starting states:** states in which a system is started.
T	**time step interval:** the time between two control time steps.
$\bar{T}_{\text{stuck}}$	the time period for which no system movement will be permitted before exploration is initiated.
$\bar{T}_{\text{uniform}}$	the average time required to cross each uniform region.
$U(i)$	**possible actions:** the set of actions that can be performed in each state i.
$U_e(i)$	**entry actions:** all the actions that can result in a transition to state i from some other state (in a valid environment, with the existing set of TPs).
V_μ	**evaluation function:** the set of evaluation function values $V_\mu(i)$ over the entire state space S.
V_{μ^*}	**optimal evaluation function:** the evaluation function when the policy is optimal.
$V_\mu(i)$	**evaluation function value:** the expected total infinite-horizon discounted cost if the existing policy is applied from state i onward.
$V_{\mu^*}(i)$	**optimal evaluation function value:** the evaluation function value of state i when the policy is optimal.
V_{expected}	**expected evaluation function value:** an estimate of what the evaluation function value will be at the next TP state.

$X(i,j)$	the set of all possible state transition routes $\vec{x}$ from state i to state j.
$a_t(u)$	**activity level:** the signal level that each driving neuron would output if it was not being mutually inhibited by the other driving neurons.
$c(i,u)$	**immediate cost:** the cost incurred when action u is applied in state i.
$c_{\text{allocation}}$	**allocation cost:** a cost placed on the allocation and preservation of each TP.
d	the number of time steps between TP states.
d_{quick}	the number of time steps required to pass through a single discretized state at the highest system velocity.
i	a particular system state.
j	a particular system state.
k	iteration number.
l	a state line.
$m_t(u)$	**driving neuron output:** the output level of driving neuron u at time step t.
$o_t(i)$	**identification neuron output:** the output level of each identification neuron i at time step t.
$p_j(i,u)$	**state transition probability:** the probability that a transition will occur to state j if action u is applied in state i.
$p_{\text{ineffective}}$	the probability that a specified car acceleration instruction had no effect (see Section 5.1.2).

s_t	the state at time t.
t	**time step:** the current time step.
$t_{\textbf{last-TP}}$	the last time step in which a TP state was encountered.
$t_{\textbf{update}}$	**update time:** the time step when the next Q-value, R-value and weight update should occur.
u	an action taken in a given state.
u_{TP}	**TP action:** the action associated with a TP.
$\tilde{u}$	a hypothetical non-action action.
$w(i,u)$	**weight:** the merit of the TP associating action u with state i.
$w_{\textbf{initial}}$	the initial setting of a TP weight (see Section 4.2.1).
$w_{\textbf{max}}$	the maximum value of a TP weight (see Section 4.2.1).
$w_{\textbf{thr}}$	a threshold TP weight value (see Section 4.2.2).
x_t	the horizontal position of the car at time step t (see Section 5.1.2).
$\dot{x}_t$	the horizontal velocity of the car at time step t (see Section 5.1.2).
$\dot{x}_L$	the horizontal velocity limit of the car (see Section 5.1.2).
$\vec{x}$	a possible state transition route between two states.
y_t	the vertical position of the car at time step t (see Section 5.1.2).
$\dot{y}_t$	the vertical velocity of the car at time step t (see Section 5.1.2).

$\dot{y}_L$	the vertical velocity limit of the car (see Section 5.1.2).
$z_t(l)$	the signal level of state line l at time step t.

Bibliography

Aboaf, E. W., C. G. Atkeson and D. J. Reinkensmeyer (1988), "Task-level robot learning", *Proceedings of the 1988 International Conference on Robotics and Automation*, vol. 2, 1988, pp. 1309-1310.

Albus, J. S. (1975*a*), "A new approach to manipulator control: the cerebellar model articulation controller (CMAC)", *Transactions of the ASME*, vol. 97, Sept. 1975, pp. 220-227.

Albus, J. S. (1975*b*), "Data storage in the cerebellar model articulation controller (CMAC)", *Transactions of the ASME*, vol. 97, Sept. 1975, pp. 228-233.

Albus, J. S. (1983), "A structure for generation and control of intelligent behavior", *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, 1983, pp. 25-28.

Albus, J. S. (1988), "The central nervous system as a low and high level control system", *NATO ASI Series: Sensors and Sensory Systems for Advanced Robots*, vol. F43, ed.: P. Dario, Berlin: Springler-Verlag, 1988, pp. 3-20.

Albus, J. S. (1991), "Outline for a theory of intelligence", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-21, no. 3, May/June 1991, pp. 473-509.

Alkon, D. L. (1989), "Memory storage and neural systems", *Scientific America*, July, 1989, pp. 42-50.

Alkon, D. L., K. T. Blackwell, G. S. Barbour, A. K. Rigler and T. P. Vogl (1990), "Pattern-recognition by an artificial network derived from biologic neuronal systems", *Biological Cybernetics*, vol. 62, 1990, pp. 363-376.

Anderson, C. W. (1989*a*), "Learning to control an inverted pendulum using neural networks", *IEEE Control Systems Magazine*, vol. 9, no. 3, Apr. 1989, pp. 31-37.

Anderson, C. W. (1989*b*), "Towers of Hanoi with connectionist networks: learning new features", *Machine Learning: Proceedings of the 6th International Conference*, San Mateo, California: Morgan Kaufmann Publishers, 1989, pp. 345-350.

Anderson, C. W. (1993), "Q-learning with hidden unit restarting", *Advances in Neural Information Processing Systems 5*, San Mateo, California: Morgan Kaufmann Publishers, 1993, pp. 81-88.

Atkeson, C. G. and D. J. Reinkensmeyer (1988), "Using associative content-addressable memories to control robots", *Proceedings of the 27th Conference on Decision and Control*, Austin, Texas, Dec. 1988, pp. 792-797.

Atkeson, C. G. (1989), "Learning arm kinematics and dynamics", *Annual Review of Neuroscience*, vol. 12, 1989, pp. 157-183.

Atkeson, C. G. (1991), "Using locally weighted regression for robot learning", *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, Sacramento, California, Apr. 1991, pp. 958-963.

Ayestaran, H. E. and R. W. Prager (1993), "The logical gates growing network", *Report CUED/F-INFENG/TR 137*, Cambridge University Engineering Department, Cambridge, July 1993.

Baker, W. L. and J. A. Farrell (1992), "An introduction to connectionist learning control systems", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 35-63.

Barto, A. G., R. S. Sutton and C. W. Anderson (1983), "Neuronlike elements that can solve difficult learning control problems", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 13, no. 5, 1983, pp. 835-846.

Barto, A. G., R. S. Sutton and C. J. C. H. Watkins (1989), "Learning and sequential decision making", *COINS Technical Report 89-95*, University of Massachusetts, Sept. 1989.

Barto, A. G. and S. P. Singh (1990*a*), "Reinforcement learning and dynamic programming", *Proceedings of the 6th Yale Workshop on Adaptive and Learning Systems*, Yale, 1990, pp. 83-88.

Barto, A. G. and S. P. Singh (1990*b*), "On the computational economics of reinforcement learning", *Connectionist Models: Proceedings of the 1990 Summer School*, San Mateo, California: Morgan Kaufmann Publishers, 1990, pp. 35-44.

Barto, A. G., R. S. Sutton and C. J. C. H. Watkins (1990*c*), "Sequential decision problems and neural networks", *Advances in Neural Information Processing Systems 2*, San Mateo, California: Morgan Kaufmann Publishers, 1990, pp. 686-693.

Barto, A. G., S. J. Bradtke and S. P. Singh (1991), "Real-time learning and control using asynchronous dynamic programming", *COINS Technical Report 91-57*, University of Massachusetts, Aug. 1991.

Barto, A. G. (1992), "Reinforcement learning and adaptive critic methods", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 469-491.

Barto, A. G., S. J. Bradtke and S. P. Singh (1993), "Learning to act using real-time dynamic programming", *Department of Computer Science*, University of Massachusetts, Jan. 1993.

Bellman (1957), *Dynamic Programming*, Princeton: Princeton University Press, 1957.

Buckland, K. M. and P. D. Lawrence (1993), "A connectionist approach to direct dynamic programming control", *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, 1993, vol. 1, pp. 284-287.

Chapman, D. and L. P. Kaelbling (1991), "Input generalization in delayed reinforcement-learning: an algorithm and performance comparisons", *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, Aug. 1991, pp. 726-731.

Chinchuan, C., C. Y. Maa and M. A. Shanblatt (1990), "An artificial neural network algorithm for dynamic programming", *International Journal of Neural Systems*, vol. 1, no. 3, 1990, pp. 211-220.

Dayan, P. (1991), "Navigating through temporal difference", *Advances in Neural Information Processing Systems 3*, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 464-470.

Fahlman, S. E. and G. E. Hinton (1987), "Connectionist architectures for intelligence", *Computer*, Jan. 1987, pp. 100-109.

Feldman, J. A. and D. H. Ballard (1982), "Connectionist models and properties", *Cognitive Science*, vol. 6, 1982, pp. 205-254.

Gardner, M. (1973), "Mathematical games", *Scientific American*, 228:108, Jan. 1973.

Girosi, F. and T. Poggio (1989), "Networks and the best approximation property", *Artificial Intelligence Laboratory*, Massachusetts Institute of Technology, A.I. Memo No. 1164, C.B.I.P. Paper No. 45, Oct. 1989.

Hillis, W. D. (1985), *The Connection Machine*, Cambridge, Massachusetts: MIT Press, 1985.

Jaakkola, T., M. I. Jordan and S. P. Singh (1993), "On the convergence of stochastic iterative dynamic programming algorithms", *Advances in Neural Information Processing Systems 5*, San Mateo, California: Morgan Kaufmann Publishers, 1993.

Jacobs, R. A. and M. I. Jordan (1991), "A competitive modular connectionist architecture", *Advances in Neural Information Processing Systems 3*, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 767-773.

Kaelbling, L. P. (1990), *Learning in Embedded Systems*, Ph.D. Thesis, Stanford University, Department of Computer Science, Stanford, California, Tech. Rep. TR-90-04, 1990.

Korf, R. E. (1990), "Real-time heuristic search", *Artificial Intelligence*, vol. 42, 1990, pp. 189-211.

Kraft, L. G. W. T. Miller and D. Dietz (1992), "Development and application of CMAC neural network-based control", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 215-232.

Lin, L. J. (1991*a*), "Programming robots using reinforcement learning and teaching", *Proceedings of the 9th International Conference on Artificial Intelligence*, Cambridge, Massachusetts: MIT Press, 1991, pp. 781-786.

Lin, L. J. (1991*b*), "Self-improvement based on reinforcement learning, planning and teaching", *Machine Learning: Proceedings of the 8th International Workshop*, eds.: L. Birnbaum and G. Collins, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 323-327.

Mahadevan, S. and J. Connell (1992), "Automatic programming of behavior-based robots using reinforcement learning", *Artificial Intelligence*, vol. 55, 1992, pp. 311-365.

Massone, L. and E. Bizzi (1989), "A neural network model for limb trajectory formation", *Biological Cybernetics*, vol. 61, 1989, pp. 417-425.

Mendel, J. M. (1973), "Reinforcement learning models and their applications to control problems", *Learning Systems: A Symposium of the AACC Theory Committee*, 1973, pp. 3-18.

Michie, D. and R. A. Chambers (1968), "BOXES: an experiment in adaptive control", *Machine Intelligence 2*, eds.: E. Dale and D. Michie, Oliver and Boyd, 1968, pp. 137-152.

Minsky, M. L. (1985), *The Society of Mind*, New York: Simon and Schuster, 1985.

Moore, A. W. (1991), "Variable resolution dynamic programming: efficiently learning action maps in multivariate real-valued state-spaces", *Machine Learning: Proceedings of the 8th International Workshop*, eds.: L. Birnbaum and G. Collins, San Mateo, California: Morgan Kaufmann Publishers, 1991.

Moore, A. W. (1992), "Fast, robust adaptive control by learning only forward models", *Advances in Neural Information Processing Systems 4*, San Mateo, California: Morgan Kaufmann Publishers, 1992, pp. 571-578.

Moore, A. W. and C. G. Atkeson (1993), "Prioritized sweeping: reinforcement learning with less data and less real time", *Machine Learning*, Oct. 1993.

Narendra, K. S. and M. A. L. Thathachar (1989), *Learning Automata: An Introduction*, New Jersey: Prentice-Hall, 1989.

Narendra, K. S. (1992), "Adaptive control of dynamical systems using neural networks", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 141-183.

Ogata, K. (1970), *Modern Control Engineering*, Englewood Cliffs, New Jersey: Prentice-Hall, 1970.

Omohundro, S. M. (1987), "Efficient algorithms with neural network behavior", *Complex Systems*, vol. 1, 1987, pp. 273-347.

Peng, J. and R. J. Williams (1992), "Efficient search control in Dyna", College of Computer Science, Northeastern University, Mar. 1992.

Poggio, T. and F. Girosi (1990), "Networks for approximation and learning", *Proceedings of the IEEE*, vol. 78, no. 9, Sept. 1990, pp. 1481-1497.

Ross, S. (1983), *Introduction to Stochastic Dynamic Programming*, New York: Academic Press, 1983.

Rumelhart, D. E. and D. Zipser (1985), "Feature discovery by competitive learning", *Cognitive Science*, vol. 9, 1985, pp. 75-112.

Rumelhart, D. E., G. E. Hinton and R. J. Williams (1986), "Learning internal representations by error propagation", *Parallel Distributed Processing*, eds.: D. E. Rumelhart and J. L. McClelland, Boston: MIT Press, 1986, vol. 1, pp. 318-362.

Samuel, A. L. (1959), "Some studies in machine learning using the game of checkers", *IBM Journal on Research and Development*, 1959, pp. 210-229.

Singh, S. P. (1991*a*), "Transfer of learning across compositions of sequential tasks", *Machine Learning: Proceedings of the 8th International Workshop*, eds.: L. Birnbaum and G. Collins, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 348-352.

Singh, S. P. (1991*b*), "Transfer of learning by composing solutions of elemental sequential tasks", *Department of Computer Science*, University of Massachusetts, Dec. 1991.

Singh, S. P. (1992*a*), "Scaling reinforcement learning algorithms by learning variable temporal resolution models", *Proceedings of the 9th Machine Learning Conference*, eds.: D. Sleeman and P. Edwards, July 1992.

Singh, S. P. (1992*b*), "The efficient learning of multiple task sequences", *Advances in Neural Information Processing Systems 4*, San Mateo, California: Morgan Kaufmann Publishers, 1992, pp. 251-258.

Schmidhuber, J. (1990*a*), "A local learning algorithm for dynamic feedforward and recurrent networks", *Report FKI-124-90*, Institut für Informatik, Technische Universität München, Munich, Germany, 1990.

Schmidhuber, J. (1990*b*), "Learning algorithms for networks with internal and external feedback", *Connectionist Models: Proceedings of the 1990 Summer School*, eds.: D. S. Touretzky *et. al.*, San Mateo, California: Morgan Kaufmann Publishers, 1990, pp. 52-61.

Standish, T. A. (1980), *Data Structure Techniques*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1980.

Sutton, R. S. (1988), "Learning to predict by the methods of temporal differences", *Machine Learning*, vol. 3, 1988, pp. 9-43.

Sutton, R. S. (1990), "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming", *Machine Learning: Proceedings of the 7th International Conference*, eds.: L. Birnbaum and G. Collins, San Mateo, California: Morgan Kaufmann Publishers, 1990, pp. 216-224.

Sutton, R. S. (1991), "Integrated modeling and control based on reinforcement learning and dynamic programming", *Advances in Neural Information Processing Systems 3*, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 471-478.

Sutton, R. S., A. G. Barto and R. J. Williams (1992), "Reinforcement learning is direct adaptive optimal control", *IEEE Control Systems Magazine*, vol. 12, no. 2, Apr. 1992, pp. 19-22.

Tesauro, G. J. (1991), "Practical issues in temporal difference learning", *Research Report RC 17223 (#76307)*, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, 1991.

Tham, C. K. and R. W. Prager (1993), "Reinforcement learning methods for multi-linked manipulator obstacle avoidance and control", *Engineering Department*, Cambridge University, Cambridge, Mar. 1993.

Thrun, S. B. (1992), "The role of exploration in learning control", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 527-559.

Thrun, S. B. and A. Schwartz (1993), "Issues in using function approximation for reinforcement learning", *Proceedings of the 4th Connectionist Models Summer School*, Hillsdale, New Jersey: Lawrence Erlbaum Publisher, Dec. 1993.

Utgoff, P. E. and J. A. Clouse (1991), "Two kinds of training information for evaluation function learning", *Proceedings of the 9th Annual Conference on Artificial Intelligence*, San Mateo, California: Morgan Kaufmann Publishers, 1991, pp. 596-600.

Watkins, C. J. C. H. (1989), *Learning from Delayed Rewards*, Ph.D. Thesis, Cambridge University, Cambridge, England, 1989.

Watkins, C. J. C. H. and P. Dayan (1992), "Q-learning", *Machine Learning*, vol. 8, 1992, pp. 279-292.

Werbos, P. J. (1990), "Consistency of HDP applied to a simple reinforcement learning problem", *Neural Networks*, vol. 3, no. 3, 1990, pp. 179-189.

White, D. A. and M. I. Jordan (1992), "Optimal control: a foundation for intelligent control", *Handbook of Intelligent Control*, eds.: D. A. White and D. A. Sofge, New York: Van Nostrand Reinhold, 1992, pp. 185-214.

Widrow, B., N. K. Gupta and S. Maitra (1973), "Punish/reward: learning with a critic in adaptive threshold systems", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 3, no. 5, 1973, pp. 455-465.

Williams, R. J. (1986), "Reinforcement learning in connectionist networks: a mathematical analysis", *ICS Report 8605*, Institute for Cognitive Science, University of California, San Diego, June 1986.

Williams, R. J. (1987*a*), "A class of gradient-estimating algorithms for reinforcement learning in neural networks", *Proceedings of the IEEE First International Conference on Neural Networks*, San Diego, California, 1987, vol. 2, pp. 601-608.

Williams, R. J. (1987*b*), "Reinforcement-learning connectionist systems", *Technical Report NU-CCS-87-3*, College of Computer Science, Northeastern University, Boston, Feb. 1987.

Yee, R. C. (1992), "Abstraction in control learning", *COINS Technical Report 92-16*, University of Massachusetts, Mar. 1992.

Appendix A

Proof of the Convergence of the TPDP Version of Q-Learning

Proof of the convergence of Q-learning performed using TP updating Equation 3.18 is based on the dynamic programming convergence proofs developed by Jaakkola *et. al.* (1993). Theorem 1 of Jaakkola *et. al.*'s work (1993) is:

Theorem 1: A random iterative process $\Delta_{n+1}(x) = (1 - \alpha_n(x))\Delta_n(x) + \beta_n(x)F_n(x)$ converges to zero w.p.1 under the following assumptions:

1. The state space is finite.
2. $\sum_n \alpha_n(x) = \infty$, $\sum_n \alpha_n^2(x) < \infty$, $\sum_n \beta_n(x) = \infty$, $\sum_n \beta_n^2(x) < \infty$, and $E\{\beta_n(x)|P_n\} \leq E\{\alpha_n(x)|P_n\}$ uniformly w.p.1.
3. $\|E\{F_n(x)|P_n\}\|_W < \gamma\|\Delta_n\|_W$, where $\gamma \in (0,1)$.
4. $\mathrm{Var}\{F_n(x)|P_n\} \leq K(1 + \|\Delta_n\|_W)^2$, where K is some constant.

Here $P_n = \{\Delta_n, \Delta_{n-1}, ..., F_{n-1}, ..., \alpha_{n-1}, ...\beta_{n-1}, ...\}$ stands for the past at step n. $F_n(x), \alpha_n(x)$ and $\beta_n(x)$ are allowed to depend on the past insofar as the above conditions remain valid. The notation $\|\cdot\|_W$ refers to some weighted maximum norm.

The proof of Jaakkola *et. al.*'s Theorem 1 will not be presented (see Jaakkola *et. al.*, 1993), but the convergence of Q-learning performed using TP updating Equation 3.18 will be proven by relating the updating process to the converging stochastic process defined by Theorem 1.

Theorem A:[1] Updating Q-values with the following form of the TP updating Equation 3.18[2]:

$$Q_{t+d}(s_t, u_t) = [1 - \alpha_t(s_t, u_t)]\, Q_t(s_t, u_t) + \alpha_t(s_t, u_t) \left[\left(\sum_{n=0}^{d-1} \gamma^n c(s_{t+n}, u_t) \right) + \gamma^d V_t(s_{t+d}) \right] \tag{A.52}$$

results in convergence to optimal Q-values $Q^*(s, u)$ if:

1. The state and action spaces are finite.
2. $\sum_t \alpha_t(s, u) = \infty$ and $\sum_t \alpha_t^2(s, u) < \infty$ uniformly w.p.1.
3. $\mathrm{Var}\{c(s, u)\}$ is bounded, and $c(s, u) \geq 0 \;\forall\; s, u$.
4. If $\gamma = 1$ all policies lead to a cost free absorbing state w.p.1.

Proof: By subtracting $Q^*(s_t, u_t)$ from both sides of updating Equation A.52 and by defining:

$$\Delta_t(s, u) = Q_t(s, u) - Q^*(s, u)$$

$$F_t(s_t, u_t) = \left(\sum_{n=0}^{d-1} \gamma^n c(s_{t+n}, u_t) \right) + \gamma^d V_t(s_{t+d}) - Q^*(s_t, u_t)$$

updating Equation A.52 can be seen to have the form of the process in Jaakkola *et. al.*'s Theorem 1 with $\beta_t(s, u) = \alpha_t(s, u)$.

Conditions 1 and 2 of Jaakkola *et. al.*'s Theorem 1 are met by conditions 1 and 2 of Theorem A, so it remains to be shown that $F_t(s, u)$ has the properties required by Theorem 1 – those defined by its conditions 3 and 4.

[1]Theorem A closely follows the conventional Q-learning theorem presented by Jaakkola *et. al.* (1993).
[2]This form is Equation 3.18 with a few minor notation changes.

To show that condition 3 of Jaakkola *et. al.*'s Theorem 1 is met, we have:

$$\begin{aligned} E\{F_t(s,u)\} &= C_t(s,S_B) + \sum_{j\in S_B} \eta_t(s,j)V_t(j) - C_t(s,S_B) - \sum_{j\in S_B} \eta_t(s,j)V_{\mu^*}(j) \\ &= \sum_{j\in S_B} \eta_t(s,j)(V_t(j) - V_{\mu^*}(j)) \end{aligned} \tag{A.53}$$

Where $C_t(s, S_B)$ is the excluded cost of state transitions from state s to the TP states S_B (with all costs incurred after the TP states S_B are encountered being excluded from the total – see Section 3.3.4), and $\eta_t(s,j)$ is the participation factor of TP state j in the evaluation function value of state s (see Section 3.3.4).

Manipulating $E\{F_t(s,u)\}$ produces:

$$\begin{aligned} |E\{F_t(s,u)\}| &= |\sum_{j\in S_B} \eta_t(s,j)(V_t(j) - V_{\mu^*}(j))| \\ &\leq \sum_{j\in S_B} \eta_t(s,j)|V_t(j) - V_{\mu^*}(j)| \\ &\leq \sum_{j\in S_B} \eta_t(s,j)|\min_v Q_t(j,v) - \min_v Q^*(j,v)| \\ &\leq \sum_{j\in S_B} \eta_t(s,j)\max_v |Q_t(j,v) - Q^*(j,v)| \end{aligned} \tag{A.54}$$

The maximum value the right hand side of Equation A.54 can have is when a one-step transition is made, w.p.1, to a TP state j that has the largest Q-value error ($\max_v |Q_t(j,v) - Q^*(j,v)|$) of all the TP states S_B. Using weighted maximum norm weights of 1 the following can thus be stated:

$$\|E\{F_t(s,u)\}\|_W \leq \gamma \max_j \max_v |Q_t(j,v) - Q^*(j,v)| \tag{A.55}$$

$$\|E\{F_t(s,u)\}\|_W \leq \gamma \|\Delta_t(j,v)\|_W \tag{A.56}$$

Since the notation of $\|\Delta_t(j,v)\|_W$ can be arbitrarily changed to $\|\Delta_t(s,u)\|_W$, condition 3 of Jaakkola *et. al.*'s Theorem 1 is met.

To show that condition 4 of Jaakkola *et. al.*'s Theorem 1 is met, we have:

$$\begin{aligned}
\mathrm{Var}\{F_t(s,u)\} &= \mathrm{Var}\left\{\left(\sum_{n=0}^{d-1}\gamma^n c(s_n,u)\right)+\gamma^d V_t(s_d)-Q^*(s,u)\middle| s_0=s\right\} \\
&\leq 3\mathrm{Var}\left\{\sum_{n=0}^{d-1}\gamma^n c(s_n,u)\middle| s_0=s\right\}+3\mathrm{Var}\left\{\gamma^d V_t(s_d)\middle| s_0=s\right\} \\
&\leq K_1+3\mathrm{Var}\left\{\gamma^d V_t(s_d)\middle| s_0=s\right\} \qquad \text{(A.57)}
\end{aligned}$$

Where K_1 is a constant reflecting the fact that the variance of the immediate costs incurred is bounded as a result of conditions 3 and 4 of Theorem A. Continuing on (and assuming throughout that $s_0=s$):

$$\mathrm{Var}\{F_t(s,u)\}\leq K_1+3\sum_{j\in S_B}\sum_{\vec{x}\in X(s,j)}\left[\left(\prod_{k=0}^{|\vec{x}|-1}p_{x_{k+1}}(x_k,u)\right)\left(\gamma^{|\vec{x}|}V_t(j)-E\left\{\gamma^d V_t(s_d)\right\}\right)^2\right] \qquad \text{(A.58)}$$

Where $X(s,j)$ is the set of all possible state transition routes $\vec{x}$ from state s to TP state j ($\vec{x}\in X(s,j)$), $\vec{x}=[x_0,x_1,...,x_n]$ is one possible state transition route from state s to TP state j of variable length n ($x_0=s, x_n=j$), and $|\vec{x}|$ is the number of states along each such route.

The maximum value the right hand side of Equation A.58 can have is when a one-step transition is made, w.p.1, to a TP state j that has the largest evaluation function value $V_t(j)$ of all the TP states S_B, and when the expected value of $\gamma^d V_t(s_d)$ is 0. Using weighted maximum norm weights of 1 the following can thus be stated[3]:

$$\begin{aligned}
\mathrm{Var}\{F_t(s,u)\} &\leq K_1+3\left(\gamma\max_j V_t(j)-0\right)^2 \\
\mathrm{Var}\{F_t(s,u)\} &\leq K_1+3\gamma^2\left(\max_j V_t(j)\right)^2 \qquad \text{(A.59)}
\end{aligned}$$

[3]The outcome is the same if $V_t(j)$ is minimized and the expected value of $\gamma^d V_t(s_d)$ is maximized.

By again using weighted maximum norm weights of 1, the bound of condition 4 of Jaakkola *et. al.*'s Theorem 1 can be expressed as follows:

$$\begin{aligned} K(1+\|\Delta_t\|_W)^2 &= K\Big(1+\max_s\max_u|Q_t(s,u)-Q^*(s,u)|\Big)^2 \\ &\geq K\Big(1+\max_s|\min_u Q_t(s,u)-\min_u Q^*(s,u)|\Big)^2 \\ &\geq K\Big(1+\max_s|V_t(s)-V_{\mu^*}(s)|\Big)^2 \end{aligned} \tag{A.60}$$

Since the notation of Equation A.60 can be arbitrarily changed to:

$$K(1+\|\Delta_t\|_W)^2 \;\geq\; K\Big(1+\max_j|V_t(j)-V_{\mu^*}(j)|\Big)^2 \tag{A.61}$$

Equations A.59 and A.61 can be employed to verify that condition 4 of Jaakkola *et. al.*'s Theorem 1 is met:

$$K_1+3\gamma^2\Big(\max_j V_t(j)\Big)^2 \;\overset{?}{\leq}\; K\Big(1+\max_j|V_t(j)-V_{\mu^*}(j)|\Big)^2 \tag{A.62}$$

Equation A.62 is true because conditions 3 and 4 of Theorem A ensure that the maximum value of $V_t(j)$ for any state j is bounded – the left hand side of Equation A.62 is therefore also bounded. The right hand side of Equation A.62 always has a value of at least K, and K can be arbitrarily increased to exceed the left hand side bound. As a result:

$$\begin{aligned} K_1+3\gamma^2\Big(\max_j V_t(j)\Big)^2 &\leq K\Big(1+\max_j|V_t(j)-V_{\mu^*}(j)|\Big)^2 \\ \mathrm{Var}\{F_t(s,u)\} &\leq K(1+\|\Delta_t\|_W)^2 \end{aligned} \tag{A.63}$$

condition 4 of Jaakkola *et. al.*'s Theorem 1 is therefore met. □

Appendix B

Full Description of the Practical TPDP Algorithm

Appendix Section B.2 fully describes the Practical TPDP Algorithm, and Appendix Section B.3 fully describes the Stack Updating Procedure it calls.

B.1 General Operation of the Practical TPDP Algorithm

This Appendix Section repeats part of Section 4.2.4.

The general operation of the Practical TPDP Algorithm is as follows. The Practical TPDP controller can be in one of three exploration modes. The mode in effect at any time is identified by the variable *explore* as 'none', 'internal' or 'external'. When no exploration is occurring actions are randomly chosen from those specified by the TPs at the states encountered. The immediate costs incurred when those actions are taken are observed, and the Q-values, R-values and weights of the TPs specifying those actions are updated accordingly. Internal and external exploration (see Section 4.2.3) are randomly initiated in the midst of this process and are allowed to continue until a TP state is encountered. Internal and external exploration facilitate the allocation of new TPs that can be further assessed.

B.2 The Practical TPDP Algorithm

Considering the Practical TPDP Algorithm presented in Figure B.40 (a conveniently located copy of Figure 4.3):

Lines 1 to 5

These lines initialize the algorithm by chosing a starting state s_0 and a starting action $u_0 \in U(s_0)$ for the trial. They also set a number of algorithm variables to starting values. These include *explore*; t, the time step; t_{update}, an indicator of when the next stack updating should occur; and $t_{\text{last-TP}}$, the last time step in which a TP state was encountered.

The variable *explore* is initialized to 'external' if the starting state s_0 has no TP because an initial TP action must be specified for the exploration mode to be made 'none'. The exploration mode 'external' results in random movement of the system through the state space until a TP state is encountered. This is a reasonable behavior if there is no TP associated with the starting state which specifies some sort of initial action.

Lines 10 to 11, and 60 to 63

These lines determine when the calculations are made for Practical TPDP. Calculations are made whenever the system enters a new state, or after a period T_{stuck} has passed in the same state (Line 11). The latter condition is used to prevent Practical TPDP from permanently remaining in the same state, continually specifying the same action. Line 10 ensures that Practical TPDP calculations continue until the system has reached one of the absorbing states S_A.

Line 61 is used to update the *expected evaluation function value* V_{expected}, a running estimate of what the evaluation function value $V_\mu(i)$ will be at the next TP state i encountered. This estimate is based on the evaluation function value of the last TP state encountered (Line 33) and the immediate costs experienced since then. The update of V_{expected} is done recursively and is based on Equation 2.2.

Line 63 calls the Stack Updating Procedure (Figure B.41) to perform a final updating when an absorbing state has been reached.

Lines 20 to 34

These lines define the three algorithm operations that can be performed when the system encounters a TP state.

The first operation (Lines 22 to 25) is to initiate internal exploration at the current state. This operation is performed when the value of the random parameter $\sigma_{\text{swap-TP}}$ is greater than zero (Line 21). The value of $\sigma_{\text{swap-TP}}$ has some fixed probability of being greater than zero each time it is considered[1].

When internal exploration is initiated the stack is updated using the evaluation function value $V_\mu(s_t)$ ($V_\mu(s_t) = Q(s_t, \mu(s_t))$) for the current state s_t (Line 22, see Appendix Section B.3). Then the *explore* variable is set to indicate the mode of exploration occurring (Line 23), and an experimental action u_t is randomly chosen from the set $U(s_t)$ (Line 24). Finally, the current time step, state and action information is stored on the stack for future updating (Line 25).

The second operation (Lines 27 to 31) is to randomly choose an action from those specified by the TPs at the current state s_t (Line 30). This facilitates evaluation of the TP specifying that action. To prepare for such evaluation the stack is updated (Line 27), and the *explore* variable is set to indicate that no internal or external exploration is occurring (Line 28). The current time step, state and action information is also stored on the stack for future updating (Line 31).

The second operation is always performed if internal or external exploration has been occurring (excluding internal exploration which has just been initiated – in Lines 21 to 25), and it terminates all such exploration (Line 26). The second operation is also

[1]Section 4.2.6 describes how the random distribution of $\sigma_{\text{swap-TP}}$ is determined.

performed if no exploration has been occurring, but the update time t_{update} has been reached (see Section 4.2.5).

The third operation (Line 32) is performed if neither of the other operations is. That is, the third operation is performed only if no exploration is occurring and the update time t_{update} has not yet been reached. This operation involves simply storing the current time step, state and action information in the stack for future updating. When the stack update is performed, the Q-values, R-values and weights will be updated for each state identified in this manner. As will be explained in Appendix Section B.3, this is done because even though no actions are specified in the states concerned, it will still be possible to update some of the Q-values, R-values and weights associated with them.

The *specified* field stored in the stack entry during the third operation is set to 'false' to indicate that no action was specified at the state concerned (Line 32). This field is set to 'true' when the second operation, that of choosing a TP action, is performed (Line 31). This field is set to a null value ($\emptyset$) when the first operation, the initiation of internal exploration, is performed (Line 25). This is because the value of this variable has no meaning when exploration is occurring (see Appendix Section B.3).

Finally, Lines 33 and 34 update variables V_{expected}, $V_{\text{last-TP}}$ and $t_{\text{last-TP}}$ – all of which are based on the last TP state encountered. The expected evaluation function value V_{expected} is a running estimate of what the evaluation function value $V_\mu(i)$ will be at the next TP state i encountered. As explained earlier in this Appendix Section (Lines 10 to 11, and 60 to 63), this estimate is set to the evaluation function value of the last TP state encountered ($V_\mu(s_t) = Q(s_t, \mu(s_t))$), and then recursively updated as immediate costs are experienced (Line 61).

The variable $t_{\text{last-TP}}$ indicates the time step in which the last TP state was encountered. This variable is used if internal or external exploration is initiated between TP states. It facilitates a stack update using the evaluation function value of the last TP state encountered (see Appendix Section B.3).

Lines 40 to 43

These lines are used to make random changes in the route followed through the state space during internal and external exploration. Route changes are made when exploration is already occurring, and the system is making state transitions through non-TP states. The determination of when to make route changes is made based on the random parameter σ_{change}, which has some fixed probability of being greater than zero each time it is considered (Line 40)[2].

When a route change is made a new experimental action is randomly selected (Line 42), and the current time step, state and action information are stored on the stack to record the change (Line 43). If external exploration is occurring, the stack is first flushed so that only the last action specified will be stored on it when a TP state is encountered and a stack update is performed (Line 41). As explained in Section 4.2.3, this ensures that only the last action specified before a TP state is encountered is allocated a TP.

Lines 50 to 55

These lines are used to initiate internal and external exploration when the system is making state transitions through non-TP states. The determination of when to initiate this exploration is made based on the random parameter $\sigma_{\text{add-TP}}$, which has some fixed probability of being greater than zero each time it is considered (Line 50)[3].

When exploration is initiated the stack is updated using the time step $t_{\text{last-TP}}$ in which the last TP state was encountered (Line 51). Then the determination of whether the exploration should be internal or external is made based on the random parameter

[2]Section 4.2.6 describes how the random distribution of σ_{change} is determined.
[3]Section 4.2.6 describes how the random distribution of $\sigma_{\text{add-TP}}$ is determined.

σ_{external}, which has some fixed probability of being greater than zero each time it is considered (Line 52)[4]. Finally, a new action is randomly selected (Line 54) and the current time step, state and action information are stored on the stack to record the change (Line 55).

B.3 The Stack Updating Procedure

The Stack Updating Procedure (see Figure B.41 – a conveniently located copy of Figure 4.4) operates by retrieving state and action combinations from the stack in reverse of the order in which they were experienced and recorded. As a result, updating begins at the last TP state encountered before the stack update was initiated, and the *experienced cost* C_{total} (the total infinite-horizon discounted cost experienced after the system leaves each of the states recorded in the stack) can be recursively calculated starting with the evaluation function value V_{update} of that last TP state (Lines 2 and 13). The recursively calculated C_{total} is used during updating in various ways, depending on the mode of exploration in effect.

Parameters passed

The Stack Updating Procedure is passed four parameters by the Practical TPDP Algorithm. These are *explore*, the mode of exploration occurring when the stack update was requested; the updating time step t, which is initialized to the time step when the last TP state was encountered; V_{update}, the evaluation function value of the last TP state encountered; and V_{expected}, the running estimate of what the evaluation function value $V_{\mu}(i)$ will be at the next TP state i encountered (see Section B.2).

The mode of exploration passed to the Stack Updating Procedure as *explore* is always

[4]Section 4.2.6 describes how the random distribution of σ_{external} is determined.

the same as the mode of exploration that was occurring when each of the stack entries was stored. This is because the stack is updated every time the mode of exploration is changed (see Section B.2 and Figure B.40).

Lines 1 to 2

Line 1 removes all entries from the stack whose time step t_s equals or exceeds the initial value of t – the time step when the last TP state was encountered. This operation will only have an effect if the current state is a non-TP state and stack entries have been recorded since the last TP state was encountered. It removes all such entries because, unless the system is at a TP state, no evaluation function value will be available with which these non-TP states can be updated.

Line 2 initializes the value of C_{total}.

Lines 10 to 14

These lines determine which state should next be updated as entries representing the states are retrieved from the stack. For each recorded state the values t_s, i_s, u_s and *specified*$_s$ are retrieved from the stack (Line 11). The experienced cost is also recursively calculated by discounting C_{total} and adding the immediate cost $c(i_s, u_s)$ to it for each time step (Lines 12 to 14).

Lines 20 to 33

These lines perform the updating of states when no internal or external exploration has been occurring. If, at the state i_s being updated, a TP exists that specifies an action corresponding to the action u_s taken in that state, the Q-value of that TP is updated (Line 23). This is done whether the TP itself specified the action, or whether it had been specified previously at another TP state. In either case the Q-value can legitimately be updated based on the consequences of that action being applied to the system. The Q-value updating is done according to the TP updating Equation 3.18, and the update makes use of C_{total}. Incorporated in C_{total} is both the evaluation function value V_{update} of the last TP state encountered by the system, and the immediate costs experienced as the system moved from i_s to that state. Once the Q-value updating has been performed, the weight of the TP concerned is updated according to the rules presented in Section 4.2.1 (Lines 24 to 29).

Lines 30 to 31 are used to update R-values according to the R-value updating Equation 3.27. If there is only one TP associated with state i_s, and it did not specify an action when state i_s was encountered (as indicated by *specified*$_s$; see Appendix Section B.2, Lines 20 to 34), the R-value of that TP is updated. If there is more than one TP associated with state i_s and one of them specifies the action u_s that was taken in state i_s, the R-values of all of the other TPs are updated.

These two cases facilitate updating that is very effective in producing R-values that can be used to assess the merits of each TP. The former case results in the R-value for a single TP at state i_s reflecting the costs incurred if that TP did not exist. This R-value can then be directly compared to the Q-value of the single TP to assess the value of that TP. The latter case results in the R-value for each TP associated with state i_s reflecting the costs incurred when *other* TPs at state i_s specify an action. The R-value calculated for a TP in this manner can be compared with the Q-value of that TP to assess the value of that TP relative to the other TPs. If this comparison reveals that the Q-value of a TP is higher than its R-value, then it is known that other TPs at state i_s result in lower costs. The TP is removed as a result. The elimination of high Q-value TPs in this manner results in the R-values of the remaining TPs being lowered, which leads to more TP removals. Eventually a single TP will remain, and it will then be assessed solely on the basis of the costs that will be incurred if it is removed (as described previously). This whole process is basically one of competitive learning (Rumelhart *et. al.*, 1985).

Finally, the policy TP for each state i_s identified in the stack must be determined (see Section 4.2.2). Lines 32 to 33 make this determination using Equations 4.40 and 4.41.

Lines 40 to 44

These lines perform the updating of TP states when internal exploration has been occurring. At most one such state, the oldest state in the stack, will be identified during each stack update. This is because all exploration is terminated when a TP state is encountered, and the only TP state stored on the stack during internal exploration will be the one in which the exploration was initiated (see Section B.2 and Figure B.40).

If internal exploration is initiated in a TP state i_s, the merit of the route taken through the state space during that exploration can be determined simply by comparing the experienced cost C_{total} with the evaluation function value $V_\mu(i_s)$ $(V_\mu(i_s) = Q(i_s, \mu(i_s)))$ of that state. If C_{total} is less than $Q(i_s, \mu(i_s))$ (Line 40), then the action u_s specified at the initiation of the internal exploration is worthy of further consideration and a TP is allocated to specify that action at the TP state concerned (Line 41).

The Q-value and R-value of any TPs allocated are initially set to C_{total} (Line 42). This is the best initial estimate of the Q-value. The R-value is also set to this value so that it will diverge away from the initial Q-value to become higher or lower. If the true R-value of a TP is higher than its true Q-value, and it is initialized to some value lower than the initial Q-value, it will have to be updated until it exceeds the Q-value. During the period that it remains lower than the Q-value, the weight of that TP will be fallaciously decreased (the TP may even be removed as a result). In a similar manner, the weight of a TP may be fallaciously increased if its R-value is initialized to some value higher than its initial Q-value. Initializing the Q-values and R-values of each TP with the same value is therefore the best strategy.

When a new TP is allocated its weight is initialized to w_{initial} (Line 43). This weight is also compared to the weight of the policy TP to see if the new TP should be made the policy TP (Line 44). This comparison is a reduced version of the comparison made in Line 33, with the reductions resulting from conditions that are sure to be met when a new TP is being allocated.

Lines 50 to 54

These lines perform the updating of non-TP states when internal or external exploration has been occurring.

If external exploration has been occurring, the stack will only have one entry (see Appendix Section B.2, Lines 40 to 43). For the reasons presented in Section 4.2.3, a new TP is unconditionally allocated at the appropriate state to specify the action indicated by that stack entry.

If internal exploration has been occurring, the merit of the route taken through the state space during that exploration can be determined by making use of the expected evaluation function value V_{expected} (see Figure B.40 and Appendix Section B.2, Lines 10 to 11, and 60 to 63). If the actual evaluation function value of the TP state encountered, V_{update}, is lower than V_{expected} (Line 50), then the route taken through the state space during the internal exploration was a low cost one and the actions specified along that route are worthy of further consideration. In this case new TPs are allocated to specify each such action at the states where they were specified during exploration.

When new TPs are allocated at non-TP states, the allocation operations are the same as those performed when allocating new TPs at states that already have TPs (Lines 41 to 44). The only difference is that, being the only TPs at the states concerned, the new TPs are automatically made the policy TPs (Line 54).

```
1.  randomly choose starting state $s_0 \in S_S$
2.  choose a starting action $u_0 \in U(s_0)$
3.  if (state $s_0$ has a TP): ‘none’ $\Rightarrow$ $explore$
4.  otherwise: ‘external’ $\Rightarrow$ $explore$
5.  $0 \Rightarrow t$, $0 \Rightarrow t_{\text{update}}$, $0 \Rightarrow t_{\text{last-TP}}$
10. while $s_t$ is not an absorbing state $S_A$:
11.     if (state $s_t \neq s_{t-T}$) or (state $s_t = s_{t-T}$ for time $T_{\text{stuck}}$):
20.         if (state $s_t$ has a TP):
21.             if ($\sigma_{\text{swap-TP}} > 0$):
22.                 update-stack($explore$, $t$, $Q(s_t, \mu(s_t))$, $V_{\text{expected}}$)
23.                 ‘internal’ $\Rightarrow$ $explore$
24.                 randomly choose action $u_t \in U(s_t)$
25.                 push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
26.             otherwise if ($explore \neq$ ‘none’) or ($t > t_{\text{update}}$):
27.                 update-stack($explore$, $t$, $Q(s_t, \mu(s_t))$, $V_{\text{expected}}$)
28.                 ‘none’ $\Rightarrow$ $explore$
29.                 $t + \sigma_{\text{delay}} \Rightarrow t_{\text{update}}$
30.                 randomly choose action $u_t$ from TP actions in $U(s_t)$
31.                 push-on-stack($t$, $s_t$, $u_t$, ‘true’)
32.             otherwise: push-on-stack($t$, $s_t$, $u_t$, ‘false’)
33.             $Q(s_t, \mu(s_t)) \Rightarrow V_{\text{expected}}$, $Q(s_t, \mu(s_t)) \Rightarrow V_{\text{last-TP}}$
34.             $t \Rightarrow t_{\text{last-TP}}$
40.         if (state $s_t$ has no TP) and ($explore \neq$ ‘none’) and ($\sigma_{\text{change}} > 0$):
41.             if ($explore =$ ‘external’): flush-stack
42.             randomly choose action $u_t \in U(s_t)$
43.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
50.         if (state $s_t$ has no TP) and ($explore =$ ‘none’) and ($\sigma_{\text{add-TP}} > 0$):
51.             update-stack(‘none’, $t_{\text{last-TP}}$, $V_{\text{last-TP}}$, $\emptyset$)
52.             if ($\sigma_{\text{external}} > 0$): ‘external’ $\Rightarrow$ $explore$
53.             otherwise: ‘internal’ $\Rightarrow$ $explore$
54.             randomly choose action $u_t \in U(s_t)$
55.             push-on-stack($t$, $s_t$, $u_t$, $\emptyset$)
60.     $t + T \Rightarrow t$
61.     $\frac{1}{\gamma}(V_{\text{expected}} - c(s_t, u_t)) \Rightarrow V_{\text{expected}}$
62.     observe system for new state $s_t$
63. update-stack($explore$, $t$, $s_t$, $V_{\text{expected}}$)
```

Figure B.40: The Practical TPDP Algorithm

```
                    [Parameters passed: explore, t, $V_{\text{update}}$, $V_{\text{expected}}$]
1.  while (time at top of stack $t_s \geq t$): pop-off-stack($t_s, i_s, u_s$, specified$_s$)
2.  $V_{\text{update}} \Rightarrow C_{\text{total}}$
10. while (there are entries in stack):
11.     pop-off-stack($t_s, i_s, u_s$, specified$_s$)
12.     while ($t > t_s$):
13.         $\gamma C_{\text{total}} + c(i_s, u_s) \Rightarrow C_{\text{total}}$
14.         $t - T \Rightarrow t$
20.     if (explore = 'none'):
21.         for (each TP action $u \in U(i_s)$ in state $i_s$):
22.             if ($u = u_s$):
23.                 $(1-\alpha)Q(i_s, u_s) + \alpha C_{\text{total}} \Rightarrow Q(i_s, u_s)$
24.                 if ($Q(i_s, u_s) < R(i_s, u_s)$):
25.                     $w(i_s, u_s) + 1 \Rightarrow w(i_s, u_s)$
26.                     if ($w(i_s, u_s) > w_{\text{max}}$): $w_{\text{max}} \Rightarrow w(i_s, u_s)$
27.                 otherwise:
28.                     $w(i_s, u_s) - 1 \Rightarrow w(i_s, u_s)$
29.                     if ($w(i_s, u_s) = 0$): remove the TP
30.             if [(state $i_s$ has only one TP) and (specified$_s$ = 'false')] or
                [($u \neq u_s$) and (another TP at state $i_s$ specifies $u_s$)]:
31.                 $(1-\alpha)R(i_s, u) + \alpha C_{\text{total}} \Rightarrow R(i_s, u)$
32.         for (each TP action $u \in U(i_s)$ in state $i_s$):
33.             if [($w(i_s, \mu(i_s)) < w_{\text{thr}}$) and ($w(i_s, u) > w(i_s, \mu(i_s))$)] or
                [($w(i_s, \mu(i_s)) \geq w_{\text{thr}}$) and ($w(i_s, u) \geq w_{\text{thr}}$) and
                ($Q(i_s, u) < Q(i_s, \mu(i_s))$)]: $u \Rightarrow \mu(i_s)$
40.     if (state $i_s$ has TPs) and (memory can be allocated) and
        (explore = 'internal') and ($C_{\text{total}} < Q(i_s, \mu(i_s))$):
41.         allocate a new TP at state $i_s$ with action $u_s$
42.         $C_{\text{total}} \Rightarrow Q(i_s, u_s)$, $C_{\text{total}} \Rightarrow R(i_s, u_s)$
43.         $w_{\text{initial}} \Rightarrow w(i_s, u_s)$
44.         if ($w(i_s, u_s) > w(i_s, \mu(i_s))$) or $w(i_s, u_s) \geq w_{\text{thr}}$): $u_s \Rightarrow \mu(i_s)$
50.     if (state $i_s$ has no TPs) and (memory can be allocated) and
        [((explore = 'internal') and $V_{\text{update}} < V_{\text{expected}}$)) or (explore = 'external')]:
51.         allocate a new TP at state $i_s$ with action $u_s$
52.         $C_{\text{total}} \Rightarrow Q(i_s, u_s)$, $C_{\text{total}} \Rightarrow R(i_s, u_s)$
53.         $w_{\text{initial}} \Rightarrow w(i_s, u_s)$
54.         $u_s \Rightarrow \mu(i_s)$
```

Figure B.41: The Stack Update Procedure