Learning to Act using Real-Time Dynamic Programming*

Andrew G. Barto *Steven J. Bradtke*
Satinder P. Singh
Department of Computer Science
University of Massachusetts, Amherst MA 01003

March 22, 1993

*The authors thank Rich Yee, Vijay Gullapalli, Brian Pinette, and Jonathan Bachrach for helping to clarify the relationships between heuristic search and control. We thank Rich Sutton, Chris Watkins, Paul Werbos, and Ron Williams for sharing their fundamental insights into this subject through numerous discussions, and we further thank Rich Sutton for first making us aware of Korf's research and for his very thoughtful comments on the manuscript. We are very grateful to Dimitri Bertsekas and Steven Sullivan for independently pointing out an error in an earlier version of this article. Finally, we thank Harry Klopf, whose insight and persistence encouraged our interest in this class of learning problems. This research was supported by grants to A.G. Barto from the National Science Foundation (ECS-8912623 and ECS-9214866) and the Air Force Office of Scientific Research, Bolling AFB (AFOSR-89-0526).

Abstract

Learning methods based on dynamic programming (DP) are receiving increasing attention in artificial intelligence. Researchers have argued that DP provides the appropriate basis for compiling planning results into reactive strategies for real-time control, as well as for learning such strategies when the system being controlled is incompletely known. We introduce an algorithm based on DP, which we call Real-Time DP (RTDP), by which an embedded system can improve its performance with experience. RTDP generalizes Korf's Learning-Real-Time-A* algorithm to problems involving uncertainty. We invoke results from the theory of asynchronous DP to prove that RTDP achieves optimal behavior in several different classes of problems. We also use the theory of asynchronous DP to illuminate aspects of other DP-based reinforcement learning methods such as Watkins' Q-Learning algorithm. A secondary aim of this article is to provide a bridge between AI research on real-time planning and learning and relevant concepts and algorithms from control theory.

1 Introduction

The increasing interest of artificial intelligence (AI) researchers in systems embedded in environments demanding real-time performance is narrowing the gulf between problem solving and control engineering. Similarly, machine learning techniques suited to embedded systems are becoming more comparable to methods for the adaptive control of dynamic systems. A growing number of researchers are investigating learning systems based on dynamic programming (DP) algorithms for solving stochastic optimal control problems, arguing that DP provides the appropriate basis for compiling planning results into reactive strategies for real-time control, as well as for learning such strategies when the system being controlled is incompletely known. Learning algorithms based on DP employ novel means for improving the computational efficiency of conventional DP algorithms. Werbos [86; 88] and Watkins [81] proposed incremental versions of DP as learning algorithms, and Sutton's *Dyna* architecture for learning, planning, and reacting [69; 70] is based on these principles. The key issue addressed by DP-based learning is the tradeoff between short- and long-term performance: how can an agent learn to improve long-term performance when this may require sacrificing short-term performance? DP-based learning algoithms are examples of *reinforcement learning* methods by which autonomous agents can improve skills in environments that do not contain explicit teachers [71].

In this article we introduce a learning algorithm based on DP, which we call Real-Time Dynamic Programming (RTDP) by which an embedded problem-solving system can improve its long-term performance with experience, and we prove results about its behavior in several different types of problems. RTDP is the result of recognizing that Korf's [38] Learning-Real-Time A* (LRTA*) algorithm[1] closely related to a form of DP known as asynchronous DP [10]. This novel observation permits us to generalize the ideas behind LRTA* so that they apply to real-time problem solving tasks involving uncertainty. In particular, we apply the theory of asynchronous DP developed by Bertsekas [10] and Bertsekas and Tsitsiklis [12] to show that RTDP converges to optimal solutions when applied to several types of real-time problem solving tasks involving uncertainty. Whereas the theory of asynchronous DP was motivated by the suitability of asynchronous DP for parallel processing, we adapt this theory to the case of performing DP concurrently with problem solving or control. We also present an extension of RTDP, called *Adaptive* RTDP, applicable when information is lacking about a problem's structure in addition to its solution.

Recognizing that the theory of asynchronous DP is relevant to learning also permits us to provide new insight into Watkins' Q-Learning [81; 82] algorithm, another DP-based learning algorithm which is being explored by AI researchers. We present simulation results comparing the performance of RTDP, Adaptive RTDP, Q-Learning, and a conventional DP

[1] We use the term real-time following this usage by Korf in which it refers to problems in which actions have to be performed under hard time constraints. We do not address details of the scheduling issues that arise in using these algorithms as components of complex real-time systems.

algorithm on several simulated real-time problem solving tasks involving uncertainty.

Another aim of this article is to discuss some of the important issues that arise in using DP-based learning algorithms, with particular attention being devoted to indicating which aspects of their use have formal justification and which do not. In doing this, we attempt to clarify links between AI research on real-time planning and learning and relevant concepts from control theory. We discuss selected concepts from control theory that we believe are most relevant to the efforts in AI to develop autonomous systems cabable of performing in real-time and under uncertainty.

But there remain many issues relevant to using DP-based learning in AI that we do not discuss. For example, we adopt a rather abstract formalism and do not say much about how it might best apply to problems of interest in AI. A formalism this abstract is potentially applicable to a wide variety of specific problems, but it is not easy to specify exactly what subproblems within complex systems can best take advantage of these methods. In accord with Dean and Wellman [23], we regard DP-based reinforcement learning as a *component technology* that addresses some of the issues important for developing sophisticated embedded agents but that by itself does not address all of them.

Because the reader is unlikely to be familiar with all of the contributing lines of research, we provide the necessary background in Section 2, followed in Section 3 by a discussion of the proper relationship between some concepts from AI and control theory. Development of the theoretical material occupies Sections 4 through 9, with an introduction to a class of stochastic optimal control problems occupying Section 4 and an introduction to conventional DP occupying Section 5. There are two major parts to this theoretical development. The first part (Sections 5 and 6) concerns problems for which accurate models are available. Here, we describe RTDP, its convergence properties, and its relationship to LRTA*. The second part (Section 7) concerns the additional complexity present in the case of incomplete information, i.e., when an accurate model of the problem is lacking. Section 8 is a brief disscussion of DP-based learning algorithms that are outside the theoretical scope of this article. In Section 9 we discuss some of the issues that practical implementations of DP-based learning algorithms must address. In Section 10 we use an example problem to illustrate RTDP and other algorithms. We conclude in Section 11 with an appraisal of the significance of our approach and discuss some of the open problems.

2 Background

A major influence on research leading to current DP-based algorithms has been the method Samuel [61; 62] used to modify a heuristic evaluation function for the game of checkers. His method updated board evaluations by comparing an evaluation of the current board position with an evaluation of a board position likely to arise later in the game:

> . . . we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board position of the chain of moves which most probably occur during actual play. (Samuel [61].)

As a result of this process of "backing up" board evaluations, the evaluation function should improve in its ability to evaluate the long-term consequences of moves. In one version of this algorithm, Samuel represented the evaluation function as a weighted sum of numerical features and adjusted the weights based on an error derived from comparing evaluations of current and predicted board positions.

Because of its compatibility with connectionist learning algorithms, this approach was refined and extended by Sutton [67; 68] and used heuristically in a number of single-agent problem-solving tasks (e.g., Barto, Sutton, and Anderson [2], Anderson [1], and Sutton [67]). The algorithm was implemented as a neuron-like connectionist element called the Adaptive Critic Element [2]. Sutton [68] later called these algorithms Temporal Difference (TD) methods and obtained some theoretical results about their convergence. Following the proposals of Klopf [36; 37], Sutton and Barto [72; 73; 74] developed these methods as models of animal learning. Minsky [53; 54] discussed similar ideas in the context of the credit assignment problem for reinforcement learning systems; Hampson [28] independently developed some of these ideas and related them to animal behavior; Christensen and Korf [16] experimented with a Samuel-like method for updating evaluation function coefficients using linear regression; and Holland's [30] bucket-brigade algorithm for assigning credit in his classifier systems is closely related to Samuel's method. Tesauro's recent TD-Gammon [77], a program using a TD method together with a connectionist network to improve performance in playing backgammon, has achieved remarkable success.

Independently of the approaches inspired by Samuel's checkers player, other researchers suggested similar algorithms based on the theory of optimal control, where DP provides important solution methods. As applied to control problems, DP (a term introduced by Bellman [9]) consists of methods for successively approximating optimal evaluation functions and decision rules for both deterministic and stochastic problems. In its most general form, DP applies to optimization problems in which the costs of objects in the search space have a compositional structure that can be exploited to find an object of globally minimum cost without performing exhaustive search. Kumar and Kanal [40] discuss DP at this level of generality. However, we restrict attention to DP as it applies to problems in which the objects are state sequences that can be generated in problem-solving or control tasks. DP solves these optimization problems by solving recurrence relations instead of explicitly searching in the space of state sequences. Backing up state evaluations is the basic step of DP procedures for solving these recurrence relations. We discuss several DP algorithms in detail in Section 5.

Although DP algorithms avoid exhaustive search in the state-sequence space, they are still exhaustive by AI standards because they require repeated generation and expansion of all possible states. For this reason, DP has not played a significant role in AI. Heuristic search algorithms, in contrast, are explicitly designed to avoid being exhaustive in this way. But DP algorithms are relevant to learning in a way that heuristic search algorithms are not because they systematically update the evaluations of the states; in effect, they adjust a problem's heuristic evaluation function by incorporating the results of repeated shallow searches. Although some heuristic search algorithms, such as A* [29], update estimates of the costs to reach states from an initial state (A*'s g function), they typically do not update the heuristic evaluation function estimating the cost to reach a goal from each state (the h function).[2]

Despite the fact that DP algorithms are exhaustive in the sense described above, it is possible to arrange their computational steps for use *during* control or real-time problem solving. This is the basis of RTDP and the other algorithms we describe in this article. In most cases, convergence to an optimal evaluation function still requires repeated generation and expansion of all states, but performance improves incrementally (although not necessarily monotonically) while this is being accomplished. It is this improvement rather than ultimate convergence to optimality that becomes central. This perspective was taken by Werbos [84], who proposed a method similar to that used by the Adaptive Critic Element within the framework of DP. He called this approach *Heuristic Dynamic Programming* and has written extensively about it (e.g., refs. [85; 86; 87; 88]). Related algorithms have been discussed by Witten [92; 93], and more recently, Watkins [81] extended Sutton's TD algorithms and developed others by explicitly utilizing the theory of DP. He used the term *Incremental Dynamic Programming* to refer to this class of algorithms and discussed many examples. Williams and Baird [91] theoretically analysed additional DP-based algorithms suitable for on-time application. We have also come across the work Jalali and Ferguson [32], who independently proposed a method similar to Adaptive RTDP. Sutton, Barto, and Williams [75] discussed reinforcement learning from the perspective of DP and adaptive control, and White and Jordan [89] and Barto [5] provide additional background and extensive references to current research.

Although aspects of this approach also apply to problems involving continuous time and/or state and action spaces, here we restrict attention to discrete-time problems with finite sets of states and actions because of their relative simplicity and their closer relationship to the non-numeric problems usually studied in AI. This excludes various "differential" approaches, which make use of optimization algorithms related to the connectionist error-backpropagation algorithm (e.g., Jacobson and Mayne [31], Jordan and Jacobs [33], Werbos [83; 88], White and Jordan [89]).

The relevance of DP for planning and learning in AI was articulated in Sutton's [69] *Dyna* architecture. The key idea in *Dyna* is that one can perform the computational steps of a DP algorithm sometimes using information obtained from state transitions actually taken

[2]We have found only a few exceptions to this in the heuristic search literature in algorithms proposed by Mérõ [51] and Gelperin [26]. Although these algorithms use DP-like backups to update heuristic evaluation functions, they were developed independently of DP.

by the system being controlled, and sometimes from hypothetical state transitions simulated using a model of this system. To satisfy time constraints, this approach interleaves phases of acting with planning performed using hypothetical state transitions. The underlying DP algorithm compiles the resulting information into an efficient form for directing the future course of action. Another aspect of *Dyna* is that the system model can be refined through a learning process deriving training information from the state transitions observed during control. Even without this on-line model refinement, however, executing a DP algorithm concurrently with the generation of actions has implications for planning in AI, as discussed by Sutton in ref. [70].

In this article, we introduce the fact that the theory of asynchronous DP is applicable to the analysis of DP-based reinforcement learning algorithms. Asynchronous DP algorithms differ from conventional DP algorithms in that they do not have to proceed in systematic exhaustive sweeps of the problem's state set. Bertsekas [10] and Bertsekas and Tsitsiklis [12] proved general theorems about the convergence of asynchronous DP applied to discrete-time stochastic optimal control problems. However, because they were motivated by the suitability of asynchronous DP for parallel processing, they did not relate these results to real-time variants of DP as we do in this article. To the best of our knowledge, the only other work in which explicit use is made of the theory of asynchronous DP for real-time control is that of Jalali and Ferguson [32].

Korf's [38] LRTA* algorithm is a heuristic search algorithm that caches state evaluations so that search performance improves with repeated trials. Evaluations of the states visited by the problem solver are maintained in a hash table. Each cycle of the algorithm proceeds by expanding the current state by generating all of its immediate successor states and evaluating them using previously stored evaluations if they exist in the hash table, and otherwise using an initially given heuristic evaluation function. Assuming the objective is to find a minimum-cost path to a goal state, a score is computed for each neighboring state by adding to its evaluation the cost of the edge to it from the current state. The minimum of the resulting scores becomes the new evaluation for the current state, which is stored in the hash table.[3] Finally, a move is made to this lowest-scoring neighboring state. LRTA* therefore backs up state evaluations in much the same way as do Samuel's algorithm and DP. In fact, as we shall see in what follows, with a slight caveat, *LRTA* is the deterministic specialization of asynchronous DP applied on-line.*

[3]In Korf's [38] related Real-Time A* (RTA*) algorithm, the *second* smallest score is stored. Because LRTA* is more closely related to control and DP than is RTA*, we do not discuss RTA*.

3 Heuristic Search and the Control of Dynamic Systems

Whereas AI has focused on problems having relatively little mathematical structure, control theorists have studied more restrictive classes of problems but have developed correspondingly more detailed theories. Some concepts and methods from control theory are nevertheless relevant to problems of interest in AI as discussed, for example, by Dean and Wellman [23]. In this section, as a prelude to introducing the stochastic optimal control framework in which our results are cast, we discuss the relationship between heuristic search, real-time heuristic search, and selected concepts from control theory

3.1 Heuristic Search and System Control

Heuristic search algorithms apply to state-space search problems defined by a set of states, a set of operators that map states to states, an initial state, and a set of goal states. The objective is to find a sequence of operators that maps the initial state to one of the goal states and (possibly) optimizes some measure of cost, or merit, of the solution path. These components constitute a model of some real problem, such as solving a puzzle, proving a theorem, or planning a robot path. The term control as used in the literature on heuristic search and problem solving means the process of deciding what to do next in manipulating a model of the problem in question. Despite some similarities, this is not the meaning of the term control in control theory, where it refers to the process of manipulating the behavior of a physical system in real-time by supplying it with appropriate input signals. In AI, control specifies the formal search process, whereas in control theory, it steers the behavior of a physical system over time. Unlike models manipulated by search algorithms, physical systems cannot be set immediately into arbitrary states and do not suspend activity to await the controller's decisions. Models used to formalize system control problems, called *dynamic systems*, are explicit in taking into account the passage of time. In what follows, by control we mean the control of dynamic systems, not the control of search.

In many applications, a symbolic representation of a sequence of operators is not the final objective of a heuristic search algorithm. The intent may be to execute the operator sequence to generate a time sequence of actual inputs to a physical system. Here the result is the control engineer's form of control, but this control method differs substantially from the methods addressed by most of control theory. A sequence of inputs, or actions, produced in this way through heuristic search is an *open-loop control policy*, meaning that it is applied to the system without using information about the system's actual behavior while control is underway, i.e., without execution monitoring, or feedback. In terms of control theory, heuristic search is a *control design procedure* for producing an open-loop control policy from a system model; the policy is appropriate for the given initial state. Further, under normal circumstances, it is an *off-line* design procedure because it is completed before being used to control the system, i.e, under normal circumstances, the planning phase of the problem-solving process strictly precedes the execution phase.

Open-loop control works fine when all of the following are true: 1) the model used to determine the control policy is a completely accurate model of the physical system, 2) the physical system's initial state can be exactly determined, 3) the physical system is deterministic, and 4) there are no unmodeled disturbances. These conditions hold for some of the problems studied in AI, but they are not true for most realistic control problems. Any uncertainty, either in the behavior of the physical system itself or in the process of modeling the system, implies that *closed-loop control* can produce better performance. Control is closed-loop when each action depends on current observations of the real system, perhaps together with past observations and other information internal to the controller.

A *closed-loop control policy* (also called a closed-loop control rule, law, or strategy) is a rule specifying each action as a function of current, and possibly past, information about the behavior of the controlled system. It closely corresponds to a "universal plan" [64] as discussed, for example, by Chapman [14], Ginsberg [27], and Schoppers [65]. In control theory, a closed-loop control policy usually specifies each action as a function of the controlled system's current state, not just the current values of observable variables (a distinction whose significance for universal planning is discussed by Chapman [14]). Although closed-loop control is closely associated with negative feedback, which counteracts deviations from desired system behavior, negative feedback control is merely a special case of closed-loop control.

When there is no uncertainty, closed-loop control is not in principle more competent than open-loop control. For a deterministic system with no disturbances, given any closed-loop policy and an initial state, there exists an open-loop policy that produces exactly the same system behavior, namely, the open-loop policy generated by running the system, or simulating it with a perfect model, under control of the given closed-loop policy. But this is not true in the stochastic case, or when there are unmodeled disturbances, because the outcome of random and unmodeled events cannot be anticipated in designing an open-loop policy. Note that game-playing systems always use closed-loop control for this reason: the opponent is a kind of disturbance. A game player always uses the opponent's actual previous moves in determining its next move. For exactly the same reasons, closed-loop control can be better than open-loop control for single-agent problems involving uncertainty. A corollary of this explains the almost universal use of closed-loop control by control engineers: the system model used for designing an acceptable control policy can be significantly less faithful to the actual system when it is used for designing closed-loop instead of open-loop policies. Open-loop control only becomes a practical alternative when it is expensive or impossible to monitor the controlled system's behavior with detail sufficient for closed-loop control.

Most control theory addresses the problem of designing adequate closed-loop policies off-line under the assumption that an accurate model of the system to be controlled is available. The off-line design procedure typically yields a computationally efficient method for determining each action as a function of the observed system state. If it is possible to design a complete closed-loop policy off-line, as it is in many of the control problems studied by engineers, then it is not necessary to perform any additional re-design, i.e., re-planning, for problem instances differing only in initial state. Changing control objectives, on the other hand, often does require policy re-design.

One can also design closed-loop policies on-line through *repeated* on-line design of open-loop policies. This approach has been called *receding horizon control* [42; 50]. For each current state, an open-loop policy is designed with the current state playing the role of the initial state. The design procedure must terminate within the time constraints imposed by on-line operation. This can be done by designing a finite-horizon open-loop policy, for example, by using a model for searching to a fixed depth from the current state. After applying the first action specified by the resulting policy, the remainder of the policy is discarded, and the design process is repeated for the next observed state. Despite requiring on-line design, which in AI corresponds to on-line planning through projection, or prediction, using a system model, receding horizon control produces a control policy that is reactive to each current system state, i.e., a closed-loop policy. According to this view, then, a closed-loop policy can involve explicit planning through projection, but each planning phase has to complete in a fixed amount of time to retain the system's reactivity to the observed system states. In contrast to methods that design closed-loop policies off-line, receding horizon control can react on-line to changes in control objectives.

3.2 Optimal Control

The most familiar control objective is to control a system so that its output matches a reference output or tracks a reference trajectory as closely as possible in the face of disturbances. These are called regulation and tracking problems respectively. In an optimal control problem, on the other hand, the control objective is to extremize some function of the controlled system's behavior, where this function need not be defined in terms of a reference output or trajectory. One typical optimal control problem requires controlling a system to go from an initial state to a goal state via a minimum-cost trajectory. In contrast to tracking problems—where the desired trajectory is part of the problem specification—the trajectory is part of the solution of this optimal control problem. Therefore, optimal control problems such as this are closely related to the problems to which heuristic search algorithms apply.

Specialized solution methods exist for optimal control problems involving linear systems and quadratic cost functions, and methods based on the calculus of variations can yield closed-form solutions for restricted classes of problems. Numerical methods applicable to problems involving nonlinear systems and/or nonquadratic costs include gradient methods as well as DP. Whereas gradient methods for optimal control are closely related to some of the gradient descent methods being studied by connectionists (such as the error-backpropagation algorithm [43; 85; 89]), DP methods are more closely related to heuristic search. Like a heuristic search algorithm, DP is an off-line procedure for designing an optimal control policy. However, unlike a heuristic search algorithm, DP produces an optimal closed-loop policy instead of an open-loop policy for a given initial state.

3.3 Real-Time Heuristic Search

Algorithms for real-time heuristic search as defined by Korf [38] apply to state-space search problems in which the underlying model is extended to account for the passage of time. The model thus becomes a dynamic system. Real-time heuristic search algorithms apply to state-space search problems with the additional properties that 1) at each time there is a unique current state of the system being controlled, which is known by the searcher/controller, 2) during each of a sequence of time intervals of constant bounded duration, the searcher/controller must commit to a unique action, i.e., choice of operator, and 3) the system changes state at the end of each time interval in a manner depending on its current state and the searcher/controller's most recent action. These factors imply that there is a fixed upper bound on the amount of time the searcher/controller can take in deciding what action to make if that action is to be based on the most up-to-date state information. Thus, whereas a traditional heuristic search algorithm is a *design* procedure for an open-loop policy, a real-time heuristic search algorithm is a *control* procedure, and it can accommodate the possibility of closed-loop control.

Korf's [38] LRTA* algorithm is a kind of receding horizon control because it is an on-line method for designing a closed-loop policy. However, unlike receding horizon control as studied by control engineers, LRTA* *accumulates* the results of each local design procedure so that the effectiveness of the resulting closed-loop policy tends to improve over time. It stores information from the shallow searches forward from each current state by updating the evaluation function by which control decisions are made. Because these updates are the basic steps of DP, we view LRTA* as the result of interleaving the steps of DP with the actual process of control so that control policy design occurs concurrently with control.

This approach is advantageous when the control problem is so large and unstructured mathematically that complete control design is not even feasible off-line. This case requires a *partial* closed-loop policy, that is, a policy useful for a subregion of the problem's state space. Designing a partial policy on-line allows actual experience to influence the subregion of the state space where design effort is concentrated. Design effort is not expended for parts of the state space that are not likely to be visited during actual control. Although *in general* it is not possible to design a policy that is optimal for a subset of the states unless the design procedure considers the entire state set, this is possible under certain conditions such as those required by Korf's convergence theorem for LRTA*.

3.4 Adaptive Control

Control theorists use the term adaptive control for cases in which an accurate model of the system to be controlled is not available for designing a policy off-line. These are sometimes called control problems with incomplete information. Adaptive control algorithms design policies on-line based on information about the control problem that accumulates over time as the controller and system interact. A distinction is sometimes made between adaptive control and learning control, where only the latter takes advantage of *repetitive* control experiences from which information is acquired that is useful over the long term. Although this distinction may be useful for some types of control problems, we think its utility is limited when applied to the kinds of problems and algorithms we consider in this article. According to what we mean by adaptive control in this article, even though algorithms like LRTA* and Samuel's algorithm [61] are learning algorithms, they are *not* adaptive control algorithms because they assume the existence of an accurate model of the problem being solved. Although it certainly seems odd to us that a control algorithm that learns is not ipso facto adaptive, this is forced upon us when we adopt the control engineer's restricted definition of adaptive control. In Section 7 we describe several algorithms that have properties of both learning and adaptive control algorithms.

4 Markovian Decision Problems

The basis for our theoretical framework is a class of stochastic optimal control problems called *Markovian decision problems*. This is the simplest class of problems that is general enough to include stochastic versions of the problems to which heuristic search algorithms apply, while at the same time allowing us to borrow from a well-developed control and operations research literature. Frameworks that include stochastic problems are important due to the uncertainty present in applications and the fact that it is the presence of uncertainty that gives closed-loop, or reactive, control advantages over open-loop control. In a Markovian decision problem, operators take the form of actions, i.e., inputs to a dynamic system, that probabilistically determine successor states. Although an action is a "primitive" in the theory, it is important to understand that in applications an action can be a high-level command that executes one of a repertoire of complex behaviors. Many problems of practical importance have been formulated as Markovian decision problems, and extensive treatment of the theory and application of this framework can be found in many books, such as those by Bertsekas [11] and Ross [60].

A Markovian decision problem is defined in terms of a discrete-time stochastic dynamic system with finite state set $S = \{1, \ldots, n\}$. Time is represented by a sequence of time steps $t = 0, 1, \ldots$. In Section 6 introducing RTDP, we treat this as a sequence of specific instants of real time, but until then it is best to treat it merely as an abstract sequence. At each time step, a controller observes the system's current state and selects a control action, or simply an *action*,[4] which is executed by being applied as input to the system. If i is the observed state, then the action is selected from a finite set $U(i)$ of admissible actions. When the controller executes action $u \in U(i)$, the system's state at the next time step will be j with state-transition probability $p_{ij}(u)$. We further assume that the application of action u in state i incurs an *immediate cost* $c_i(u)$.[5] When necessary, we refer to states, actions, and immediate costs by the time steps at which they occur by using s_t, u_t, and c_t to denote, respectively, the state, action, and immediate cost at time step t, where $u_t \in U(s_t)$ and $c_t = c_{s_t}(u_t)$. We do not discuss a significant extension of this formalism in which the controller cannot observe the current state with complete certainty. Although this possibility has been studied extensively and is important in practice, the complexities it introduces are beyond the scope of this article.

A closed-loop policy specifies each action as a function of the observed state. Such a policy is denoted $\mu = [\mu(1), \ldots, \mu(n)]$, where the controller executes action $\mu(i) \in U(i)$ whenever it observes state i. This is a *stationary* policy because it does not change over time. Throughout this article, when we use the term policy, we always mean a stationary policy. For any policy μ, there is a function, f^μ, called the *evaluation function*, or the *cost function*, corresponding to policy μ. It assigns to each state the total cost expected to accumulate over time when the controller uses the policy μ starting from the given state. Here, for any policy μ and state i, we define $f^\mu(i)$ to be the expected value of the *infinite-horizon discounted cost* that will accrue over time given that the controller uses policy μ and i is the initial state:

$$f^\mu(i) = E_\mu \left[\sum_{t=0}^{\infty} \gamma^t c_t | s_0 = i \right], \tag{1}$$

where γ, $0 \leq \gamma \leq 1$, is a factor used to discount future immediate costs, and E_μ is the expectation assuming the controller always uses policy μ. We refer to $f^\mu(i)$ simply as the *cost* of state i under policy μ. Thus, whereas the *immediate cost* of state i under policy μ is $c_i(\mu(i))$, the *cost* of state i under policy μ is the expected discounted sum of all the immediate costs that will be incurred over the future starting from state i. Theorists study Markovian decision problems with other types of evaluation functions, such as the function giving average cost per-time-step, but we do not consider those formulations here.

The objective of the type of Markovian decision problem we consider is to find a policy that minimizes the cost of each state i as defined by Equation 1. A policy that achieves this objective is an *optimal policy* which, although it depends on γ and is not always unique, we denote $\mu^* = [\mu^*(1), \ldots, \mu^*(n)]$. To each optimal policy corresponds the same evaluation function, which is the *optimal evaluation function*, or *optimal cost function*, denoted f^*; that

[4] In control theory, this is simply called a *control*. We use the term action because it is the term commonly used in AI.

[5] To be more general, we can alternatively regard the immediate costs as (bounded) random numbers depending on states and actions. In this case, if $c_i(u)$ denotes the *expected* immediate cost of the application of action u in state i, the theory discussed below remains unchanged.

is, if μ^* is any optimal policy, then $f^{\mu^*} = f^*$. For each state i, $f^*(i)$, the *optimal cost* of state i, is the least possible cost for state i for any policy.

This infinite-horizon discounted version of a Markovian decision problem is the simplest mathematically because discounting ensures that the costs of all states are finite for any policy and, further, that there is always an optimal policy that is stationary.[6] The discount factor, γ, determines how strongly expected future costs should influence current control decisions. When $\gamma = 0$, the cost of any state is just the immediate cost of the transition from that state. This is because $0^0 = 1$ in Equation 1 so that $f^\mu(i) = E_\mu[c_0|s_0 = i] = c_i(\mu(i))$. In this case, an optimal policy simply selects actions to minimize the immediate cost for each state, and the optimal evaluation function just gives these minimum immediate costs. As γ increases toward one, future costs become more significant in determining optimal actions, and solution methods generally require more computation.

When $\gamma = 1$, the undiscounted case, the cost of a state given by Equation 1 need not be finite, and additional assumptions are required to produce well-defined decision problems. We consider one set of assumptions for the undiscounted case because the resulting decision problems are closely related to problems to which heuristic search is applied. In these problems, which Bertsekas and Tsitsiklis [12] call *stochastic shortest path problems* (thinking of immediate costs as arc lengths in a graph whose nodes correspond to states), there is an absorbing set of states, i.e., a set of states that once entered is never left, and the immediate cost associated with applying an action to any of the states in the absorbing set is zero. These assumptions imply that the infinite-horizon evaluation function for any policy taking the system into the absorbing set assigns finite costs to every state even when $\gamma = 1$. This is true because all but a finite number of the immediate costs incurred by such a policy over time must be zero. Additionally, as in the discounted case, there is always at least one optimal policy that is stationary. The absorbing set of states corresponds to the set of goal states in a deterministic shortest path problem, and we call it the *goal set*. However, unlike tasks typically solved via heuristic search, here the objective is to find an optimal closed-loop policy, not just an optimal path from a given initial state.

AI researchers studying reinforcement learning often focus on shortest path problems in which all the immediate costs are zero until a goal state is reached, when a "reward" is delivered to the controller and a new trial begins. These are special kinds of the stochastic shortest path problems that address the issue of *delayed reinforcement* [67] in a particularly stark form. Rewards correspond to negative costs in the formalism we are using. In the discounted case when all the rewards are of the same magnitude, an optimal policy produces a shortest path to a rewarding state. Another example of a stochastic shortest path problem receiving attention is identical to this one except that all the non-rewarding immediate costs have the same positive value instead of zero. In this case, an optimal policy produces a

[6] In finite horizon problems, optimal policies are generally nonstationary because different actions can be optimal for a given state depending on how many actions remain until the horizon is reached.

shortest path to a goal state in the undiscounted case. Such problems are examples of minimum-time optimal control problems.

4.1 An Example: The Race Track Problem

To illustrate the Markovian decision framework, we formalize a game called Race Track described by Martin Gardner [25] that simulates automobile racing. We modify the game, which we use in Section 10 to compare the performance of various DP-based learning algorithms, by considering only a single car and by making it probabilistic.

A race track of any shape is drawn on graph paper, with a starting line at one end and a finish line at the other consisting of designated squares. Each square within the boundary of the track is a possible location of the car. Figure 1 shows two example tracks. At the start of each of a sequence of trials, the car is placed on the starting line at a random position, and moves are made in which the car attempts to move down the track toward the finish line. Acceleration and deceleration are simulated as follows. If in the previous move the car moved h squares horizontally and v squares vertically, then the present move can be h' squares vertically and v' squares horizontally, where the difference between h' and h is -1, 0, or 1, and the difference between v' and v is -1, 0, or 1. This means that the car can maintain its speed in either dimension, or it can slow down or speed up in either dimension by one square per move. If the car hits the track boundary,[7] we move it back to a random position on the starting line, reduce its velocity to zero (i.e., $h' - h$ and $v' - v$ are considered to be zero), and continue the trial. The objective is to learn to control the car so that it crosses the finish line in as few moves as possible. Figures 2 and 4 show examples of optimal and near-optimal paths for the race tracks shown in Figure 1.

In addition to the difficulty of discovering faster ways to reach the finish line, it is very easy for the car to gather too much speed to negotiate the track's curves. To make matters worse, we introduce a random factor into the problem. With a probability p, the actual accelerations or decelerations at a move are zero independently of the intended accelerations or decelerations. Thus, $1 - p$ is the probability that the controller's intended actions are executed. One might think of this as simulating driving on a track that is unpredictably slippery so that sometimes braking and throttling up have no effect on the car's velocity.

Although the Race Track problem is suggestive of robot motion and navigation problems, it is not our intention to formulate it in a way best suited to the design of an autonomous vehicle with realistic sensory and motor capabilities. Instead, we regard it as a representative example of problems requiring learning difficult skills, and we formulate the entire task as an abstract stochastic shortest path problem. The first step in this formulation is to define the dynamic system being controlled. The state of the system at each time step $t = 0, 1, \ldots$ can be represented as a quadruple of integers $s_t = (x_t, y_t, \dot{x}_t, \dot{y}_t)$. The first two integers are

[7] For the computational experiments described in Section 10, this means that the projected path of the car for a move intersects the track boundary at any place not on the finish line.

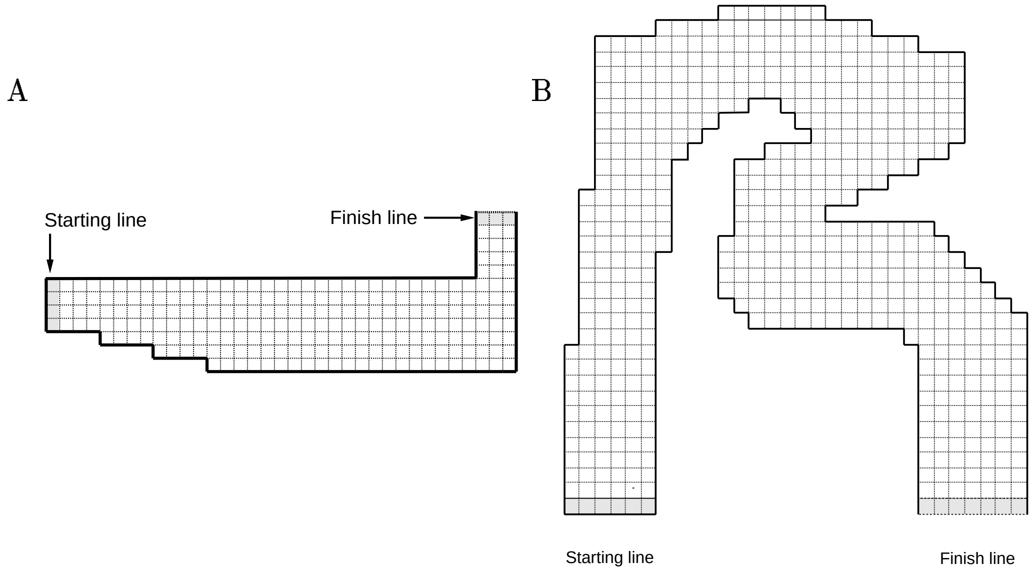


Figure 1: Example Race Tracks. Panel A: Small race track. Panel B: Larger race track. See Table 1 for details.

the horizontal and vertical coordinates of the car's location, and the second two integers are its speeds in the horizontal and vertical directions. That is, $\dot{x}_t = x_t - x_{t-1}$ is the horizontal speed of the car at time step t; similarly $\dot{y}_t = y_t - y_{t-1}$ (we assume $x_{-1} = y_{-1} = 0$). The set of admissible actions for each state is the set of pairs (u^x, u^y), where u^x and u^y are both in the set $\{-1, 0, 1\}$. We let $u_t = (u^x_t, u^y_t)$ denote the action at time t. A closed-loop policy μ assigns an admissible action to each state: the action at time step t is $\mu(s_t) \in \{(-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1)\}$.

The following equations define the state transitions of this system. With probability $1-p$, the controller's action is reliably executed so that the state at time step $t+1$ is

$$\begin{aligned} x_{t+1} &= x_t + \dot{x}_t + u^x_t \\ y_{t+1} &= y_t + \dot{y}_t + u^y_t \\ \dot{x}_{t+1} &= \dot{x}_t + u^x_t \\ \dot{y}_{t+1} &= \dot{y}_t + u^y_t, \end{aligned} \tag{2}$$

and with probability p, the system ignores the controller's action, so that the state at time step $t+1$ is

$$\begin{aligned} x_{t+1} &= x_t + \dot{x}_t \\ y_{t+1} &= y_t + \dot{y}_t \end{aligned}$$

$$\begin{aligned} \dot{x}_{t+1} &= \dot{x}_t \\ \dot{y}_{t+1} &= \dot{y}_t. \end{aligned} \tag{3}$$

This assumes that the straight line joining the point (x_t, y_t) to the point (x_{t+1}, y_{t+1}) lies entirely within the track, or intersects only the finish line. If this is not the case, then the car has collided with the track's boundary, and the state at $t+1$ is $(x, y, 0, 0)$, where (x, y) is a randomly chosen position on the starting line. A move that takes the car across the finish line is treated as a valid move, but we assume that the car subsequently stays in the resulting state until a new trial begins. This method for keeping the car on the track, together with Equations 3 and 2, define the state-transition probabilities for all states and admissible actions.

To complete the formulation of the stochastic shortest path problem, we need to define the set of start states, the set of goal states, and the immediate costs associated with each action in each state. The set of start states consists of all the zero-velocity states on the starting line, i.e., all the states $(x, y, 0, 0)$ where (x, y) are coordinates of the squares making up the starting line. The set of goal states consists of all states that can be reached in one time step by crossing the finish line from inside the track. According to the state-transition function defined above, this set is absorbing. The immediate cost for all non-goal states is one independently of the action taken, i.e., $c_i(u) = 1$ for all non-goal states i and all admissible actions u. The immediate cost associated with a transition from any goal state is zero. If we restrict attention to policies that are guaranteed to take the car across the finish line, we do not need to use discounting. For such a policy μ, the undiscounted infinite-horizon cost, $f^\mu(i)$, of a state i under μ is the expected number of moves for the car to cross the finish line from state i when it is being controlled by a policy μ. An optimal policy, which minimizes this cost of each state, is therefore a policy by which the car is expected to cross the finish line as quickly as possible starting from any state. The optimal cost of a state i, $f^*(i)$, is the smallest expected number of moves to the finish line.

The total number of states depends on the configuration of the race track, but because we have not imposed a limit on the car's speed, it is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the stochastic shortest path problem.

4.2 The Optimality Equation

To set the stage for discussing DP, we provide more detail about the relationship between policies and evaluation functions. Although the evaluation function f^μ gives the cost of each state under policy μ, μ does not necessarily select actions that lead to the best successor states as evaluated by f^μ. In other words, μ is not necessarily a greedy policy with respect to its own evaluation function.

To define a greedy policy in this stochastic case we use Watkins' [81] "Q" notation, which plays a role in the Q-Learning method described in Section 7.3. Let f be a real-valued function of the states; it may be the evaluation function for some policy, a guess for a good evaluation function (such as a heuristic evaluation function in heuristic search), or an arbitrary function. For each state i and action $u \in U(i)$, let

$$Q^f(i,u) = c_i(u) + \gamma \sum_{j \in S} p_{ij}(u) f(j). \tag{4}$$

$Q^f(i,u)$ is the cost of action u in state i as evaluated by f. It is the sum of the immediate cost and the discounted expected value of the costs of the possible successor states under action u. If the system's state transitions are deterministic, then Equation 4 simplifies to

$$Q^f(i,u) = c_i(u) + \gamma f(j),$$

where j is the successor of state i under action u (i.e, node j is the child of node i along the edge corresponding to operator u). In the deterministic case, one can therefore think of $Q^f(i,u)$ as a summary of the result of a one-ply lookahead from node i along the edge corresponding to operator u as evaluated by f. The stochastic case requires a generalization of this view because many edges correspond to each operator, each having a different probability of being followed. If f is the evaluation function for some policy, $Q^f(i,u)$ gives the cost of generating action u in state i and thereafter following this policy.

Using these "Q-values," a policy μ is greedy with respect to f if for all states i, $\mu(i)$ is an action satisfying

$$Q^f(i,\mu(i)) = \min_{u \in U(i)} Q^f(i,u).$$

Although there can be more than one greedy policy with respect to f if more than one action minimizes the set of Q-values for some state, we let μ^f denote any policy that is greedy with respect to f. Also note that any policy is greedy with respect to many different evaluation functions.

A key fact underlying all DP methods is that *the only policies that are greedy with respect to their own evaluation functions are optimal policies.* That is, if μ^* is any optimal policy, then its evaluation function is the optimal evaluation function f^*, and $\mu^* = \mu^{f^*}$. This means that for any state i, $\mu^*(i)$ satisfies

$$Q^{f^*}(i,\mu^*(i)) = \min_{u \in U(i)} Q^{f^*}(i,u). \tag{5}$$

Furthermore, any policy that is greedy with respect to f^* is an optimal policy. Thus, if f^* is known, it is possible to define an optimal policy simply by defining it satisfy Equation 5, i.e., defining it to be greedy with respect to f^*. Due to the definition of Q-values (Equation 4), this generalizes to the stochastic case the fact that an optimal policy is any policy that is best-first with respect to f^* as determined by a one-ply search from each current state. Deeper search is never necessary because f^* already summarizes all the information that such a search would obtain.

Letting $Q^*(i,u) = Q^{f^*}(i,u)$ to simplify notation, a related key fact is that a necessary and sufficient condition for f^* to be the optimal evaluation function is that for each state i it must be true that

$$\begin{aligned} f^*(i) &= \min_{u \in U(i)} Q^*(i,u) \\ &= \min_{u \in U(i)} \left[c_i(u) + \gamma \sum_{j \in S} p_{ij}(u) f^*(j) \right]. \end{aligned} \tag{6}$$

This is one form of the *Bellman Optimality Equation* which can be solved for each $f^*(i)$, $i \in S$, by a DP algorithm. It is a set of n (the number of states) simultaneous nonlinear equations. The form of the equations depends on the dynamic system and the immediate costs underlying the decision problem.

Once f^* has been found, an optimal action for state i can be determined as follows. The Q-values $Q^*(i,u)$ for all admissible actions $u \in U(i)$ are determined via Equation 4. In general, this takes $O(mn)$ computational steps, where n is the number of states and m is the number of admissible actions for state i. However, if one knows which of the state-transition probabilities from state i are zero (as one usually does in the deterministic case), then the amount of computation can be much less ($O(m)$ in the deterministic case). Computing these Q-values amounts to a one-ply lookahead search from state i, which requires knowledge of the system's state-transition probabilities. Using these Q-values, an optimal action can be determined via Equation 5, which takes $m - 1$ comparisons. The computational complexity of finding an optimal action using this method is therefore dominated by the complexity of finding f^*, i.e., by the complexity of the DP algorithm.

5 Dynamic Programming

Given a complete and accurate model of a Markovian decision problem in the form of knowledge of the state-transition probabilities, $p_{ij}(u)$, and the immediate costs, $c_i(u)$, for all states i and actions $u \in U(i)$, it is possible—at least in principle—to solve the decision problem off-line by applying one of various well-known DP algorithms. We describe several versions of a basic DP algorithm called *value iteration*. There are other DP algorithms, including one called *policy iteration*, but learning algorithms based on them are beyond the scope of this article, although we briefly discuss policy iteration in Section 8. We treat DP as referring only to value iteration unless otherwise noted. As used for solving Markovian decision problems, value iteration is a successive approximation procedure that converges to the optimal evaluation function, f^*. It is a successive approximation method for solving the Bellman Optimality Equation whose basic operation is "backing up" estimates of the optimal state costs. There are several variations of value interation depending on how the computations are organized. We first describe the version that applies the backup operations synchronously.

5.1 Synchronous Dynamic Programming

Let f_k denote the estimate of f^* available at stage k of the DP computation, where $k = 0, 1, \ldots$. At stage k, $f_k(i)$ is the estimated optimal cost of state i, which we refer to simply as the *stage-k cost* of state i; similarly, we refer to f_k as the *stage-k evaluation function*, even though it may not actually be the evaluation function for any policy. (We use the index k for the stages of a DP computation, whereas we use t to denote the time step of the control problem being solved.) In synchronous DP, for $k = 0, 1, \ldots$, f_{k+1} is defined in terms of f_k as follows: for each state i,

$$\begin{aligned} f_{k+1}(i) &= \min_{u \in U(i)} \left[c_i(u) + \gamma \sum_{j \in S} p_{ij}(u) f_k(j) \right] \\ &= \min_{u \in U(i)} Q^{f_k}(i, u), \end{aligned} \tag{7}$$

where f_0 is some given initial estimate of f^*. We refer to the application of this update equation for state i as *backing up i's cost*. Although backing up costs is a common operation in a variety of search algorithms in AI, there it does not always mean that the backed-up cost is *saved* for future use. Here, however, the backed-up cost is always saved by updating the evaluation function.

The iteration defined by Equation 7 is synchronous because no values of f_{k+1} appear on the right-hand side of the equation. If one imagines having a separate processor associated with each state, applying Equation 7 for all states i means that each processor backs up the cost of its state at the same time, using the old costs of the other states supplied by the other processors. This process updates all values of f_k simultaneously. Alternatively, a sequential implementation of this iteration requires temporary storage locations so that all the stage-$(k+1)$ costs are computed based on the stage-k costs. The sequential ordering of the backups is irrelevant to the result.

If there are n states and m is the largest number of admissible actions for any state, then each iteration, which consists of backing up the cost of each state exactly once, requires at most $O(mn^2)$ operations in the stochastic case and $O(mn)$ operations in the deterministic case. For the large state sets typical in AI and in many control problems, it is not desirable to try to complete even one iteration, let alone repeat the process until it converges to f^*. For example, because backgammon has about 10^{20} states, a single iteration of value iteration in this case would take more than 1,000 years using a 1,000 MIPS processor.

If $\gamma < 1$, repeated synchronous iterations produce a sequence of functions that converges to the optimal evaluation function, f^*, for any initial estimate, f_0. Although the cost of a state need not get closer to its optimal cost on each iteration, the *maximum* error between $f_k(i)$ and $f^*(i)$ over all states i must decrease (e.g., ref. [11]).

Synchronous DP, as well as the other off-line versions of value iteration we discuss below, generates a sequence of functions that converges to f^* if $\gamma < 1$, *but it does not explicitly generate a sequence of policies.* To each stage-k evaluation function there corresponds at least one greedy policy, but these policies are never explicitly formed. Ideally, one would wait until the sequence converges to f^* and then form a greedy policy corresponding to f^*, which would be an optimal policy. But this is not possible in practice because value iteration converges asymptotically. Instead, one executes value iteration until it meets a test for approximate convergence and then forms a policy from the resulting evaluation function.[8]

It is important to note that a function in the sequence of evaluation functions generated by value iteration does not have to closely approximate f^* in order for a corresponding greedy policy to be an optimal policy. Indeed, a policy corresponding to the stage-k evaluation function for some k may be optimal long before the algorithm converges to f^*. *But unaided by other computations, value iteration does not detect when this first happens.* This fact is an important reason that the on-line variants of value iteration we discuss in this article can have advantages over the off-line variants. Because the controller always uses a policy defined by the current evaluation function, it can perform optimally before the evaluation function converges to the optimal evaluation function.

Bertsekas [11] and Bertsekas and Tsitsiklis [12] give conditions ensuring convergence of synchronous DP for stochastic shortest path problems in the undiscounted case ($\gamma = 1$). Using their terminology, a policy is *proper* if its use implies a nonzero probability of eventually reaching the goal set starting from any state. Using a proper policy also implies that the goal set will be reached eventually from any state with probability one. The existence of a proper policy is the generalization to the stochastic case of the existence of a path from any initial state to the goal set.

Synchronous DP converges to f^* in undiscounted stochastic shortest path problems under the following conditions:

1. the initial cost of every goal state is zero,
2. there is at least one proper policy, and
3. all policies that are not proper incur infinite cost for at least one state.

The third condition ensures that every optimal policy is proper, i.e, it rules out the possibility that a least-cost path exists that never reaches the goal set. One condition under which

[8] Policy iteration, in contrast, explicitly generates a sequence of policies that converges to an optimal policy after a finite number of iterations (when there are a finite number of states and admissible actions, as we are assuming here). However, policy iteration has other shortcomings which we discuss in Section 8.

this is true is when all immediate costs for transitions from non-goal states are positive, i.e, $c_i(u) > 0$ for all non-goal states i and actions $u \in U(i)$.[9] In the deterministic case, conditions 2 and 3 are satisfied if there is at least one solution path from every state and the sum of the immediate costs in every loop is positive.

5.2 Gauss-Seidel Dynamic Programming

Gauss-Seidel DP differs from the synchronous version in that the costs are backed up one state at a time in a sequential "sweep" of all the states, with the computation for each state using the most recent costs of the other states. If we assume that the states are numbered in order, as we have here, and that each sweep proceeds in this order, then the result of each iteration of Gauss-Seidel DP can be written as follows: for each state i and each $k = 0, 1, \ldots,$

$$\begin{aligned} f_{k+1}(i) &= \min_{u \in U(i)} \left[c_i(u) + \gamma \sum_{j \in S} p_{ij}(u) f(j) \right] \\ &= \min_{u \in U(i)} Q^f(i, u). \end{aligned} \tag{8}$$

where

$$f(j) = \begin{cases} f_{k+1}(j) & \text{if } j < i \\ f_k(j) & \text{otherwise.} \end{cases}$$

Unlike synchronous DP, the order in which the states' costs are backed up influences the computation. Nevertheless, Gauss-Seidel DP converges to f^* under the same conditions under which synchronous DP converges. When $\gamma < 1$, repeated Gauss-Seidel sweeps produce a sequence of functions that converges to f^*. For undiscounted stochastic shortest path problems, the conditions described above that ensure convergence of synchronous DP also ensure convergence of Gauss-Seidel DP [12]. Because each cost backup uses the latest costs of the other states, Gauss-Seidel DP generally converges faster than synchronous DP. Furthermore, it should be clear that some state orderings produce faster convergence than others, depending on the problem. For example, in shortest path problems, sweeping from goal states backwards along likely shortest paths usually leads to faster convergence than sweeping in the forward direction.

Although Gauss-Seidel DP is not one of the algorithms of direct interest in this article, we used it to solve the example problem described in Section 4.1 and it serves as a bridge between synchronous DP and the asynchronous form discussed next.

[9]The assumption of positive immediate costs can be weakened to nonnegativity, i.e., $c_i(u) \geq 0$ for all $i \in S$ and $u \in U(i)$, if there exists at least one *optimal* proper policy [12].

5.3 Asynchronous Dynamic Programming

Asynchronous DP is similar to Gauss-Seidel DP in that it does not back up state costs simultaneously. However, it is not organized in terms of systematic successive sweeps of the state set. As proposed by Bertsekas [10] and further developed by Bertsekas and Tsitsiklis [12], asynchronous DP is suitable for multi-processor systems with communication time delays and without a common clock. For each state $i \in S$ there is a separate processor dedicated to backing up the cost of state i (more generally, each processor may be responsible for a number of states). The times at which each processor backs up the cost of its state can be different for each processor. To back up the cost of its state, each processor uses the costs for other states that are available to it when it "awakens" to perform a backup. Multi-processor implementations have obvious utility in speeding up DP and thus have practical significance for all the algorithms we discuss below (see, e.g., Lemmon [44]). However, our interest in asynchronous DP lies in the fact that it does not require state costs to be backed up in any systematically organized fashion.

Although in the full asynchronous model, the notion of discrete computational stages does not apply because a processor can awaken at any of a continuum of times, we use a notion of an iteration stage because it facilitates our discussion of RTDP in the next section. As in the other forms of DP, let f_k denote the estimate of f^* available at stage k of the computation. At each stage k, the costs of a *subset* of the states are backed up synchronously, and the costs remain unchanged for the other states. The subset of states whose costs are backed up changes from stage to stage, and the choice of these subsets determines the precise nature of the algorithm. For each $k = 0, 1, \ldots$, if $S_k \subseteq S$ is the set of states whose costs are backed up at stage k, then f_{k+1} is computed as follows:

$$f_{k+1}(i) = \begin{cases} \min_{u \in U(i)} Q^{f_k}(i,u) & \text{if } i \in S_k \\ f_k(i) & \text{otherwise.} \end{cases} \tag{9}$$

According to this algorithm, then, f_{k+1} may differ from f_k on one state, on many states, or possibly none, depending on S_k. Further, unlike Gauss-Seidel DP, the costs of some states may be backed up several times before the costs of others are backed up once. Asynchronous DP includes the synchronous and Gauss-Seidel algorithms as special cases: synchronous DP results if $S_k = S$ for each k; Gauss-Seidel DP results when each S_k consists of a single state and the collection of S_ks is defined to implement successive sweeps of the entire state set (e.g., $S_0 = \{1\}$, $S_1 = \{2\}$, $\ldots$, $S_{n-1} = \{n\}$, $S_n = \{1\}$, $S_{n+1} = \{2\}$, $\ldots$).

Discounted asynchronous DP converges to f^* provided that the cost of each state is backed up infinitely often, i.e., provided that each state is contained in an infinite number of the subsets S_k, $k = 0, 1, \ldots$. In practice, this means that the strategy for selecting states for cost backups should never eliminate any state from possible selection in the future. In the undiscounted case ($\gamma = 1$), additional assumptions are necessary to ensure convergence. It follows from a result by Bertsekas and Tsitsiklis [12] (p. 446) that asynchronous DP converges in undiscounted stochastic shortest path problems if the cost of each state is backed up infinitely often and the conditions given in Section 5.1 for convergence of synchronous DP are met: 1) the initial cost of every goal state is zero, 2) there is at least one proper policy, and 3) all policies that are not proper incur infinite cost for at least one state.

It is important to realize that a single backup of a state's cost in asnychronous DP *does not necessarily improve it as an estimate of the state's optimal cost*; it may in fact make it worse. However, under the appropriate conditions, the cost of each state converges to its optimal cost with repeated backups. Further, as in Gauss-Seidel DP, the order in which states' costs are backed up can influence the rate of convergence in a problem-dependent way. This fact underlies the utility of various strategies for "teaching" DP-based learning algorithms by supplying experience dictating selected orderings of the backups (e.g., Lin [48], Utgoff and Clouse [80], and Whitehead [90]).

6 Dynamic Programming in Real Time

The DP algorithms described above are off-line algorithms for solving Markovian decision problems. Although they successively approximate the optimal evaluation function through a sequence of stages, these stages are not related to the time steps of the decision problem being solved. Here we consider algorithms in which the controller performs asynchronous DP *concurrently* with the actual process of control, i.e., concurrently with the process of executing actions. The concurrent DP and control processes interact as follows: 1) control decisions are based on the most up-to-date information from the DP computation, and 2) the state sequences generated during control influence the selection of states to which the DP backup operation is applied and whose estimated costs have to be stored. The asynchronous version of DP is appropriate for this role due to the flexibility with which its stages can be defined. As a consequence of this interaction, the controller automatically uses intermediate results of the DP computation to guide its behavior: the DP computation can *focus* on regions of the state set that are most relevant for control as revealed in the system's behavior. The algorithm we call Real-Time DP (RTDP) results when this interaction has specific characteristics that we present below.

Throughout this section we assume that there is a complete and accurate model of the decision problem, the case Sutton [70] discusses in relation to planning in AI. In Section 7, we discuss the adaptive case, in which a complete and accurate model of the decision problem is not available. When there is a model of the decision problem, then the concurrent execution of DP and control can also be carried out in *simulation mode*, where the model is used as a surrogate for the actual system underlying the decision problem. The result is a novel off-line DP computation that can have computational advantages over conventional off-line DP due to its ability to focus on relevant parts of the state set. Despite its not being a real-time computation, we regard the concurrent execution of DP and control in simulation mode to be a form of learning. This is in fact how learning was accomplished in the game-playing programs of Samuel [61; 62] and Tesauro [77]. Learning occurred during many simulated games in which these learning systems competed against themselves. Although we emphasize the real-time use of DP-based learning algorithms, the reader should be aware that our discussion also applies to the use of these algorithms in simulation mode.

To describe the concurrent execution of DP and control, we think of the time steps $t = 0, 1, \ldots$ of the abstract discrete-time formulation of a Markovian decision problem as the indices of a sequence of instants of real time at which the controller must execute control actions. Let s_t be the last state observed before time t, and let k_t be the total number of asynchronous DP stages completed up to time t. Then f_{k_t} is the latest estimate of the optimal evaluation function available when the controller must select action $u_t \in U(s_t)$. When the controller executes u_t, it incurs the immediate cost $c_{s_t}(u_t)$, and the system's state changes to s_{t+1}. By the time the next action, u_{t+1}, has to be selected, some additional stages of asynchronous DP stages are completed to yield $f_{k_{t+1}}$. We let B_t denote the set of states whose costs are backed up in these stages. Note that some states in B_t might have their costs backed up more than once in these stages.

6.1 Real-Time DP

RTDP refers to cases in which the concurrently executing DP and control processes influence one another as follows. First, the controller always follows a policy that is greedy with respect to the most recent estimate of f^*. This means that u_t is always the greedy action with respect to f_{k_t}. Moreover, any ties in selecting these actions must be resolved randomly, or in some other way that ensures the continuing selection of all the greedy actions. Second, between the execution of u_t and u_{t+1}, the cost of s_t is always backed up, i.e., $s_t \in B_t$ for all t. In the simplest case, $B_t = \{s_t\}$ for all t, i.e., the cost of *only* s_t is backed up at each time step t, but more generally, B_t can contain *any* states (in addition to s_t) such as those generated by any type of lookahead search. For example, B_t might consist of the states generated by an exhaustive search from s_t forward to some fixed search depth, or it might consist of the states generated by a search that is best-first according to f_{k_t}.

We say that RTDP converges when the associated asynchronous DP computation converges to f^*. Because the controller always takes actions that are greedy with respect to the current estimate of f^*, when RTDP converges, optimal control performance is attained.[10] The conditions described in Section 5.3 ensuring that asynchronous DP converges to f^* still apply when it is executed concurrently with control. Consequently, in the discounted case, the only condition required for convergence of RTDP is that no state is ever completely ruled out for having its cost backed up. Because RTDP always backs up the cost of the current

[10]When there is more than one optimal policy, the controller will continue to switch between optimal policies because RTDP continues to select among all the greedy actions. This results in a *nonstationary* optimal policy because different optimal actions can be taken from the same state on different occasions.

state, one way to achieve this is to make sure that the controller always continues to visit each state. There are several approaches to ensuring this. One approach is to assume, as is often done in the engineering literature, that the Markov process resulting from the use of any policy is ergodic. This means that there is a nonzero probablility of visiting any state no matter what actions are executed. Discounted RTDP converges under this assumption. However, this assumption is unsatisfactory for stochastic shortest path problems because it does not allow proper subsets of states to be absorbing; it is satisfied only in the trivial stochastic shortest path problem in which every state is a goal state.

A second way to ensure that each state is visited infinitely often is to use multiple *trials*. A trial consists of a time interval of nonzero bounded duration during which RTDP is performed. After this interval, the system is set to a new starting state, and a new trial begins.[11] Obviously, this method cannot be used when it is impossible to to set the system state to selected start states, but for many problems this approach is possible, and it is always possible when RTDP is used in simulation mode.

6.2 Trial-Based RTDP

If the initial states of trials are selected so that every state will be selected infinitely often in an infinite series of trials, then obviously every state will be visited infinitely often—if only at the start of an infinite number of trials. A simple way to accomplish this is to start each trial with a randomly selected state, where each state has a nonzero probablility of being selected. By *Trial-Based* RTDP we mean RTDP used with trials initiated so that every state will, with probability one, be a start state infinitely often in an infinite series of trials. Then the following theorem is an immediate result of noting that in the discounted case, Trial-Based RTDP gives rise to a convergent asynchronous DP computation with probability one for any method of terminating trials:

Theorem 1 *For any discounted Markov decision problem (as defined in Section 4) and any initial evaluation function, Trial-Based RTDP converges (with probability one).*

It is natural to use Trial-Based RTDP in undiscounted stochastic shortest path problems, where trials terminate when a goal state is first reached, or after a predetermined number of time steps. Because Trial-Based RTDP gives rise to a convergent asynchronous DP computation with probability one in an undiscounted stochastic shortest path problems under the conditions enumerated in Section 5.3, we have the following result:

Theorem 2 *In undiscounted stochastic shortest path problems, Trial-Based RTDP converges (with probability one) under the following conditions: 1) the initial cost of every goal state*

[11] RTDP must be interrupted at the end of a trial so that the cost of the last state in a trial is not influenced by the cost of the starting state of the next trial. This prevents the state transitions caused by the "trainer" from influencing the evaluation function.

is zero, 2) there is at least one proper policy, and 3) all policies that are not proper incur infinite cost for at least one state.

Trial-Based RTDP is more interesting if we relax the requirement that it should yield a *complete* optimal evaluation function and a *complete* optimal policy. Consider a trial-based approach to solving undiscounted stochastic shortest path problems in which there is a designated subset of *start states* from which trials always start. We say that a state i is *relevant* if a start state s and an optimal policy exist such that i can be reached from state s when the controller uses that policy. It suffices to find a policy that is optimal when restricted to relevant states because the other states (*irrelevant states*) will never occur during the use of that (or any other) optimal policy. If one somehow knew which states were relevant, then one could apply DP to just these states and possibly save a considerable amount of time and space. But clearly this is not possible because knowing which states are relevant requires knowledge of optimal policies, which is what one is seeking.

However, under certain conditions, without continuing to back up the costs of irrelevant states, Trial-Based RTDP converges to a function that equals f^* on all relevant states, and the controller's policy converges to a policy that is optimal on all relevant states. The costs of some irrelevant states may not have to be backed up at all. Moreover, if memory for the estimated state costs is allocated incrementally during trials, the exhaustive memory requirement of conventional DP can be avoided because Trial-Based RTDP tends to focus computation onto the set of relevant states, and eventually restricts computation to this set. Conditions under which this is possible are stated precisely in the following theorem, whose proof is given in Appendix A:

Theorem 3 *In undiscounted stochastic shortest path problems, Trial-Based RTDP, with the initial state of each trial restricted to a set of start states, converges (with probability one) to f^* on the set of relevant states, and the controller's policy converges to an optimal policy (possibly nonstationary) on the set of relevant states, under the following conditions: 1) the initial cost of every goal state is zero, 2) there is at least one proper policy,*[12] *3) all immediate costs incurred by transitions from non-goal states are positive, i.e., $c_i(u) > 0$ for all non-goal states i and actions $u \in U(i)$, and 4) the initial costs of all states are non-overestimating, i.e., $f_0(i) \leq f^*(i)$ for all states $i \in S$.*

Condition 4 can be satisfied by simply setting $f_0(i) = 0$ for all i. The significance of Theorem 3 is that it gives conditions under which a policy that is optimal on the relevant states can be achieved without continuing to devote computational effort to backing up the costs of irrelevant states. Under these conditions, RTDP can yield an optimal policy when

[12]If trials are allowed to time out before a goal state is reached, it is possible to eliminate the requirement that at least one proper policy exists. Timing out prevents getting stuck in fruitless cycles, and the time-out period can be extended systematically to ensure that it becomes long enough to let all the optimal paths be followed without interruption.

state and action sets are too large to feasibly apply conventional DP algorithms, although the amount of computation saved will clearly depend on characteristics of the problem being solved such as its branching structure. Moreover, if RTDP is applied on-line instead of in simulation mode, whenever the evaluation function changes so that its greedy policy shows improvement, the controller automatically takes advantage of this improvement. This can occur before the evaluation function is close to f^*.

Although in both discounted and undiscounted problems, the eventual convergence of RTDP does not depend critically on the choice of states whose costs are backed between the execution of actions (except that the cost of the current state must be backed up), judicious selection of these states can accelerate convergence. Sophisticated exploration strategies can be implemented by selecting these states based on prior knowledge and on the information contained in the current evaluation function. For example, in a trial-based approach to a stochastic shortest path problem, guided exploration can reduce the expected trial duration by helping the controller find goal states. It also makes sense for RTDP to back up the costs of states whose current costs are not yet accurate estimates of their optimal costs but whose successor states do have accurate current costs. Techniques for "teaching" DP-based learning systems by suggesting certain back ups over others (refs. [46; 90; 80]) rely on the fact that the order in which the costs of states are backed up can influence the rate of convergence of asynchronous DP, whether applied off- or on-line. A promising approach recently developed by Peng and Williams [58] and Moore and Atkeson [57], which the latter authors call "prioritized sweeping," directs the application of DP backups to the most likely predecessors of states whose costs change significantly. Exploration such as this—whose objective is to facilitate finding an optimal policy when there is a complete model of the decision problem—must be distinguished from exploration designed to facilitate learning a model of the decision problem when one is not available. We discuss this latter objective for exploration in Section 7.

6.3 RTDP and LRTA*

Theorem 3 is a generalization of Korf's [38] convergence theorem for LRTA*. RTDP extends LRTA* in two ways: it generalizes LRTA* to stochastic problems, and it includes the option of backing up the costs of many states in the time intervals between the execution of actions. Using our notation, the simplest form of LRTA* operates as follows: to determine action $u_t \in U(s_t)$, the controller first backs up the cost of s_t by setting $f_t(s_t)$ to the minimum of the values $c_{s_t}(u) + \gamma f_{t-1}(j)$ for all actions $u \in U(s_t)$, where j is s_t's successor under action u and $f_{t-1}(j)$ is j's current cost.[13] The costs of all the other states remain the same. The controller then inputs this minimizing action to the system, observes s_{t+1}, and repeats the process.

[13]Note that because $B_t = \{s_t\}$ for all t in LRTA*, k_t always equals t.

This form of LRTA* is almost the special case of RTDP as applied to a deterministic problem in which $B_t = \{s_t\}$ for all $t = 0, 1, \ldots$. It differs from this special case in the following way. Whereas RTDP executes an action that is greedy with respect to f_t, LRTA* executes an action that is greedy with respect to f_{t-1}. This is usually an inconsequential difference because in LRTA* $f_t(j)$ can differ from $f_{t-1}(j)$ only when $j = s_t$, i.e., when s_t is its own successor. LRTA* saves computation by requiring only one minimization at each time step: the minimization required to perform the backup also gives the greedy action. However, in the general case, when RTDP backs up more than one state's cost during each time interval, it makes sense to use the latest estimate of f^* to select an action.

An extended form of LRTA* can also be related to RTDP. In his discussion, Korf [38] assumes that the evaluation of a state may be *augmented by look-ahead search.* This means that instead of using the costs $f_{t-1}(j)$ of s_t's successor states, LRTA* can perform an off-line forward search from s_t to a depth determined by the amount of time and computational resources available. It applies the evaluation function f_{t-1} to the frontier nodes and then backs up these costs to s_t's immediate successors. This is done (roughly) by setting the backed-up cost of each state generated in the forward search to the minimum of the costs of its successors (Korf's "minimin" procedure). These backed-up costs of the successor states are then used to update $f_{t-1}(s_t)$, as described above, but neither these costs nor the backed-up costs of the states generated in the forward search are saved. Despite the fact that backed-up costs for many states have been computed, the new evaluation function, f_t, differs from the old only for s_t. However, within the limits of space constraints, it makes sense to store the backed-up costs for as many states as possible, especially when the controller will experience multiple trials with different starting states. In contrast to LRTA*, RTDP can save all of these backed-up costs in f_{k_t} by executing approriately defined stages of asynchronous DP.

Specifically, saving the backed-up costs produced by Korf's minimin procedure corresponds to executing a number of stages of asynchronous DP equal to one less than the depth of the forward search tree. The first stage synchronously backs up the costs of all the immediate predecessors of the frontier states (using the current costs of the frontier states), the second stage backs up the costs of the states that are the immediate predecessors of these states, etc. Then one additional stage of asnychronous DP to back up the cost of s_t completes the computation of f_{k_t}. Not only does this procedure also apply in the stochastic case, it suggests that other stages of asynchronous DP might be useful as well. These stages might back up the costs of states not in the forward search tree, or they might back up the costs of states in this tree more than once. For example, noting that in general the forward search might generate a graph with cycles, multiple backups of the costs of these states can further improve the information contained in f_{k_t}. All of these possibilities are basically different instances of RTDP and thus converge under the conditions described in the theorems above.

With repeated trials, the information accumulating in the developing estimate of the optimal evaluation function improves control performance. Consequently, LRTA* and RTDP are indeed learning algorithms, as suggested by the name chosen by Korf. However, they do not directly apply to *adaptive* control problems as this term is used in control theory, where it applies to problems in which a complete and accurate model of the system to be controlled is lacking. In the next section we discuss how RTDP can be used in adaptive control problems.

7 Adaptive Control

The versions of value iteration described above—synchronous, Gauss-Seidel, asynchronous, and real-time—require prior knowledge of the system underlying the Markovian decision problem. That is, they require knowledge of the state-transition probabilities, $p_{ij}(u)$, for all states i, j, and all actions $u \in U(i)$, and they require knowledge of the immediate costs $c_i(u)$ for all states i and actions $u \in U(i)$. If the system is deterministic, this means that one must know the successor states and the immediate costs for all the admissible actions for every state. Finding, or approximating, an optimal policy when this knowledge is not available is known as a *Markovian decision problem with incomplete information*, and solution methods for these problems are examples of *adaptive control* methods.[14]

There are two major classes of adaptive methods for Markovian decision problems with incomplete information. Bayesian methods rest on the assumption of a known a priori probability distribution over the class of possible stochastic dynamic systems. As observations accumulate, this distribution is revised via Bayes' rule. Actions are selected by using DP to find a policy that minimizes the expected cost over the set of possible systems as well as over time. Non-Bayesian approaches, in contrast, attempt to arrive at an optimal policy *asymptotically* for *any* system within some pre-specified class of systems. Actions may not be optimal on the basis of prior assumptions and accumulated observations, but the policy should approach an optimal policy in the limit as experience accumulates. Kumar [39] surveys the large literature on both classes of methods and conveys the sublety of the issues as well as the sophistication of the existing theoretical results. Here we restrict attention to non-Bayesian methods because they are more practical for large problems.

Two types of non-Bayesian methods are distinguished. *Indirect* methods explicitly model the dynamic system being controlled. They use *system identification* algorithms to update parameters whose values determine the current system model at any time during control. They typically make control decisions under the assumption that the current model is the true model of the system (what control theorists call the *certainty equivalence principle* [11]). *Direct* methods, on the other hand, form policies without using explicit system models. They directly estimate a policy or information other than a system model, such as an evaluation

[14] Markovian decision problems with incomplete information are not the same as problems with incomplete *state* information in which the controller does not have complete knowledge of the system state at each time step of control. These are sometimes called *partially observable Markovian decision problems*, which despite their relevance for many applications, are beyond the scope of this article.

function, from which a policy can be determined.

For both indirect and direct methods, a central issue is the conflict between controlling the system and exploring its behavior in order to discover how to control it better. This is often called the *conflict between identification and control* because it appears in indirect methods as the conflict between conducting enough exploration to achieve model convergence and the objective of eventually following an optimal policy. Direct methods also require exploration and involve these same issues. Adaptive optimal control algorithms require mechanisms for resolving these problems, but no mechanism is universally favored. Some of the approaches for which rigorous theoretical results are available are reviewed by Kumar [39], and a variety of more heuristic approaches have been studied by Barto and Singh [6], Kaelbling [34], Moore [55], Schmidhuber [63], Sutton [69], Watkins [81], Thrun [79], and Thrun and Möller [78].

In the following subsections, we describe several non-Bayesian methods for solving Markovian decision problems with incomplete information. Although these methods can form the basis of algorithms that can be proved to converge to optimal policies, we do not describe exploration mechanisms with enough rigor for developing the theory in this direction. We call the first method the *generic indirect method.* A system identification algorithm updates a system model at each time step of control, and a conventional DP algorithm is executed *at each time step* based on the current system model. Although this method's computational complexity severely limits its utility, it is representative of most of the approaches described in the engineering literature, and it serves as a reference point for comparative purposes. Next, we describe another indirect method that is the simplest modification of the generic indirect method that takes advantage of RTDP. We call this method *Adaptive* RTDP. The third method we describe is the direct Q-Learning method of Watkins [81]. We then briefly describe hybrid direct/indirect methods.

7.1 The Generic Indirect Method

Indirect adaptive methods for Markovian decision problems with incomplete information estimate the unknown state-transition probabilities and immediate costs based on the history of state transitions and immediate costs observed while the controller and system interact. The usual approach is to define the state-transition probabilies in terms of a parameter, θ, contained in some parameter space, Θ. Thus, for each pair of states $i, j \in S$ and each action $u \in U(i)$, $p(i,j,u,\theta)$ is the state-transition probability corresponding to parameter $\theta \in \Theta$, where the functional dependence on θ has a known form. Further, one usually assumes that there is some $\theta^* \in \Theta$ that is the true parameter, so that $p_{ij}(u) = p(i,j,u,\theta^*)$. The identification task is to estimate θ^* from experience. A common approach takes as the estimate of θ^* at each time step the parameter having the highest probability of generating the observed history, i.e., the maximum-likelihood estimate of θ^*.

The simplest form of this approach to identification is to assume that the unknown parameter is a list of the actual transition probabilities. Then at each time step t the system model consists of the maximum-likelihood estimates, denoted $p^t_{ij}(u)$, of the unknown state-transition probabilities for all pairs of states i, j and actions $u \in U(i)$. Let $n^u_{ij}(t)$ be the observed number of times before time step t that action u was executed when the system was in state i and made a transition to state j. Then $n^u_i(t) = \sum_{j \in S} n^u_{ij}(t)$ is the number of times action u was executed in state i. The maximum-likelihood state-transtion probabilities at time t are

$$p^t_{ij}(u) = \frac{n^u_{ij}(t)}{n^u_i(t)}. \tag{10}$$

If the immediate costs, $c_i(u)$, are also unknown, they can be determined simply by memorizing them as they are observed.[15] If in an infinite number of time steps each action would be taken infinitely often in each state, then this system model converges to the true system. As mentioned above, it is nontrivial to ensure that this occurs while the system is being controlled.

At each time step t, the generic indirect method uses some (non real-time) DP algorithm to determine the optimal evaluation function for the latest system model. Let f^*_t denote this optimal evaluation function. Of course, if the model were correct, then f^*_t would equal f^*, but this is generally not the case. A *certainty equivalence optimal policy* for time step t is any policy that is greedy with respect to f^*_t. Let $\mu^*_t = [\mu^*_t(1), \ldots, \mu^*_t(n)]$ denote any such policy. Then at time step t, $\mu^*_t(s_t)$ is the *certainty equivalence optimal action*. Any of the off-line DP algorithms described above can be used to determine f^*_t, including asynchronous DP. Here it makes sense at each time step to initialize the DP algorithm with final estimate of f^* produced by the DP algorithm completed at the previous time step. The small change in the system model from time step t to $t+1$ means that f^*_t and f^*_{t+1} probably do not differ significantly. As pointed out above, however, the computation required to perform even one DP iteration can be prohibitive in problems with large numbers of states.

What action should the controller execute at time t? The certainty equivalence optimal action, $\mu^*_t(s_t)$, *appears* to be the best based on observations up to time t. Consequently, in pursuing its objective of control, the controller should always execute this action. However, because the current model is not necessarily correct, the controller must also pursue the identification objective, which dictates that it must sometimes select actions other than certainty equivalence optimal actions. It is easy to generate examples in which always following the current certainty equivalence optimal policy prevents convergence to a true optimal policy due to lack of exploration (see, for example, Kumar [39]).

One of the simplest ways to induce exploratory behavior is to make the controller use randomized policies in which actions are chosen according to probabilities that depend on the current evaluation function. Each action always has a nonzero probability of being executed, with the current certainty equivalence optimal action having the highest probability. To

[15] In problems in which the immediate cost is a random function of the current state and action, the maximum-likelihood estimate of an immediate cost is the observed average of the immediate cost for that state and action.

facilitate comparison of algorithms in the simulations described in Section 4.1, we adopt the action-selection method based on the Boltzmann distribution that was used by Watkins [81], Lin [47], and Sutton [69].

This method assigns an execution probability to each admissible action for the current state, where this probability is determined by a rating of each action's utility. We compute a rating, $r(u)$, of each action $u \in U(s_t)$ as follows:

$$r(u) = Q^{f_t^*}(s_t, u).$$

We then transform these ratings (which can be negative and do not sum to one) into a probability mass function over the admissible actions using the Boltzmann distribution: at time step t, the probability that the controller executes action $u \in U(s_t)$ is

$$\mathrm{Prob}(u) = \frac{e^{-r(u)/T}}{\sum_{v \in U(s_t)} e^{-r(v)/T}}, \tag{11}$$

where T is a positive parameter controlling how sharply these probabilities peak at the certainty equivalence optimal action, $\mu_t^*(s_t)$. As T increases, these probabilities become more uniform, and as T decreases, the probability of executing $\mu_t^*(s_t)$ approaches one, while the probabilities of the other actions approach zero. T acts as a kind of "computational temperature" as used in simulated annealing [35] in which T decreases over time. Here it controls the necessary tradeoff between identification and control. At "zero temperature" there is no exploration, and the randomized policy equals the certainty equivalence optimal policy, whereas at "infinite temperature" there is no attempt at control.

In the simulations described in Section 4.1, we introduced exploratory behavior by using the method just described for generating randomized policies, and we let T decrease over time to a pre-selected minimum value as learning progressed. Our choice of this method was dictated by simplicity and our desire to illustrate algorithms that are as "generic" as possible. Without doubt, more sophisticated exploratory behavior would have beneficial effects on the behavior of these algorithms.

7.2 Adaptive Real-Time Dynamic Programming

The generic indirect method just presented relies on executing a non real-time DP algorithm until convergence at each time step. It is straightforward to substitute RTDP, resulting in the indirect method we call *Adaptive* RTDP. This method is exactly the same as RTDP as described in Section 6.1 except that 1) a system model is updated using some on-line system identification method, such as the maximum-likelihood method given by Equation 10; 2) the current system model is used in performing the stages of RTDP instead of the true system model; and 3) the action at each time step is determined by the randomized policy given by Equation 11, or by some other method that balances the identification and control objectives.

Adaptive RTDP is related to a number of algorithms that have been investigated by others. Although Sutton's *Dyna* architecture [69] focuses on Q-Learning and methods based on policy iteration (Section 8), it also encompasses algorithms such as Adaptive RTDP, as he discusses in ref. [70]. Lin [47; 46] also discusses methods closely related to Adaptive RTDP. In the engineering literature, Jalali and Ferguson [32] describe an algorithm that is similar to Adaptive RTDP, although they focus on Markovian decision problems in which performance is measured by the average cost per-time-step instead of the discounted cost we have discussed.

Performing RTDP concurrently with system identification, as in Adaptive RTDP, provides an opportunity to let progress in identification influence the selection of states to which the backup operation is applied. Sutton [69] suggested that it can be advantageous to back up the costs of states for which there is good confidence in the accuracy of the estimated state-transition probabilities. One can devise various measures of confidence in these estimates and direct the algorithm to the states whose cost backups use the most reliable state-transition information according to this confidence measure. At the same time, it is possible to use a confidence measure to direct the selection of actions so that the controller tends to visit regions of the state space where the confidence is *low* so as to improve the model for these regions. This strategy produces exploration that aids identification but can conflict with control. Kaelbling [34], Lin [47], Moore [55], Schmidhuber [63], Sutton [69], Thrun [79], and Thrun and Möller [78] discuss these and other possibilities.

7.3 Q-Learning

Q-Learning is a method proposed by Watkins [81] for solving Markovian decision problems with incomplete information.[16] Unlike the indirect adaptive methods discussed above, it is a direct method because it does not use an explicit model of the dynamic system underlying the decision problem. It directly estimates the optimal Q-values for pairs of states and admissible actions (which we call admissible state-action pairs). Recall from Equation 6 that $Q^*(i,u)$, the optimal Q-value for state i and action $u \in U(i)$, is the cost of generating action u in state i and thereafter following an optimal policy. Any policy selecting actions that are greedy with respect to the optimal Q-values is an optimal policy. Thus, if the optimal Q-values are available, an optimal policy can be determined with relatively little computation.

[16] Watkins [81] actually proposed a family of Q-Learning methods, and what we call Q-Learning in this article is the simplest case, which he called "one-step Q-Learning." He observed that although Q-Learning methods are based on a simple idea, they had not been suggested previously as far as he knew. He further observed, however, that because these problems had been so intensively studied for over thirty years, it would be surprising if no one had studied them earlier. Although the idea of assigning values to state-action pairs formed the basis of Denardo's [24] approach to DP, we have not seen algorithms like Q-Learning for estimating these values that predate Watkins' 1989 dissertation.

We depart somewhat in our presentation from the view taken by Watkins [81] and others (e.g., Sutton [69], Barto and Singh [6]) of Q-Learning as a method for adaptive on-line control. To emphasize Q-Learning's relationship with asynchronous DP, we first present the basic Q-Learning algorithm as an *off-line* asynchronous DP method that is unique in not requiring direct access to the state-transition probabilities of the decision problem. We then describe the more usual on-line view of Q-Learning.

7.3.1 Off-Line Q-Learning

Instead of maintaining an explicit estimate of the optimal evaluation function, as is done by all the methods described above, Q-Learning maintains estimates of the optimal Q-values for each admissible state-action pair. For any state i and action $u \in U(i)$, let $Q_k(i,u)$ be the estimate of $Q^*(i,u)$ available at stage k of the computation. Recalling that f^* is the minimum of the optimal Q-values for each state (Equation 6), we can think of the Q-values at stage k as implicitly defining f_k, a stage-k estimate of f^*, which is given for each state i by

$$f_k(i) = \min_{u \in U(i)} Q_k(i,u). \tag{12}$$

Although Q-values define an evaluation function in this way, they contain more information than the evaluation function. For example, actions can be ranked on the basis of Q-values alone, whereas ranking actions using an evaluation function also requires knowledge of the state-transition probabilities and immediate costs.

Instead of having direct access to the state-transition probabilities, Off-Line Q-Learning only has access to a random function that can generate samples according to these probabilities. Thus, if a state i and an action $u \in U(i)$ are input to this function, it returns a state j with probability $p_{ij}(u)$. Let us call this function `successor` so that $j = \texttt{successor}(i,u)$. The `successor` function amounts to an accurate model of the system in the form of its state-transition probabilities, but Q-Learning does not have access to the probabilities themselves. As we shall see below, in on-line Q-Learning, the role of the `successor` function is played by the system itself.

At each stage k, Off-Line Q-Learning synchronously updates the Q-values of a subset of the admissible state-action pairs and leaves unchanged the Q-values for the other admissible pairs. The subset of admissible state-action pairs whose Q-values are updated changes from stage to stage, and the choice of these subsets determines the precise nature of the algorithm. For each $k = 0, 1, \ldots$, let $S_k^Q \subseteq \{(i,u) | i \in S, u \in U(i)\}$ denote the set of admissible state-action pairs whose Q-values are updated at stage k. For each state-action pair in S_k^Q, it is necessary to define a learning rate parameter that determines how much of the new Q-value is determined by its old value and how much by a backed-up value. Let $\alpha_k(i,u)$, $0 < \alpha_k(i,u) < 1$, denote the learning rate parameter for updating the Q-value of (i,u) at stage k. Then Q_{k+1} is computed as follows: if $(i,u) \in S_k^Q$ then

$$Q_{k+1}(i,u) = (1 - \alpha_k(i,u))Q_k(i,u) + \alpha_k(i,u)[c_i(u) + \gamma f_k(\texttt{successor}(i,u))], \tag{13}$$

where f_k is given by Equation 12. The Q-values for the other admissible state-action pairs remain the same, i.e.,

$$Q_{k+1}(i,u) = Q_k(i,u),$$

for all admissible $(i,u) \notin S_k^Q$. By a Q-Learning backup we mean the application of Equation 13 for a single admissible state-action pair (i,u).

If the Q-value for each admissible state-action pair (i,u) is backed up infinitely often in an infinite number of stages, and if the learning rate parameters $\alpha_k(i,u)$ decrease over the stages k in an appropriate way, then the sequence $\{Q_k(i,u)\}$ generated by Off-Line Q-Learning converges with probability one to $Q^*(i,u)$ as $k \to \infty$ for all admissible pairs (i,u). This is essentially proved by Watkins [81], and Watkins and Dayan present a revised proof in ref. [82]. Appendix B describes a method for meeting the required learning rate conditions that was developed by Darken and Moody [19]. We used this method in obtaining the results for Real-Time Q-Learning on our example problems presented in Section 4.1.

One can gain insight into Off-Line Q-Learning by relating it to asynchronous DP. The stage-k Q-values for all admissible state-action pairs define the evaluation function f_k given by Equation 12. Thus, one can view a stage of Off-Line Q-Learning defined by Equation 13 as updating f_k to f_{k+1}, where for each state i,

$$f_{k+1}(i) = \min_{u \in U(i)} Q_{k+1}(i,u).$$

This evaluation function update does not correspond to a stage of any of the usual DP algorithms because it is based only on samples from `successor` for selected actions determined by the state-action pairs in S_k^Q. A conventional DP backup, in contrast, uses the true expected successor costs over all the admissible actions for a given state.[17]

It is accurate to think of Off-Line Q-Learning as a *more asynchronous version of asynchronous DP*. Asynchronous DP is asynchronous at the level of states, and the backup operation for each state requires minimizing expected costs over all admissible actions for that state. The amount of computation required to determine the expected cost for each admissible action depends on the number of *possible* successor states for that action, which can as large as the total number of states in stochastic problems. Off-Line Q-Learning, on the other hand, is asynchronous at the level of admissible state-action pairs. Although each Q-Learning backup requires minimizing over all the admissible actions for a give state

[17]However, stage k of Off-Line Q-Learning has the same effect as the stage of asynchronous DP using S_k in the special case in which 1) the problem is deterministic, 2) S_k^Q is the set of all admissible state-action pairs for states in S_k, and 3) $\alpha_k(i,u) = 1$ for all admissible state-action pairs (i,u).

in order to calculate (via Equation 12) $f_k(\texttt{successor}(i,u))$ used in Equation 13,[18] it does not require computation proportional to the number of possible successor states. Thus, in the stochastic case, an asynchronous DP backup can require $O(mn)$ computational steps, whereas a Q-Learning backup requires only $O(m)$. This advantage is offset by the increased space complexity of Q-Learning and the fact that a Q-Learning backup takes less information into account than does a backup of asynchronous DP: an asynchronous DP backup is comparable to many Q-Learning backups. Nevertheless, because the computation required by a Q-Learning backup can be much less than that required by an asynchronous DP backup, Q-Learning can be advantageous when stages have to be computed quickly despite a large number of possible successor states, as in real-time applications which we discuss next.

7.3.2 Real-Time Q-Learning

Off-Line Q-Learning can be turned into an on-line algorithm by executing it concurrently with control. If a current system model provides an approximate **successor** function, the result is an indirect adaptive method identical to Adaptive RTDP (Section 7.2) except that stages of Off-Line Q-Learning substitute for stages of asynchronous DP. This can have advantages over Adaptive RTDP when the number of admissible actions is large. However, we use the term *Real-Time Q-Learning* for the case originally discussed by Watkins [81] in which there is no model of the system underlying the decision problem and the real system acts as the **successor** function. This direct adaptive algorithm backs up the Q-value for only a single state-action pair at each time step of control, where this state-action pair consists of the observed current state and the action actually executed. Using Real-Time Q-Learning, therefore, one can compute an optimal policy without froming an explicit model of the system underlying the decision problem.

Specifically, assume that at each time step t the controller observes state s_t and has available the estimated optimal Q-values produced by all the preceding stages of Real-Time Q-Learning. We denote these estimates $Q_t(i,u)$ for all admissible state-action pairs (i,u). The controller selects an action $u_t \in U(s_t)$ using this information in some manner that allows for exploration. After executing u_t, the controller receives the immediate cost $c_{s_t}(u_t)$ while the system state changes to s_{t+1}. Then Q_{t+1} is computed as follows:

$$Q_{t+1}(s_t,u_t) = (1-\alpha_t(s_t,u_t))Q_t(s_t,u_t) + \alpha_t(s_t,u_t)[c_{s_t}(u_t) + \gamma f_t(s_{t+1})], \tag{14}$$

[18] This complete minimization can sometimes be avoided as follows. Whenever a $Q_k(i,u)$ is backed up, if its new value, $Q_{k+1}(i,u)$, is smaller than $f_k(i)$, then $f_{k+1}(i)$ is set to this smaller value. If its new value is larger than $f_k(i)$, then if $f_k(i) = Q_k(i,u)$ and $f_k(i) \neq Q_k(i,u')$ for any $u' \neq u$, then $f_{k+1}(i)$ is found by explicitly minimizing the current Q-values for state i over the admissible actions. This is the case in which u is the sole greedy action with respect to $f_k(i)$. Otherwise, nothing is done, i.e., $f_{k+1}(i) = f_k(i)$. This procedure therefore computes the minimization in Equation 12 explicitly only when updating the Q-values for state-action pairs (i,u) in which u is the sole greedy action for i and the Q-value increases.

where $f_t(s_{t+1}) = \min_{u \in U(s_{t+1})} Q_t(s_{t+1}, u)$ and $\alpha_t(s_t, u_t)$ is the learning rate parameter at time step t for the current state-action pair. The Q-values for all the other admissible state-action pairs remain the same, i.e,

$$Q_{t+1}(i, u) = Q_t(i, u),$$

for all admissible $(i, u) \neq (s_t, u_t)$. This process repeats for each time step.

As far as convergence is concerned, Real-Time Q-Learning is the special case of Off-Line Q-Learning in which S_t^Q, the set of state-action pairs whose Q-values are backed up at each step (or stage) t, is $\{(s_t, u_t)\}$. Thus, the sequence of Q-values generated by Real-Time Q-Learning converges to the true values given by Q^* under the conditions required by for convergence of Off-Line Q-Learning. This means that each admissible action must be performed in each state infinitely often in an infinite number of control steps. It is also noteworthy, as pointed out by Dayan [22], that when there is only one admissible action for each state, Real-Time Q-Learning reduces to the TD(0) algorithm investigated by Sutton [68].

To define a complete adaptive control algorithm making use of Real-Time Q-Learning it is necessary to specify how each action is selected based on the current Q-values. Convergence to an optimal policy requires the same kind of exploration required by indirect methods to facilitate system identification as discussed above. Therefore, given a method for selecting an action from a current evaluation function, such as the randomized method described above (Equation 11), if this method leads to convergence of an indirect method, it also leads to convergence of the corresponding direct method based Real-Time Q-Learning.

7.3.3 Other Q-Learning Methods

In Real-Time Q-Learning, the real system underlying the decision problem plays the role of the `successor` function. However, it is also possible to define the `successor` function sometimes by the real system and sometimes by a system model. For state-action pairs actually experienced during control, the real system provides the `successor` function; for other state-action pairs, a system model provides an approximate `successor` function. Sutton [69] has studied this approach in an algorithm called *Dyna-Q*, which performs the basic Q-Learning backup using both actual state transitions as well as hypothetical state transitions simulated by a system model. Performing the Q-Learning backup on hypothetical state transitions amounts to running multiple stages of Off-Line Q-Learning in the intervals between times at which the controller executes actions. A step of Real-Time Q-Learning is performed based on each actual state transition. This is obviously only one of many possible ways to combine direct and indirect adaptive methods as emphasized in Sutton's discussion of the general Dyna learning architecture [69].

It is also possible to modify the basic Q-Learning method in a variety of ways in order to enhance its efficiency. For example, Lin [47] has studied a method in which Real-Time

Q-Learning is augmented with model-based Off-Line Q-Learning only if one action does not clearly stand out as preferable according to the current Q-values. In this case, Off-Line Q-Learning is carried out to backup the Q-values for all of the admissible actions that are "promising" according to the latest Q-values for the current state. Watkins [81] describes a family of Q-Learning methods in which Q-values are backed up based on information gained over sequences of state transitions. One way to implement this kind of extension is to use the "eligibility trace" idea [2; 37; 67; 72; 68] to back up the Q-values of all the state-action pairs experienced in the past, with the magnitudes of the backups decreasing to zero with increasing time in the past. Sutton's [68] TD(λ) algorithms illustrate this idea. Attempting to present all of the combinations and variations of Q-Learning methods that have been, or could be, described is well beyond the scope of the present article. Barto and Singh [6], Dayan [20; 21], Lin [47; 46], Moore [57], and Sutton [69] present comparative empirical studies of some of the adaptive algorithms based on Q-Learning.

8 Methods based on Explicit Policy Representations

All of the DP-based learning algorithms described above, both non-adaptive and adaptive cases, use an explicit representation of either an evaluation function or a function giving the Q-values of admissible state-action pairs. These functions are used in computing the action at each time step, but the policy so defined is not explicitly stored. There are a number of other real-time learning and control methods based on DP in which policies as well as evaluation functions are stored and updated at each time step of control. Unlike the methods addressed in this article, these methods are more closely related to the *policy iteration* DP algorithm than the value iteration algorithms discussed in Section 5.

Policy iteration (see, e.g., Bertsekas [11]) alternates two phases: 1) a *policy evaluation* phase, in which the evaluation function for the current policy is determined, and 2) a *policy improvement* phase, in which the current policy is updated to be greedy with respect to the current evaluation function. One way to evaluate a policy is by executing one of the value iteration algorithms discussed in Section 5 under the assumption that there is only one admissible action for each state, namely, the action specified by the policy being evaluated. Alternatively, explicit matrix inversion methods can be used. Although policy evaluation does not require repeated minimizing over all admissible actions, it can still require too much computation to be practical for large state sets. More feasible is *modified policy interation* [59], which is policy iteration except that the policy evaluation phase is not executed to completion before each policy improvement phase. Real-time algorithms based on policy iteration effectively work by executing an asynchronous form of modified policy iteration concurrently with control.

Examples of such methods appear in the pole-balancing system of Barto, Sutton, and Anderson [2; 67] (also refs. [1; 67]) and the *Dyna-PI* method of Sutton [69] (where PI means

Policy Iteration). Barto, Sutton, and Watkins [4; 3] discuss the connection between these methods and policy iteration in some detail. In this article we do not discuss learning algorithms based on policy iteration because their theory is not yet as well understood as is the theory of learning algorithms based on asynchronous value iteration. However, Williams and Baird [91] have made a valuable contribution to this theory by addressing DP algorithms that are asynchronous at a grain finer than that of either asychronous DP or Q-Learning. These algorithms include value iteration, policy iteration, and modified policy iteration as special cases. Integrating their theory with that presented here is beyond the scope of this article.

9 Storing Evaluation Functions

An issue of great practical importance in implementing any of the algorithms described in this article is how evaluation functions are represented and stored.[19] The theoretical results we have described assume a lookup-table representation of evaluation functions, which—at least in principle—is always possible when the number of states and admissible actions is finite, as assumed throughout this article. In applying conventional DP to problems involving continuous states and/or actions, the usual practice is to discretize the ranges of the continuous state variables and then use the lookup-table representation (cf. the "boxes" representation used by Michie and Chambers [52] and Barto, Sutton, and Anderson [2]). This leads to space complexity exponential in the number of state variables, the situation prompting Bellman [9] to coin the phrase "curse of dimensionality." The methods described in this article based on asynchronous DP and Q-Learning do not circumvent the curse of dimensionality, although the focusing behavior of Trial-Based RTDP in stochastic shortest path problems with designated start states can reduce the storage requirement if memory is allocated incrementally during trials.

A number of methods exist for making the lookup-table representation more efficient when it is not necessary to store the costs of all possible states. Hash table methods, as assumed by Korf [38] for LRTA*, permit efficient storage and retrieval when the costs of a small enough subset of the possible states need to be stored. Similarly, using the *kd-tree* data structure to access state costs, as explored by Moore [55; 56], can provide efficient storage and retrieval of the costs of a finite set of states from a k-dimensional state space. The theoretical results described in this article extend to these methods because they preserve the integrity of the stored costs (assuming hash collisions are resolved).

Other approaches to storing evaluation functions use function approximation methods based on parameterized models. For example, in Samuel's [61] checkers player, the evaluation function was approximated as a weighted sum of the values of a set of features describing checkerboard configurations. The basic backup operation was performed on the weights,

[19]All of our comments here also apply to storing the Q-values of admissible state-action pairs.

not on the state costs themselves. The weights were adjusted to reduce to the discrepancy between the current cost of a state and its backed-up cost. This approach inspired a variety of more recent studies using parameterized function approximations. The discrepancy supplies the error for any error-correction procedure that approximates functions based on a training set of function samples. This is a form of supervised learning, or learning from examples, and provides the natural way to make use of connectionist networks as shown, for example, by Anderson [1] and Tesauro [77]. Parametric approximations of evaluation functions are useful because they can generalize beyond the training data to supply cost estimates for states that have not yet been visited, an important factor for large state sets.

In fact, almost any supervised learning method, and its associated manner of representing hypotheses, can be adapted for approximating evaluation functions. This includes symbolic methods for learning from examples. These methods also generalize beyond the training information, which is derived from the back-up operations of various DP-based algorithms. For example, Chapman and Kaelbling [15] and Tan [76] adapt decision-tree methods, and Mahadevan and Connell [49] use a statistical clustering method. Yee [94] discusses function approximation from the perspective of its use with DP-based learning algorithms.

Despite the large number of studies in which the principles of DP have been combined with generalizing methods for approximating evaluation functions, *the theoretical results presented in this article do not automatically extend to these approaches.* Although generalization can be helpful in approximating an optimal evaluation function, it is often detrimental to the convergence of the underlying asynchronous DP algorithm, as pointed out by Watkins [81] and illustrated with a simple example by Bradtke [13]. Even if a function approximation scheme can adequately represent the optimal evaluation function when trained on samples from this function, it does not follow that an adequate representation will result from an iterative DP algorithm that uses such an approximation scheme at each stage. The issues are much the same as those that arise in numerically solving differential equations. The objective of these problems is to approximate the function that is the solution of a differential equation (for given boundary conditions) in the absence of training examples drawn from the true solution. In other words, the objective is to *solve approximately* the differential equation, not just to approximate its solution. Here, we are interested in approximately solving the Bellman Optimality Equation and not the easier problem of approximating a solution that is already available.

There is an extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus [7], Bellman, Kalaba, and Kotkin [8], Daniel [18], Kushner and Dupuis [41]). However, most of this literature is devoted to off-line algorithms for cases in which there is a complete model of the decision problem. Adapting techniques from this literature to produce approximation methods for RTDP and other DP-based learning algorithms is a challenge for future research.

To the best of our knowledge, there are only a few theoretical results that directly address the use of generalizing methods with DP-based learning algorithms. The results of Sutton [68] and Dayan [22] concern using TD methods to evaluate a given policy as a linear combination of a complete set of linearly independent basis vectors. Unfortunately these results do not address the problem of representing an evaluation function more compactly than it would be represented in a lookup table. Bradtke [13] addresses the problem of learning Q-values that are quadratic functions of a continuous state, but these results are restricted to linear quadratic regulation problems. However, Singh and Yee [66] point out that in the discounted case, small errors in approximating an evaluation function (or a function giving Q-values) lead at worst to small decrements in the performance of a controller using the approximate evaluation function as the basis of control. Without such a result, it might seem plausible that small evaluation errors can drastically undermine control performance—a condition which, if true, would raise concerns about combining DP-based learning with function approximation. Much more research is needed to provide a better understanding of how function approximation methods can be used effectively with the algorithms described in this article.

10 Illustrations of DP-based Learning

We used the race track problem described in Section 4.1 to illustrate and compare conventional DP, RTDP, Adaptive RTDP, and Real-Time Q-Learning using the two race tracks shown in Figure 1. The small race track shown in Panel A has 4 start states, 87 goal states, and $9,115$ states reachable from the start states by any policy. We have not shown the squares on which the car might land after crossing the finish line. The larger race track shown in Panel B has 6 start states, 590 goal states, and $22,576$ states reachable from the start states. We set $p = 0.1$ so that the controller's intended actions were executed with probability 0.9.

We applied conventional Gauss-Seidel DP to each race track problem, by which we mean Gauss-Seidel value iteration as defined in Section 5.2, with $\gamma = 1$ and with the initial evaluation function assigning zero cost to each state. Gauss-Seidel DP converges under these conditions because it is a special case of asnychronous DP, which converges here because the conditions given in Section 5.3 are satisfied. Specifically, it is clear that there is at least one proper policy for either track (it is possible for the car to reach the finish line from any reachable state, although it may have to hit the wall and restart to do so) and every improper policy incurs infinite cost for at least one state because the immediate costs of all non-goal states are positive. We selected a state ordering for applying Gauss-Seidel DP without concern for any influence it might have on convergence rate (although we found that with the selected ordering, Gauss-Seidel DP converged in approximately half the number of sweeps as did synchronous DP).

Table 1 summarizes the small and larger race track problems and the computational effort required to solve them using Gauss-Seidel DP. Gauss-Seidel DP was considered to have converged to the optimal evaluation function when the maximum cost change over all states between two successive sweeps was less than 10^{-4}. We estimated the number of relevant states for each race track, i.e., the number of states reachable from the start states under any optimal policy, by counting the states visited while executing optimal actions for 10^7 trials.

We also estimated the earliest point in the DP computation at which the optimal evaluation function approximation was good enough so that the corresponding greedy policy was an optimal policy. (Recall that an optimal policy can be a greedy policy with respect to many evaluation functions.) We did this by running 10^7 test trials after each sweep using a policy that was greedy with respect to the evaluation function produced by that sweep. For each sweep, we recorded the average path length produced over these test trials. After convergence of Gauss-Seidel DP, we compared these averages with the optimal expected path length obtained by the DP algorithm, noting the sweep after which the average path length was first within 10^{-2} of the optimal. The resulting numbers of sweeps and backups are listed in Table 1 in the columns labeled "Number of GSDP sweeps to optimal policy" and "Number of GSDP backups to optimal policy." Although optimal policies emerged considerably earlier in these computations than did the optimal evaluation functions, it is important to note that this estimation process is not a part of conventional off-line value iteration algorithms and requires a considerable amount of additional computation.[20] Nevertheless, the resulting numbers of backups are useful in assessing the computational requirements of the real-time algorithms, which should allow controllers to follow optimal policies after comparable numbers of backups.

We applied RTDP, Adaptive RTDP, and Real-Time Q-Learning to both race track problems. Because all the immediate costs are positive, we know that $f^*(i)$ must be non-negative for all states i. Thus, setting the initial costs of all the states to zero produces a non-overestimating initial evaluation function as required by Theorem 3. We applied the real-time algorithms in a trial-based manner, starting each trial with the car placed on the starting line with zero velocity, where each square on the starting line was selected with equal probability. A trial ended when the car reached a goal state. Thus, according to Theorem 3, with $\gamma = 1$, RTDP will converge to the optimal evaluation function with repeated trials. Although RTDP and Adaptive RTDP can back up the costs of many states at each control step, we restricted attention to the simplest case in which they only back up the cost of the current state at each time step. This is the case in which $B_t = \{s_t\}$ for all t. Obviously, all of these algorithms were applied in simulation mode.

We executed 25 runs of each algorithm using different random number seeds, where a run is a sequence of trials beginning with the evaluation function initialized to zero. To

[20] Policy iteration algorithms address this problem by explicitly generating a sequence of improving policies, but updating a policy requires computing its corresponding evaluation function, which is generally a time-consuming computation.

	Small track	*Larger track*
Number of reachable states	$9,115$	$22,576$
Number of goal states	87	590
Est. number of relevant states	599	$2,618$
Optimum exp. path length	14.67	24.10
Number of GSDP sweeps to convergence	28	38
Number of GSDP backups to convergence	$252,784$	$835,468$
Number of GSDP sweeps to optimal policy	15	24
Number of GSDP backups to optimal policy	$136,725$	$541,824$

Table 1: Example Race Track Problems. The results were obtained by executing Gauss-Seidel DP (GSDP).

monitor the performance of each algorithm, we kept track of path lengths, that is, how many moves the car took in going from the starting line to the finish line, in each trial of each run. To record these data, we divided each run into a sequence of disjoint *epochs*, where an epoch is a sequence of 20 consecutive trials. By an *epoch path length* we mean the average of the path lengths generated during an epoch using a given algorithm. Adaptive RTDP and Real-Time Q-Learning were applied under conditions of incomplete information, and for these algorithms we induced exploratory behavior by using randomized policies based on the Boltzmann distribution as described in Section 7.1. To control the tradeoff between identification and control, we decreased the parameter T in Equation 11 after each move until it reached a pre-selected minimum value; T was initialized at the beginning of each run. Parameter values and additional simulation details are provided in Appendix B.

Figure 2 shows results for RTDP (Panel A), Adaptive RTDP (Panel B), and Real-Time Q-Learning (Panel C). The central line in each graph shows the epoch path length averaged over the 25 runs of the corresponding algorithm. The upper and lower lines show ± 1 standard deviation about this average for the sample of 25 runs. Although the average epoch path lengths for the initial several epochs of each algorithm are too large to show on the graphs, it is useful to note that the average epoch path lengths for the first epoch of RTDP, Adaptive RTDP, and Real-Time Q-Learning are respectively 455, 866, and 13,403 moves. That these initial average path lengths are so large, especially for Real-Time Q-Learning, reflects the primitive nature of our exploration strategy.

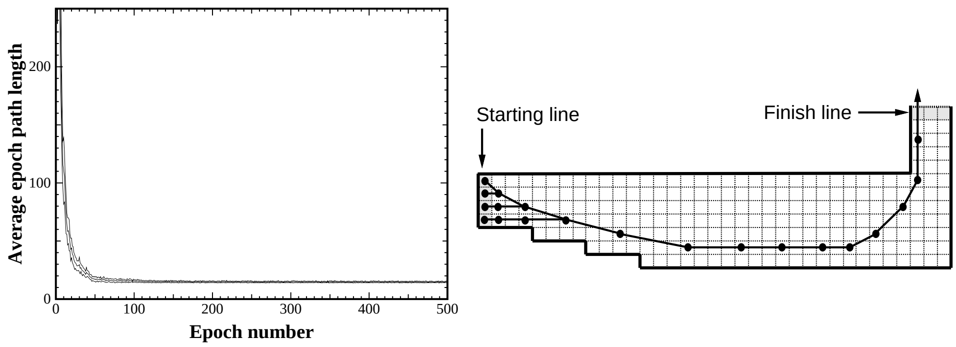
It is clear from the graphs that in this problem RTDP learned faster (and with less variance) than Adaptive RTDP and Real-Time Q-Learning, when learning rate is measured in terms of the number of epochs (numbers of moves are given in Table 2 discussed below). This is not surprising given the differences between the versions of the problem with complete information (Panel A) and with incomplete information (Panels B and C). That the performances of RTDP and Adaptive RTDP were so similar despite these differences reflects the fact that the maximum likelihood system identification procedure used by the latter algorithm converged rapidly on relevant states due to the low level of stochasticity in the problem ($p = 0.1$). These graphs also show that Real-Time Q-Learning takes very many more epochs than do RTDP and Adaptive RTDP to reach a similar level of performance. This reflects the fact that each backup in Real-Time Q-Learning takes into account less information than do the backups in RTDP or Adaptive RTDP, a disadvantage somewhat offset by the relative computational simplicity of each Q-Learning backup. Figure 3 shows the Real-Time Q-Learning results out to 5,000 epochs.

A convenient way to show the policies that result from these algorithms is to show the paths the car would follow from each start state if all sources of randomness were turned off; that is, if both random exploration and the randomness in the problem's state transition function were turned off. At the right in each panel of Figure 2 are paths generated in this way by the policies produced after each algorithm was judged to have "effectively converged." We inspected the graphs to find the smallest epoch numbers at which the average epoch path lengths essentially reached their asymptotic levels: 200 epochs for RTDP (Panel A), 300 epochs for Adaptive RTDP (Panel B), and 2,000 epochs for Real-Time Q-Learning (Panel C). Treated with appropriate caution, these effective convergence times are useful in comparing algorithms.

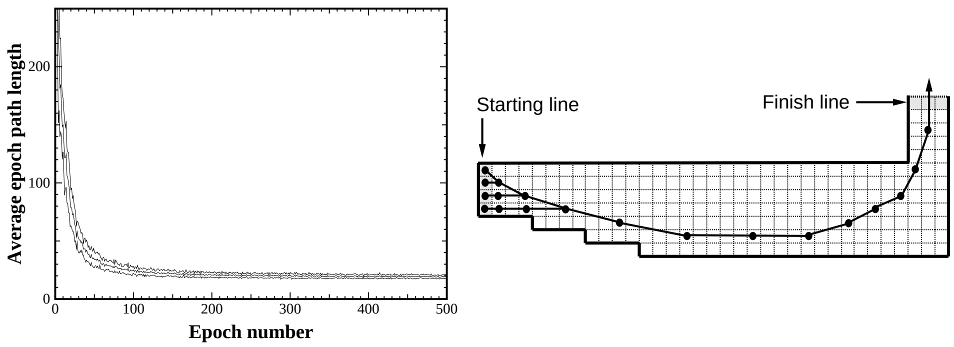
The path shown in Panel A of Figure 2 is optimal in the sense that it was produced in noiseless conditions by a policy that is optimal for the stochastic problem. The paths in Panels B and C, on the other hand, were not generated by an optimal policy despite the fact that each is a move shorter than the path of Panel A. The control decisions made toward the end of the track by these suboptimal policies produce higher probability that the car will collide with the track boundary under stochastic conditions. Although we do not illustrate it here, as the amount of uncertainty in the problem increases (increasing p), optimal policies generate paths that are more "conservative" in the sense of keeping safer distances from the track boundary and maintaining lower velocities.

Table 2 provides additional information about the performance of the real-time algorithms on the small track. For comparative purposes, the table includes a column for Gauss-Seidel DP. We estimated the path length after the effective convergence of RTDP, Adaptive RTDP, and Real-Time Q-Learning by executing 500 test trials with learning turned off using the policy produced at effective convergence of each algorithm. We also turned off the random exploration used by the latter two algorithms. The row of Table 2 labeled "Est. path length at

A



B



C

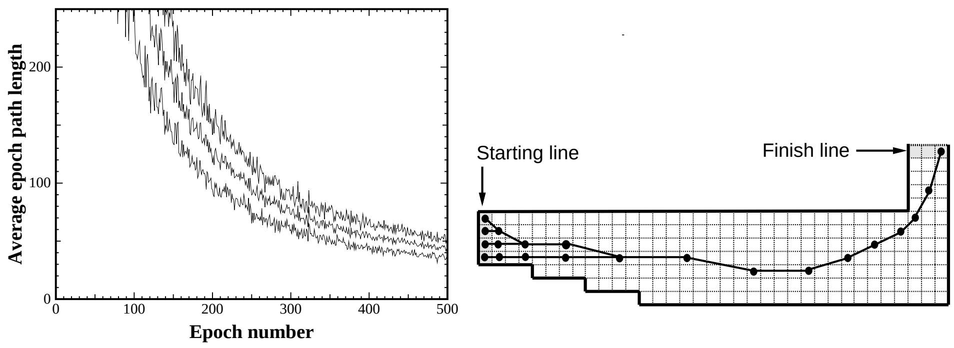


Figure 2: Performance of Three Real-Time Learning Algorithms on the Small Track. Panel A: RTDP. Panel B: Adaptive RTDP. Panel C: Real-Time Q-Learning. The central line in each graph shows the epoch path length averaged over the 25 runs of the corresponding algorithm. The upper and lower lines show ± 1 standard deviation of the epoch path length for the sample of 25 runs. Exploration was controlled for Adaptive RTDP and Real-Time Q-Learning by decreasing T after each move until it reached a pre-selected minimum value. The right side of each panel shows the paths the car would follow in noiseless conditions from each start state after effective convergence of the corresponding algorithm.

	GSDP	*RTDP*	*ARTDP*	*RTQ*
Ave. time to effective convergence	28 sweeps	200 epochs	300 epochs	2,000 epochs
Est. path length at effective convergence	14.56	14.83	15.10	15.44
Ave. number of backups	252,784	127,538	218,554	2,961,790
Ave. number of backups per epoch	-	638	728	1,481
% of states backed up ≤ 100 times	-	98.45	96.47	53.34
% of states backed up ≤ 10 times	-	80.51	65.41	6.68
% of states backed up 0 times	-	3.18	1.74	1.56

Table 2: Summary of Learning Performance on the Small Track for Real-Time DP (RTDP), Adaptive Real-Time DP (ARTDP), and Real-Time Q-Learning (RTQ). The amount of computation required by Gauss-Seidel DP (GSDP) is included for comparative purposes.

effective convergence" gives the average path length over these test trials.[21] RTDP is most directly comparable to Gauss-Seidel DP. After about 200 epochs, or 4,000 trials, RTDP improved control performance to the point where a trial took an average of 14.83 moves. RTDP performed an average of 127,538 backups in reaching this level of performance, about half the number required by Gauss-Seidel DP to converge to the optimal evaluation function. This number of backups is comparable to the 136,725 backups in the 15 sweeps of Gauss-Seidel DP after which the resulting evaluation function defines an optimal policy (Table 1).

Another way to compare Gauss-Seidel DP and RTDP is to examine how the backups they perform are distributed over the states. Whereas the cost of every state was backed up in each sweep of Gauss-Seidel DP, RTDP focused backups on fewer states. For example, in the first 200 epochs of a average run, RTDP backed up the costs of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the costs of about 290 states were not backed up at all in an average run. Although we did not collect these

[21]These path length estimates are somewhat smaller than the average epoch path lengths shown at effective convergence in the graphs of Figure 2 because they were produced with exploration turned off, whereas the graphs show path lengths produced with random exploration turned on. For Gauss-Seidel DP, we averaged over the costs of the start states given by the computed optimal evaluation function to obtain the estimated path length listed in Table 2.

statistics for RTDP after 200 epochs, it became even more focused on the states on optimal paths.

Not surprisingly, solving the problem under conditions of incomplete information requires more backups. Adaptive RTDP took 300 epochs, or an average of 218,554 backups, to achieve trials averaging 15.1 moves at effective convergence. Real-time Q-Learning took 2,000 epochs, or an average of 2,961,790 backups, to achieve a somewhat less skillful level of performance (see Figure 3). Examining how these backups were distributed over states shows that Adaptive RTDP was considerably more focused than was Real-Time Q-Learning. In the first 300 epochs Adaptive RTDP backed up 96.47% of the states no more than 100 times and 65.41% of the states no more than 10 times. On the other hand, in $2,000$ epochs Real-Time Q-Learning backed up Q-values for 53.34% of the states no more than 100 times and only 6.68% of the states no more than 10 times.[22] Again, these results for Real-Time Q-Learning reflect the inadequacy of our primitive exploration strategy for this algorithm.

Figure 4 shows results for RTDP, Adaptive RTDP, and Real-Time Q-Learning on the larger race track, and Table 3 provides additional information. These results were obtained under the same conditions described above for the small track. Figure 5 shows the Real-Time Q-Learning results for the larger track out to 7,500 epochs. We judged that RTDP, Adaptive RTDP, and Real-Time Q-Learning effectively converged at 500, 400, and 3,000 epochs respectively. That Adaptive RTDP effectively converged faster than RTDP in terms of the number of epochs is partially due to the fact that its epochs tended to have more moves, and hence more backups, than the epochs of RTDP. We can see that to achieve slightly suboptimal performance, RTDP required about 62% of the computation of conventional Gauss-Seidel DP. The average epoch path lengths for the initial epoch of each algorithm, which are too large to show on the graphs, are 7,198, 8,749, and 180,358 moves, respectively, for RTDP, Adaptive RTDP, and Real-Time Q-Learning. Again, these large numbers of moves, especially for Real-Time Q-Learning, reflect the primitive nature of our exploration strategy. The paths shown at the right in each panel of Figure 4 were generated in noiseless conditions by the policies produced at effective convergence of the corresponding algorithms. The path shown in Panel A of Figure 4 is optimal in the sense that it was produced in noiseless conditions by a policy that is optimal for the stochastic problem. The paths in Panels B and C, on the other hand, were generated by slightly suboptimal policies.

Although these simulations are not definitive comparisons of the real-time algorithms with conventional DP, they illustrate some of their features. Whereas Gauss-Seidel DP continued to back up the costs of all the states, the real-time algorithms strongly focused on subsets of the states that were relevant to the control objectives. This focus became increasingly narrow as learning continued. Because the convergence theorem for Trial-Based RTDP applies to the simulations of RTDP, we know that this algorithm eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved

[22] We considerated a Q-value for a state i to be backed up whenever $Q(i, u)$ was updated for some $u \in U(i)$.

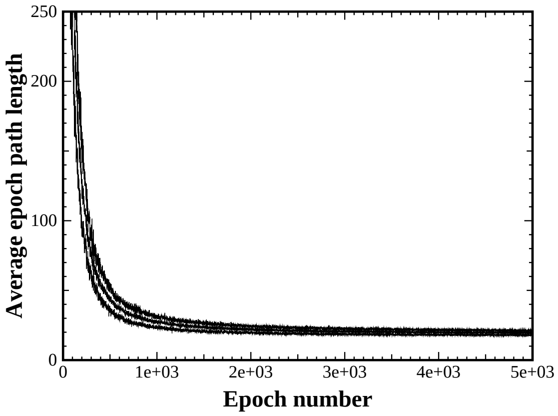


Figure 3: Performance of Real-Time Q-Learning on The Small Track for 5,000 Epochs. The initial part of the graph shows the data plotted in Panel C of Figure 2 but at a different horizontal scale.

	GSDP	*RTDP*	*ARTDP*	*RTQ*
Ave. time to effective convergence	38 sweeps	500 epochs	400 epochs	3,000 epochs
Est. path length at effective convergence	24.10	24.62	24.72	25.04
Ave. number of backups	835,468	517,356	653,774	10,330,994
Ave. number of backups per epoch	-	1,035	1,634	3,444
% of states backed up ≤ 100 times	-	97.77	90.03	52.43
% of states backed up ≤ 10 times	-	70.46	59.90	8.28
% of states backed up 0 times	-	8.17	3.53	2.70

Table 3: Summary of Learning Performance on the Larger Track for Real-Time DP (RTDP), Adaptive Real-Time DP (ARTDP), and Real-Time Q-Learning (RTQ). The amount of computation required by Gauss-Seidel DP (GSDP) is included for comparative purposes.

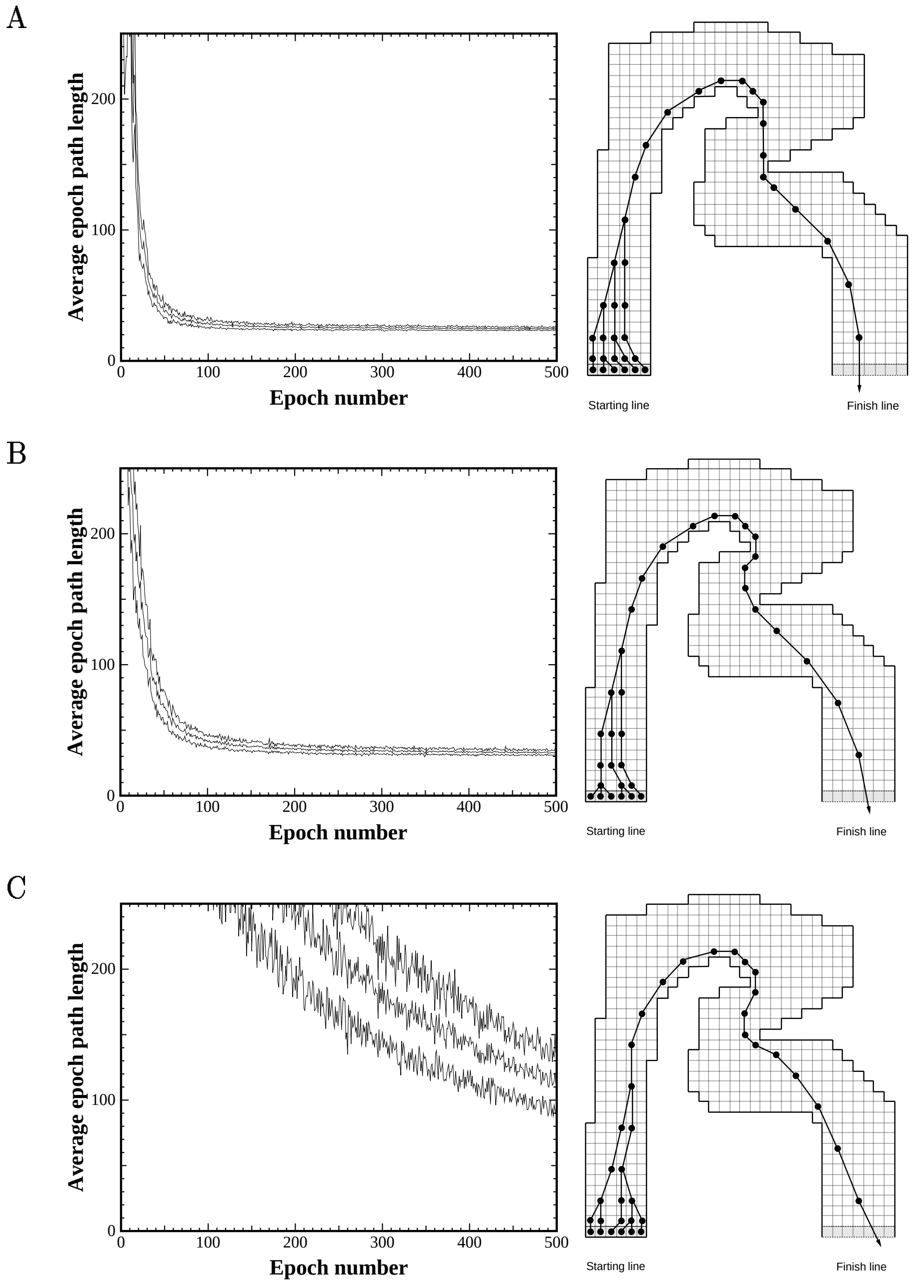


Figure 4: Performance of Three Real-Time Learning Algorithms on the Larger Track. Panel A: RTDP. Panel B: Adaptive RTDP. Panel C: Real-Time Q-Learning. The central line in each graph shows the epoch path length averaged over the 25 runs of the corresponding algorithm. The upper and lower lines show ± 1 standard deviation of the epoch path length for the sample of 25 runs. Exploration was controlled for Adaptive RTDP and Real-Time Q-Learning by decreasing T after each move until it reached a pre-selected minimum value. The right side of each panel shows the paths the car would follow in noiseless conditions from each start state after effective convergence of the corresponding algorithm.

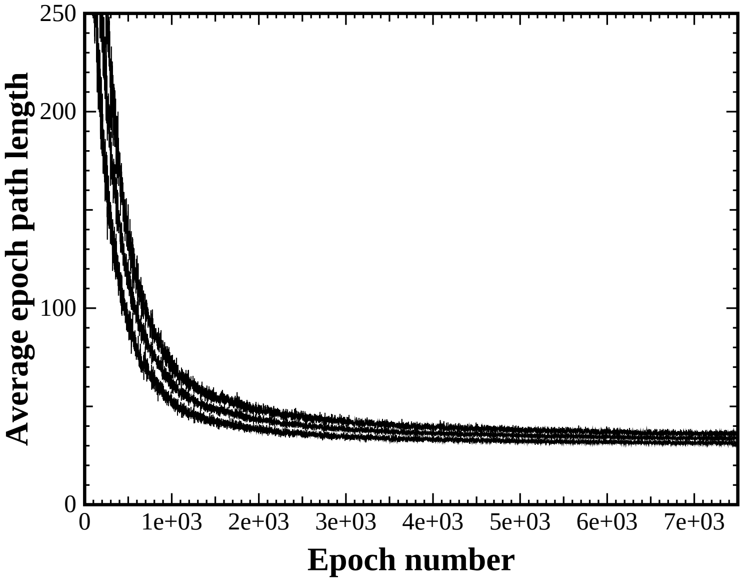


Figure 5: Performance of Real-Time Q-Learning on The Larger Track for 7,500 Epochs. The initial part of the graph shows the same data as plotted in Panel C of Figure 4 but at a different horizontal scale.

nearly optimal control performance with about 50% of the computation of Gauss-Seidel DP on the small track and about 62% of the computation of Gauss-Seidel DP on the larger track. Adaptive RTDP and Real-Time Q-Learning also focused on progressively fewer states, but we did not run the generic indirect method for comparison because it is too inefficient to apply to problems with as many states as our race track problems: It would have to perform at least one complete sweep for each move. In sharp contrast, the amount of computation required by each of the real-time algorithms for each move was small enough not to have been a limiting factor in the simulations.[23]

The results described here for Adaptive RTDP and Real-Time Q-Learning were produced by using an exploration strategy that decreased the randomness in selecting actions by decreasing T after each move until it reached a pre-selected minimum value. Although not described here, we also conducted experiments with different minimum values and with decreasing T after trials instead of after moves. Performance of the algorithms was much altered (for the worse) by these changes. Although we made no systematic attempt to investigate the effects of various exploration strategies, it is clear that the performance of these algorithms is highly sensitive to how exploration is introduced and controlled.

[23]However, in implementing Adaptive RTDP on the race track problems, we took advantage of our knowledge that for any action there are only two possible successors to any state. This allowed us to avoid performing at each move the n divisions required in a straightforward implemention of Equation 10. This is not possible under the general conditions of incomplete information.

How the algorithms scale up to larger problems is also not adequately addressed by our simulations. Although the results with the small and the larger race track give some indication as to how the algorithms might scale, this collection of problems is not adequate for studying this issue. The variability of an algorithm's performance as a function of problem details other than the size of its state and action sets make it difficult to extrapolate from its performance on just two problems. Proceeding to larger problems is hampered by the large space requirements of these algorithms if they continue to use lookup tables for storing evaluation functions. Tesauro's TD-Gammon system [77] is an encouraging data point for using DP-based learning in conjunction with function approximation methods in problems much larger than those described here, but continued theoretical research is necessary to address the computational complexity of real-time DP algorithms. What is clear from our simulations, however, is that real-time DP algorithms can confer significant computational advantages over conventional off-line DP algorithms.

In concluding our discussion of the race track problem, we again point out that it is misleading to think of our application DP-based learning algorithms to this problem as the most productive way to apply them to realistic robot navigation tasks. For example, DP-based learning applied to this formulation of a race track problem refines skill in racing on a specific track. This skill does not transfer to other tracks due to the specificity with which a track is represented. More realistic applications of DP-based learning to robot navigation requires more abstract states and actions, as in the work of Lin [45] and Mahadevan and Connell [49].

11 Discussion

Conventional DP algorithms are of limited utility for problems with large state spaces, such as the combinatorial state spaces of many problems of interest in AI, because they require fully expanding all possible states and storing a cost for each state. Heuristic search, in contrast, selectively explores a problem's state space. However, because DP algorithms successively approximate optimal evaluation functions, they are relevant to learning in a way that heuristic search is not. They effectively cache in a permanent data structure the results of repeated searches forward from each state. This information improves as the algorithm proceeds, ultimately converging to the optimal evaluation function, from which one can determine optimal policies with relative ease. Although some heuristic search algorithms (such as A*) update an estimate of the cost to reach states from an initial state, they typically do not update the heuristic evaluation function estimating the cost to reach a goal state from each state.

Although the principles of DP are relevant to learning, conventional DP algorithms are not really learning algorithms because they operate off-line. They are not designed to be applied *during* problem solving or control, whereas learning occurs as experience accumulates during actual (or simulated) attempts at problem solving or control. However, it is possible to execute an otherwise off-line DP algorithm concurrently with actual or simulated control, where the DP algorithm can influence, and can be influenced by, the ongoing control process. Doing this so as to satisfy certain requirements results in the algorithm we call RTDP, a special case of which essentially coincides with Korf's LRTA* [38] algorithm. This general approach follows previous research by others in which DP principles have been used for problem solving and learning (e.g., refs. [61; 69; 70; 81; 86; 87]).

Our contribution in this article has been to bring to bear on DP-based learning the theory of asynchronous DP as presented by Bertsekas and Tsitsiklis [12]. Although the suitability of asynchronous DP for implementation on multi-processor systems motivated this theory, we have made novel use of these results. Applying these results, especially the results on stochastic shortest path problems, to RTDP provides a new theoretical basis for DP-based learning algorithms. Convergence theorems for asynchronous DP imply that RTDP retains the competence of conventional synchronous and Gauss-Seidel DP algorithms, and the extension of Korf's LRTA* convergence theorem to this framework provides conditions under which RTDP avoids the exhaustive nature of off-line DP algorithms while still ultimately yielding optimal behavior.

We used the term *simulation mode* to refer to the execution of RTDP and related algorithms during simulated control instead of actual control. DP-based learning in simulation mode is illustrated by Samuel's checkers playing system [61; 62], Tesauro's backgammon playing system [77], and our illustrations of RTDP using the race track problem. Despite the fact that DP-based learning algorithms executed in simulation mode are actually off-line algorithms, we still treat them as learning algorithms because they incrementally improve control performance through simulated experience instead of solely through the application of more abstract computational methods. For algorithms, like RTDP, that require an accurate model of the decision problem, simulation mode is always an option and has obvious advantages due to the large number of trials often required. Applying RTDP during actual control makes sense when there is not enough time to compute a satisfactory policy by any off-line method before actual control must begin.

Whether applied during actual control or in simulation mode, RTDP can have significant advantages over conventional DP algorithms. Because RTDP is responsive to the demands of control in selecting states to which the backup operation is applied, it can focus computation onto parts of the state set for which control information is likely to be most important for improving control performance. The convergence theorem for Trial-Based RTDP applied to stochastic shortest path problems specifies conditions under which RTDP focuses on states that are on optimal paths—eventually abandoning all the other states—to produce a policy that is optimal on these relevant states without continuing to back up the costs of all the states, and possibly without backing up the costs of some states even once. Our illustrations using the race track problem show that RTDP can obtain near optimal policies in some problems with significantly less computation than is required by conventional DP.

However, more compelling is the fact that the approach illustrated by RTDP can form useful approximations to optimal evaluation functions in problems to which conventional DP cannot be feasibly applied at all. We mentioned, for example, that in backgammon, a single sweep of conventional DP would take more than 1,000 years using a 1,000 MIPS processor. This is true despite the fact that a large fraction of the states of backgammon are irrelevant in normal play.

RTDP is closely related to Monte Carlo algorithms that achieve computational efficiency by automatically allocating computation so that, for example, unimportant terms in a sum correspond to very rare events in the computational process [17]. For this reason, the asymptotic computational complexity of Monte Carlo methods can exceed that of other methods for some classes of problems. However, Monte Carlo methods are generally not competitive with deterministic methods for small problems or when high-precision answers are required. More research is needed to fully elucidate these correspondences and to exploit them in refining DP-based learning methods and understanding their computational complexity.

For problems that have very large states sets (such as backgammon), the lookup-table method for storing evaluation functions to which we have restricted attention is not practical. Much of the research on DP-based learning methods has made use of other storage schemes. For problems in which DP-based learning algorithms focus on increasingly small subsets of states, as illustrated in our simulations of the race track problem, data structures such as hash tables and kd-trees can allow the algorithms to perform well despite dramatically reduced space requirements. One can also adapt supervised learning procedures to use each back-up operation of a DP-based learning method to provide training information. If these methods can generalize adequately from the training data, they can provide efficient means for storing evaluation functions. Although some success has been achieved with methods that can generalize, such as connectionist networks, the theory we have presented in this article does not automatically extend to these cases. Generalization can disrupt the convergence of asynchronous DP. Additional research is needed to to understand how one can effectively combine function approximation methods with asynchronous DP.

In addition to the case in which an accurate model of the decision problem is available, we also devoted considerable attention to Markovian decision problems with incomplete information, i.e., problems for which an accurate model is not available. Adopting the terminology of the engineering literature, these problems require adaptive control methods. We described indirect and direct approaches to these problems. The method we called the *generic indirect method* is representative of the majority of algorithms described in the engineering literature applicable to Markovian decision problems with incomplete information. A system identification algorithm adjusts a system model on-line during control, and the controller selects actions based on a current estimate of the optimal evaluation function computed by a conventional DP algorithm under the assumption that the current model accurately models the system. The DP algorithm is re-executed whenever the system model is updated. Although this approach is theoretically convenient, it is much too costly to apply to large problems.

Adaptive RTDP results from substituting RTDP for conventional DP in the generic indirect method. This means that RTDP is executed using the most recent system model generated by the system identification algorithm. Adaptive RTDP can be tailored for the available computational resources by adjusting the number of DP stages it executes at each time step of control. Due to the additional uncertainty in this case, learning is necessarily slower than in the non-adaptive case when measured by the number of backups required. However, the amount of computation required to select each control action is roughly the same. This means that it is practical to apply Adaptive RTDP to problems that are much larger than those for which it is practical to apply methods, such as the generic indirect method, that re-execute a conventional DP algorithm whenever the system model is updated.

In addition to indirect adaptive methods, we discussed direct adaptive methods. Direct methods do not form explicit models of the system underlying the decision problem. We described Watkin's [81] Q-Learning algorithm, which approximates the optimal evaluation function without forming estimates of state-transition probabilities. Q-Learning instead uses sample state transitions, either generated by a system model or observed during actual control. Q-Learning is an asynchronous DP algorithm that operates at a finer grain than the asynchronous DP algorithm described in Section 5.3. Whereas the basic operation of asynchronous DP is backing up the cost of a state, requiring computation proportional to the number of possible successor states, the basic operation of Q-Learning is backing up the Q-value of a state-action pair, a computation that does not depend on the number of possible successor states. The fine grain of the basic Q-Learning backup allows Real-Time Q-Learning to focus on selected actions in addition to selected states in a way that is responsive to the behavior of the controlled system. The cost of this flexibility is the increased space required to store the Q-values of state-action pairs and the fact that a Q-Learning backup does not gather as much information as does a complete DP backup operation.

Sophisticated exploration strategies are important in solving Markovian decision problems under conditions of both complete and incomplete information. With complete information, a sophisticated exploration strategy can improve control performance by decreasing the time required to reach goal states or, in the case of RTDP, by focusing DP stages on states from which information most useful for improving the evaluation function is likely to be gained. Knowledgeable ordering of back ups can accelerate convergence of asynchronous DP, whether applied off- or on-line. When information is incomplete, sophisticated exploration is useful for other reasons as well. In this case, exploration strategies must also address the necessity to gather information about the unknown structure of the system being controlled. Unlike exploration in the case of complete information, which can be conducted in simulation mode, this kind of exploration must be conducted on-line. We discussed how exploration performed for this reason conflicts with the performance objective of control, at least on a short-term basis, and that a controller should not always execute actions that appear to be the best based on its current evaluation function.

Although we did not use sophisticated exploration strategies in our simulations of the race track problem, and we made no attempt in this article to analyse issues pertinent to exploration, sophisticated exploration strategies will play an essential role in making DP-based learning methods practical for larger problems. From what we did mention, however, it should be clear that it is not easy to devise a consistent set of desiderata for exploration strategies. For example, researchers have argued that an exploration strategy should 1) visit states in regions of the state space where information about the system is of low quality (to learn more about these regions), 2) visit states in regions of the state space where information about the system is of high quality (so that the back-up operation uses accurate estimates of the state-transition probabilities), or 3) visit states having successors whose costs are close to their optimal costs (so that the back-up operation efficiently propagates cost information). Each of these suggestions makes sense in the proper context, but it is not clear how to design a strategy that best incorporates all of them. It is encouraging, however, that the convergence results we have presented in this article are compatible with a wide range of exploration strategies.

Throughout this article we have assumed that the states of the system being controlled are completely and unambiguously observable by the controller. Although this assumption is critical to the theory and operation of all the algorithms we discussed, it can be very difficult to satisfy in practice. For example, the current state of a robot's world is vastly different from a list of the robot's current "sensations." On the positive side, effective closed-loop control policies do not have to distinguish between all possible sensations. However, exploiting this fact requires the ability to recognize states in the complex flow of sensations. Although the problem of state identification has been the subject of research in a variety of disciplines, and many approaches have been studied under many guises, it remains a critical factor in extending the applicability of DP-based learning methods. Any widely applicable approach to this problem must take the perspective that what constitutes a system's state for purposes of control—indeed what constitutes the system itself—is not independent of the control objectives. The framework adopted in this article in which "a dynamic system underlies the decision problem" is misleading in suggesting the existence of a single definitive grain with which to delineate events and to mark their passage. In actuality, control objectives dictate what is important in the flow of the controller's sensations, and multiple objective-dependent models at different levels of abstraction are needed to achieve them. If this caution is recognized, however, the algorithms described in this article should find wide application as components of sophisticated embedded systems.

12 Appendices

A. Proof of the Trial-Based RTDP Theorem

Here we prove Theorem 3, which extends Korf's [38] convergence theorem for LRTA* to Trial-Based RTDP applied to undiscounted stochastic shortest path problems.

Proof: We first prove the theorem for the special case in which only the cost of the current state is backed up at each time interval, i.e., $B_t = \{s_t\}$ and $k_t = t$, for $t = 0, 1, \ldots$ (see Section 6). We then observe that the proof does not change when each B_t is allowed to be an arbitrary set containing s_t. Let G denote the goal set and let s_t , u_t, and f_t respectively denote the state, action, and evaluation function at time step t in an arbitrary infinite sequence of states, actions, and evaluation functions generated by Trial-Based RTDP starting from an arbitrary start state.

First observe that the evaluation functions remain non-overestimating, i.e., at any time t, $f_t(i) \leq f^*(i)$ for all states i. This is true by induction because $f_{t+1}(i) = f_t(i)$ for all $i \neq s_t$ and if $f_t(j) \leq f^*(j)$ for all $j \in S$, then for all t

$$
\begin{aligned}
f_{t+1}(s_t) &= \min_{u \in U(i)} \left[c_{s_t}(u) + \sum_{j \in S} p_{s_t j}(u) f_t(j) \right] \\
&\leq \min_{u \in U(i)} \left[c_{s_t}(u) + \sum_{j \in S} p_{s_t j}(u) f^*(j) \right] = f^*(s_t),
\end{aligned}
$$

where the last equality restates the Bellman Optimality Equation (Equation 6).

Let $I \subseteq S$ be the set of all states that appear infinitely often in this arbitrary sequence; I must be nonempty because the state set is finite. Let $A(i) \in U(i)$ be the set of admissible actions for state i that have zero probability of causing a transition to a state not in I, i.e., $A(i)$ is the set of all actions $u \in U(i)$ such that $p_{ij}(u) = 0$ for all $j \in (S - I)$. Because states in $S - I$ appear a finite number of times, there is a finite time T_0 after which all states visited are in I. Then with probability one any action chosen an infinite number of times for any state i that occurs after T_0 must be in $A(i)$ (or else with probability one a transition out of I would occur), and so with probability one there must exist a time $T_1 \geq T_0$ such that for all $t > T_1$, we not only have that $s_t \in I$ but also that $u_t \in A(s_t)$.

We know that at each time step t, RTDP backs up the cost of s_t because $s_t \in B_t$. We can write the back-up operation as follows:

$$
f_{t+1}(s_t) = \min_{u \in U(i)} \left[c_{s_t}(u_t) + \sum_{j \in I} p_{s_t j}(u_t) f_t(j) + \sum_{j \in (S-I)} p_{s_t j}(u_t) f_t(j) \right]. \qquad (15)
$$

But for all $t > T_1$, we know that $s_t \in I$ and that $p_{s_t j}(u_t) = 0$ for all $j \in S - I$ because $u_t \in A(s_t)$. Thus, for $t > T_1$ the right-most summation in Equation 15 is zero. This means that the costs of the states in $S - I$ have no influence on the operation of RTDP after T_1. Thus, after T_1, RTDP performs asynchronous DP on a Markovian decision problem with state set I.

If no goal states are contained in I, then all the immediate costs in this Markovian decision problem are positive. Because there is no discounting, it can be shown that asynchronous DP must cause the costs of the states in I to grow without bound. But this contradicts the fact that the cost of a state can never overestimate its optimal cost, which must be finite due to the existence of a proper policy. Thus I contains a goal state with probability one.

After T_1, therefore, Trial-Based RTDP performs asynchronous DP on a stochastic shortest path problem with state set I that satisfies the conditions of the convergence theorem for asynchronous DP applied to undiscounted stochastic shortest path problems (Bertsekas and Tsitsiklis [12], Proposition 3.3, p. 318). Consequently, Trial-Based RTDP converges to the optimal evaluation function of this stochastic shortest path problem. We also know that the optimal evaluation function for this problem is identical to the optimal evaluation function for the original problem restricted to the states in I because the costs of the states in $S - I$ have no influence on the costs of states in I after time T_1.

Furthermore, with probability one I contains the set of all states reachable from any start state via any optimal policy. Clearly, I contains all the start states because each start state begins an infinite number of trails. Trial-Based RTDP always executes a greedy action with respect to the current evaluation function and breaks ties in such a way that it continues to execute all the greedy actions. Because we know that the number of policies is finite and that Trial-Based RTDP converges to the optimal evaluation function restricted to I, there is a time after which it continues to select all the actions that are greedy with respect to the optimal evaluation function, i.e., all the optimal actions. Thus with probability one, I contains all the states reachable from any start state via any optimal policy, and there is a time after which a controller using RTDP will only execute optimal actions.

Finally, with trivial revision the the above argument holds if RTDP backs up the costs of states other than the current state at each time step, i.e., if each B_t is an arbitrary subset of S.
Q.E.D.

B. Simulation Details

Except for the discount factor γ, which we set to one throughout the simulations, and the sets B_t, which we set to $\{s_t\}$ for all t, RTDP does not involve any parameters. Gauss-Seidel DP only requires specifying a state ordering for its sweeps. We selected an ordering without concern for any influence it might have on convergence rate. Both Adaptive RTDP and Real-

Time Q-Learning require exploration during the training trials, which we implemented using Equation 11. To generate the data described in Section 4.1, we decreased the parameter T with successive moves as follows:

$$T(0) = T_{\text{Max}} \tag{16}$$

$$T(k+1) = T_{\text{Min}} + \beta(T(k) - T_{\text{Min}}), \tag{17}$$

where k is the move number (cumulative over trials), $\beta = 0.992$, $T_{\text{Max}} = 75$, and $T_{\text{Min}} = 0.5$.

Real-time Q-Learning additionally requires sequences of learning rate parameters $\alpha_t(i,u)$ (Equation 14) that satisfy the hypotheses of the Q-Learning convergence theorem [81; 82]. We defined these sequences as follows. Let $\alpha_t(i,u)$ denote the learning rate parameter used when the Q-value of the state-action pair (i,u) is backed up at time step t. Let $n_t(i,u)$ be the number of backups performed on the Q-value of (i,u) up to time step t. The learning rate $\alpha_t(i,u)$ is defined as follows:

$$\alpha_t(i,u) = \frac{\alpha_0 \tau}{\tau + n_t(i,u)}$$

where α_0 is the initial learning rate. We set $\alpha_0 = 0.5$ and $\tau = 300$. This equation implements a *search-then-converge* schedule for each $\alpha_t(i,u)$ as suggested by Darken and Moody [19]. They argue that such schedules can achieve good performance in stochastic optimization tasks. It can be shown that this schedule satisfies the hypotheses of the Q-Learning convergence theorem.

References

[1] C. W. Anderson. Strategy learning with multilayer connectionist representations. Technical Report TR87-509.3, GTE Laboratories, Incorporated, Waltham, MA, 1987. (This is a corrected version of the report published in *Proceedings of the Fourth International Workshop on Machine Learning*,103–114, 1987, San Mateo, CA: Morgan Kaufmann.).

[2] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, MA, 1988.

[3] A. G. Barto, R. S. Sutton, and C. Watkins. Sequential decision problems and neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 686–693, San Mateo, CA, 1990. Morgan Kaufmann.

[4] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. In M. Gabriel and J. Moore, editors, *Learning and Computational Neuroscience:Foundations of Adaptive Networks*, pages 539–602. MIT Press, Cambridge, MA, 1990.

[5] A.G. Barto. Reinforcement learning and adaptive critic methods. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 469–491. Van Nostrand Reinhold, New York, 1992.

[6] A.G. Barto and S.P. Singh. On the computational economics of reinforcement learning. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Connectionist Models: Proceedings of the 1990 Summer School*, pages 35–44. Morgan Kaufmann, San Mateo, CA, 1991.

[7] R. Bellman and S. E. Dreyfus. Functional approximations and dynamic programming. *Math Tables and Other Aides to Computation*, 13:247–251, 1959.

[8] R. Bellman, R. Kalaba, and B. Kotkin. Polynomial approximation—A new computational technique in dynamic programming: Allocation processes. *Mathematical Computation*, 17:155–161, 1973.

[9] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[10] D. P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27:610–616, 1982.

[11] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[12] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[13] S. J. Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing 4*, San Mateo, CA. Morgan Kaufmann. To appear.

[14] D. Chapman. Penquins can make cake. *AI Magazine*, 10:45–50, 1989.

[15] D. Chapman and L. P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, 1991.

[16] J. Christensen and R. E. Korf. A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence AAAI-86*, pages 148–152, San Mateo, CA, 1986. Morgan Kaufmann.

[17] J. H. Curtiss. A theoretical comparison of the efficiencies of two classical methods and a monte carlo method for computing one component of the solution of a set of linear algebraic equations. In H. A. Meyer, editor, *Symposium on Monte Carlo Methods*, pages 191–233. Wiley, New York, 1954.

[18] J. W. Daniel. Splines and efficiency in dynamic programming. *Journal of Mathematical Analysis and Applications*, 54:402–407, 1976.

[19] C. Darken and J. Moody. Note on learning rate schedule for stochastic optimization. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 832–838, San Mateo, CA, 1991. Morgan Kaufmann.

[20] P. Dayan. Navigating through temporal difference. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 464–470, San Mateo, CA, 1991. Morgan Kaufmann.

[21] P. Dayan. *Reinforcing Connectionism: Learning the Statistical Way.* PhD thesis, University of Edinburgh, 1991.

[22] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8:341–362, 1992.

[23] T. L. Dean and M. P. Wellman. *Planning and Control.* Morgan Kaufmann, San Mateo, CA, 1991.

[24] E. V. Denardo. Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9:165–177, 1967.

[25] M. Gardner. Mathematical games. *Scientific American*, 228:108, January 1973.

[26] D. Gelperin. On the optimality of A*. *Artificial Intelligence*, 8:69–76, 1977.

[27] M. L. Ginsberg. Universal planning: An (almost) universally bad idea. *AI Magazine*, 10:40–44, 1989.

[28] S. E. Hampson. *Connectionist Problem Solving: Computational Aspects of Biological Learning.* Birkhauser, Boston, 1989.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4:100–107, 1968.

[30] J. H. Holland. Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, pages 593–623. Morgan Kaufmann, San Mateo, CA, 1986.

[31] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, New York, 1970.

[32] A. Jalali and M. Ferguson. Computationally efficient adaptive control algorithms for markov chains. In *Proceedings of the 28th Conference on Decision and Control*, pages 1283–1288, Tampa, Florida, 1989.

[33] M. I. Jordan and R. A. Jacobs. Learning to control an unstable system with forward modeling. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufmann.

[34] L. P. Kaelbling. *Learning in Embedded Systems*. MIT Press, Cambridge, MA, 1991. Revised version of Teleos Research TR-90-04, June 1990.

[35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[36] A. H. Klopf. Brain function and adaptive systems—A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA, 1972. A summary appears in *Proceedings of the International Conference on Systems, Man, and Cybernetics*, 1974, IEEE Systems, Man, and Cybernetics Society, Dallas, TX.

[37] A. H. Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemishere, Washington, D.C., 1982.

[38] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[39] P. R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329–380, 1985.

[40] V. Kumar and L. N. Kanal. The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In L. N. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 1–37. Springer-Verlag, 1988.

[41] H. J. Kushner and P. Dupuis. *Numerical Methods for Stochastic Control Problems in Continuous Time*. Springer-Verlag, New York, 1992.

[42] W. H. Kwon and A. E. Pearson. A modified quadratic cost problem and feedback stabilization of a linear system. *IEEE Transactions on Automatic Control*, 22:838–842, 1977.

[43] Y. le Cun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 21–28. Morgan Kaufmann, San Mateo, CA, 1988.

[44] M. Lemmon. Real-time optimal path planning using a distributed computing paradigm. In *Proceedings of the American Control Conference*, Boston, MA, 1991.

[45] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786, Cambridge, MA, 1991. MIT Press.

[46] Long-Ji Lin. Self-improvement based on reinforcement learning, planning and teaching. In L. A. Birnbaum and G. C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 323–327, San Mateo, CA, 1991. Morgan Kaufmann.

[47] Long-Ji Lin. Self-improving reactive agents: Case studies of reinforcement learning frameworks. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 297–305, Cambridge, MA, 1991. MIT Press.

[48] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[49] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.

[50] D. Q. Mayne and H. Michalska. Receding horizon control of nonlinear systems. *IEEE Transactions on Automatic Control*, 35:814–824, 1990.

[51] L. Mérõ. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23:13–27, 1984.

[52] D. Michie and R. A. Chambers. BOXES: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137–152. Oliver and Boyd, 1968.

[53] M. L. Minsky. *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*. PhD thesis, Princeton University, 1954.

[54] M. L. Minsky. Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8–30, 1961. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 406–450, 1963.

[55] A. W. Moore. *Efficient Memory-Based Learning for Robot Control.* PhD thesis, University of Cambridge, Cambridge, UK, 1990.

[56] A. W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In L. A. Birnbaum and G. C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 333–337, San Mateo, CA, 1991. Morgan Kaufmann.

[57] A. W. Moore and C. G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances in Neural Information Processing 4*, San Mateo, CA. Morgan Kaufmann. To appear.

[58] J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, Honolulu, HI. To appear.

[59] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137, 1978.

[60] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, 1983.

[61] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, pages 210–229, 1959. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, McGraw-Hill, New York, 1963.

[62] A. L. Samuel. Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal on Research and Development*, pages 601–617, November 1967.

[63] J. Schmidhuber. Adaptive confidence and adaptive curiosity. Technical Report FKI-149-91, Institut für Informatik, Technische Universität München, Arcisstr. 21, 800 München 2, Germany, 1991.

[64] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Menlo Park, CA, 1987.

[65] M. J. Schoppers. In defense of reaction plans as caches. *AI Magazine*, 10:51–60, 1989.

[66] S. P. Singh and R. C. Yee. An upper bound on the loss from approximate optimal value functions. Technical report. Submitted as Technical Note to *Machine Learning.*

[67] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Amherst, MA, 1984.

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

[69] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, San Mateo, CA, 1990. Morgan Kaufmann.

[70] R. S. Sutton. Planning by incremental dynamic programming. In L. A. Birnbaum and G. C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 353–357, San Mateo, CA, 1991. Morgan Kaufmann.

[71] R. S. Sutton, editor. *A Special Issue of Machine Learning on Reinforcement Learning*, volume 8. *Machine Learning*, 1992. Also published as *Reinforcement Learnng*, Kluwer Academic Press, Boston, MA 1992.

[72] R. S. Sutton and A. G. Barto. Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88:135–170, 1981.

[73] R. S. Sutton and A. G. Barto. A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ, 1987. Erlbaum.

[74] R. S. Sutton and A. G. Barto. Time-derivative models of pavlovian reinforcement. In M. Gabriel and J. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497–537. MIT Press, Cambridge, MA, 1990.

[75] R. S. Sutton, A. G. Barto, and R. J. Williams. Reinforcement learning is direct adaptive optimal control. In *Proceedings of the American Control Conference*, pages 2143–2146, Boston, MA, 1991.

[76] M. Tan. Learning a cost-sensitive internal representation for reinforcement learning. In L. A. Birnbaum and G. C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 358–362, San Mateo, CA, 1991. Morgan Kaufmann.

[77] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[78] S. B. Thrun and K. Möller. Active exploration in dynamic environments. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, San Mateo, CA, 1992. Morgan Kaufmann.

[79] S.B. Thrun. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold, New York, 1992.

[80] P. E. Utgoff and J. A. Clouse. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth Annual Conference on Artificial Intelligence*, pages 596–600, San Mateo, CA, 1991. Morgan Kaufmann.

[81] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, 1989.

[82] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[83] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[84] P. J. Werbos. Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22:25–38, 1977.

[85] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kosin, editors, *System Modeling an Optimization*. Springer-Verlag, 1982. Proceedings of the Tenth IFIP Conference, New York, 1981.

[86] P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 1987.

[87] P. J. Werbos. Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[88] P.J. Werbos. Approximate dynamic programming for real-time control and neural modeling. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 493–525. Van Nostrand Reinhold, New York, 1992.

[89] D.A. White and M.I. Jordan. Optimal control: A foundation for intelligent control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 185–214. Van Nostrand Reinhold, New York, 1992.

[90] S. D. Whitehead. Complexity and cooperation in Q-learning. In L. A. Birnbaum and G. C. Collins, editors, *Maching Learning: Proceedings of the Eighth International Workshop*, pages 363–367, San Mateo, CA, 1991. Morgan Kaufmann.

[91] R. J. Williams and L. C. Baird, III. A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pages 96–101, New Haven, CT, Aug 1990.

[92] I. H. Witten. An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, 34:286–295, 1977.

[93] I. H. Witten. Exploring, modelling and controlling discrete sequential environments. *International Journal of Man-Machine Studies*, 9:715–735, 1977.

[94] R. C. Yee. Abstraction in control learning. Technical Report 92-16, Department of Computer Science, University of Massachusetts, Amherst, MA, 1992.