Learning Evaluation Functions for Large Acyclic Domains

Justin A. Boyan and **Andrew W. Moore**
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA
jab@cs.cmu.edu, awm@cs.cmu.edu

Abstract

Some of the most successful recent applications of reinforcement learning have used neural networks and the TD(λ) algorithm to learn evaluation functions. In this paper, we examine the intuition that TD(λ) operates by approximating asynchronous value iteration. We note that on the important subclass of acyclic tasks, value iteration is inefficient compared with another graph algorithm, DAG-SP, which assigns values to states by working strictly backwards from the goal. We then present ROUT, an algorithm analogous to DAG-SP that can be used in large stochastic state spaces requiring function approximation. We close by comparing the behavior of ROUT and TD on a simple example domain and on two domains with much larger state spaces.

1 LEARNING CONTROL BACKWARDS

Computing an accurate *value function* is the key to dynamic-programming-based algorithms for optimal sequential control in Markov Decision Processes. The optimal value function $V^*(x)$ specifies, for each state x in the state space X, the expected cumulative reward when starting in state x and acting optimally thereafter. It is also the unique solution to the *Bellman equations*: (using the notation of [Watkins and Dayan, 1992]) $\forall x \in X$,

$$V(x) = \begin{cases} R(x) & \text{if } x \text{ is a terminal state} \\ \max_{a \in A(x)} \left(R(x,a) + \gamma \sum_{y \in X} \text{Prob}(x \xrightarrow{a} y) V(y) \right) & \text{otherwise.} \end{cases} \tag{1}$$

The Bellman equation at x also reveals the optimal control from x: any action which instantiates the max is an optimal choice [Bellman, 1957].

For small discrete problems, the value function can be stored in a lookup table and computed by iterative algorithms such as *value iteration* (VI) [Bellman, 1957]. VI computes V^* by repeatedly sweeping over the state space, applying Equation 1 as an assignment statement (this is called a "one-step backup") at each state in parallel. If the lookup table is initialized with all 0's, then after i sweeps of VI, the table will represent the maximum expected return of a path of length i from each state. For certain goal-oriented domains, this corresponds to the intuition that VI works by propagating correct V^* values backwards, by one step per iteration, from the terminal states.

More precisely, there are two classes of MDPs for which correct V^* values can be assigned by working strictly backwards from terminal states:

1. *deterministic* domains with no positive-reward cycles and with every state able to reach at least one terminal state. This class includes shortest-path and minimum cost-to-go problems.

2. (possibly stochastic) *acyclic* domains: domains where no legal trajectory can pass through the same state twice. Many problems naturally have this property (e.g. games like tic-tac-toe and Connect-Four; one formulation of the job-shop

scheduling domain [Zhang and Dietterich, 1995]; any finite-horizon problem for which time is a component of the state).

Using VI to solve MDPs belonging to either of these special classes can be quite inefficient, since VI performs backups over the entire space, whereas the only backups useful for improving V^* are those on the "frontier" between already-correct and not-yet-correct V^* values. In fact, there are classical algorithms for both problem classes which compute V^* more efficiently by explicitly working backwards: for the deterministic class, Dijkstra's shortest-path algorithm; and for the acyclic class, DIRECTED-ACYCLIC-GRAPH-SHORTEST-PATHS (DAG-SP) [Cormen *et al.*, 1990].[1] DAG-SP first topologically sorts the MDP, producing a linear ordering of the states in which every state x precedes all states reachable from x. Then, it runs through that list in reverse, performing one backup per state. Worst-case bounds for VI, Dijkstra, and DAG-SP in deterministic domains with X states and A actions/state are $O(AX^2)$, $O(AX \log X)$, and $O(AX)$, respectively.

Another difference between VI and working backwards is that VI repeatedly re-estimates the values at every state, using old predictions to generate new training values. By contrast, Dijkstra and DAG-SP are always explicitly aware of which states have their V^* values already known, and can hold those values fixed. This will be important when we introduce generalization and the possibility of approximation error.

VALUE FUNCTION APPROXIMATION

The VI, Dijkstra and DAG-SP algorithms all apply exclusively to MDPs for which the state space can be exhaustively enumerated and the value function represented as a lookup table. For the high-dimensional state spaces characteristic of real-world control tasks, such enumeration is intractable. Computing V^* requires generalization: a natural technique is to encode the states as real-valued feature vectors and to use a function approximator to fit V^* over this feature space.

Perhaps the most successful application of VI-based algorithms with function approximation has been in the domain of backgammon [Tesauro, 1992]. Tesauro modified Sutton's TD(λ) algorithm [Sutton, 1988], which is normally thought of as a model-free algorithm for learning to predict, into a model-based algorithm for learning to control. Table 1 shows a variant of Tesauro's algorithm adapted for the acyclic MDP case. When $\lambda = 0$, this algorithm becomes the RTDP algorithm [Barto *et al.*, 1995], which is closely related to VI; the key difference is that its backups are done along sample trajectories through the process, rather than along sweeps of the entire state space.

Tesauro's combination of TD(λ) and neural networks has been applied successfully to other domains, including combinatorial optimization [Zhang and Dietterich, 1995] and elevator control [Crites and Barto, 1996]. Nevertheless, it is important to note that when function approximators are used, TD(λ) provides no guarantees of optimality. In the case of uncontrolled Markov chains and linear function approximators, online TD(λ) does converge [Dayan, 1992, Tsitsiklis and Van Roy, 1996]—but even then, if $\lambda \neq 1$, convergence is not necessarily to a good approximation of V^* [Bertsekas, 1995]. Moreover, in the general case of arbitrary function approximators and controlled Markov processes, repeatedly applying one-step backups may propagate and enlarge approximation errors, leading to instability [Boyan and Moore, 1995, Gordon, 1995].

Thus, we have presented two reasons why working strictly backwards may be desirable: efficiency, because updates need only be done on the "frontier" rather than all over state space; and robustness, because correct V^* values, once assigned, need never again be changed. We have therefore investigated generalizations of the Dijkstra and DAG-SP algorithms specifically modified to accommodate huge state spaces and value function approximation. Our variant of Dijkstra's algorithm, called Grow-Support, was presented in [Boyan and Moore, 1995] and will not be discussed further here. Our variant of DAG-SP is a very different algorithm we call ROUT, introduced below. Table 2 summarizes the relationships among these algorithms.

2 THE "ROUT" ALGORITHM

In the huge domains for which ROUT is designed, DAG-SP's key preprocessing step—topologically sorting the entire state space—is no longer tractable. Instead, ROUT must expend some extra effort to identify states on the current frontier. Once identified (as described below), a frontier state is assigned its optimal V^* value by a simple one-step backup, and this {state→value} pair is added to a training set for a

[1] Although [Cormen *et al.*, 1990] presents DAG-SP only for deterministic acyclic problems, it applies straightforwardly to the stochastic case.

TD(λ, start states $\hat{X}$, fitter F):
/* *Assumes known world model MDP; F is parametrized by weight vector w.* */
repeat steps 1 and 2 forever:
1. Using the model and the current evaluation function F, generate a mostly-greedy trajectory from a start state to a terminal state: $x_0 \to x_1 \cdots \to x_T$.
   Also record the rewards $r_0, r_1, \ldots r_T$ received at each step.
2. Update the fitter from the trajectory as follows:
   for i := T downto 0, do:
   $$\text{targ}_i := \begin{cases} r_T \text{ (the terminal reward)} & \text{if } i = T \\ r_i + \lambda \cdot \text{targ}_{i+1} + (1-\lambda) F(x_{i+1}) & \text{otherwise} \end{cases}$$
   update F's weights by delta rule: $\Delta w := \alpha(\text{targ}_i - F(x_i)) \nabla_w F(x_i)$;
   end

Table 1: TD(λ) for learning V^* from an acyclic MDP

Alg. for lookup-table V^*	Applicable MDPs	Alg. for fun.approx. V^*
Value Iteration	arbitrary	TD(λ)
Dijkstra	deterministic	Grow-Support
DAG-SP	acyclic	ROUT

Table 2: Algorithms for generating optimal value functions

function approximator. Thus, ROUT's main loop consists of identifying a frontier state; determining its V^* value; and retraining the approximator (see Table 3). The training set, constructed adaptively, grows backwards from the goal.

ROUT's key subroutine, HuntFrontierState, is responsible for identifying a good state x to add to the training set. In particular:

1. All states reachable from x should already have their V^* values correctly approximated by the function approximator. This ensures that the policy from x onward is optimal, and that a correct target value for $V^*(x)$ can be assigned.

2. x itself should *not* already have its V^* value correctly approximated. This condition aims to keep the training set as small as possible, by excluding states whose values are correct anyway thanks to good generalization.

3. x should be a state that we care to learn about. For that reason, ROUT considers only states which occur on trajectories emanating from one of a set of problem-specific "start states."

The HuntFrontierState operation returns a state which with high probability satisfies these properties. It works by generating a number of trajectories from x, each time checking to see whether all states along the trajectory are self-consistent (i.e., satisfy Equation 1 to some tolerance ϵ). If all states after x on all sample trajectories are self-consistent, then x is deemed ready, and ROUT will add x to its training set. If, on the other hand, a trajectory from x reveals any inconsistencies in the approximated value function, then we flag that trajectory's *last* such inconsistent state, and restart HuntFrontierState from there. Figure 1 illustrates how the routine works.

The parameters of the ROUT algorithm are H, the number of trajectories generated to certify a state's readiness, and ϵ, the tolerated Bellman residual. ROUT's convergence to the optimal V^*, assuming the function approximator can fit the V^* training set perfectly, can be guaranteed in the limiting case where $H \to \infty$ (assuring exploration of all states reachable from x) and $\epsilon = 0$. In practice, of course, we want to be tolerant of some approximation error. Typical settings we used were $H = 20$ and $\epsilon = 0.1$ (or roughly 5% of the range of V^*).

3 RESULTS

We present here results with ROUT on three domains: a prediction task, a two-player dice game, and a k-

ROUT(start states $\hat{X}$, fitter F):
    /* *Assumes that the world model MDP is known and acyclic.* */
    initialize training set $S := \emptyset$, and $F :=$ an arbitrary fit;
    repeat:
        for each start state $x \in \hat{X}$ not yet marked "done", do:
            $s :=$ HuntFrontierState(x, F);
            add $\{s \mapsto$ one-step-backup(s)$\}$ to training set S and re-train fitter F on S;
            if $(s = x)$, then mark start state x as "done".
    until all start states in $\hat{X}$ are marked "done".

HuntFrontierState(state x, fit F):
    /* *If the value function is self-consistent on all trajectories from x, return x. (That is determined probabilistically by Monte Carlo trials.) Otherwise, return a state on a trajectory from x for which the self-consistency property* **is** *true.* */
    for each legal action $a \in A(x)$, do:
        repeat up to H times:
            generate a trajectory $\vec{T}$ from x to termination, starting with action a;
            let y be the *last* state on $\vec{T}$ with Bellman residual $> \epsilon$;
            if $(y \neq \emptyset)$ and $(y \neq x)$, then break out of loops, and
                restart procedure with HuntFrontierState(y, F).
    /* *reaching this point, x's subtree is deemed all self-consistent and correct!* */
    return x.

Table 3: The ROUT main loop and HuntFrontierState subroutine

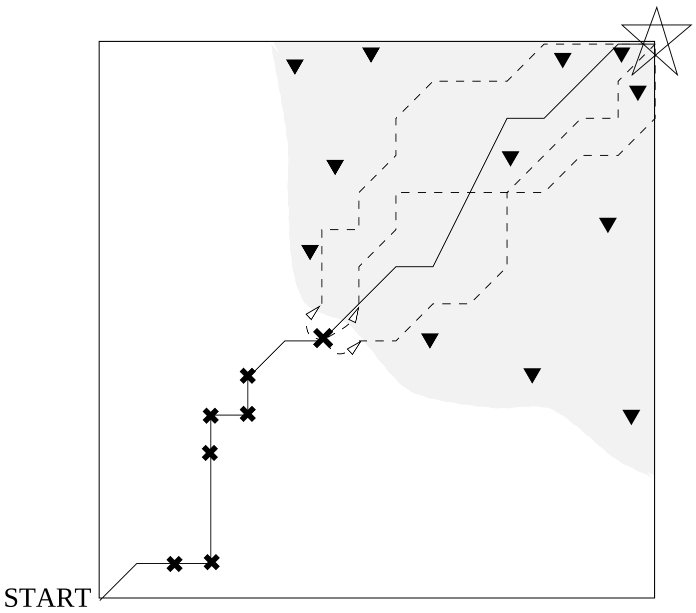

Figure 1: A schematic of ROUT working on an acyclic two-dimensional navigation domain, where the allowable actions are only $\rightarrow$, $\nearrow$, and $\uparrow$. Suppose that ROUT has thus far established training values for V^* at the triangles, and that the function approximator has successfully generalized V^* throughout the shaded region. Now, when HuntFrontierState generates a trajectory from the start state to termination (solid line), it finds that several states along that trajectory are inconsistent (marked by crosses). The last such cross becomes the new starting point for HuntFrontierState. From there, all trajectories generated (dashed lines) are fully self-consistent, so that state gets added to ROUT's training set. When the function approximator is re-trained, the shaded region of validity should grow, backwards from the goal.

armed bandit problem. For all problems, we compare ROUT's performance with that of TD(λ) given the equivalent function approximator. We measure the time to reach best performance (in terms of total number of state evaluations performed) and the quality of the learned value function (in terms of Bellman residual, closeness to the true V^*, and performance of the greedy control policy).

Task 1: Hopworld

The "Hopworld" is a small domain designed to illustrate how ROUT combines working backwards, adaptive sampling and function approximation. The domain is an acyclic Markov chain of 13 states in which each state has two equally probable successors: one step to the right or two steps to the right. The transition rewards are such that for each state $V^*(n) = -2n$. Our function approximator F makes predictions by interpolating between values at every fourth state. This is equivalent to using a linear approximator over the four-element feature vector representation depicted in Figure 2.

In ROUT, we fit the training set using a batch least-squares fit. In TD, the coefficients are updated using the delta rule with a hand-tuned learning rate. The results are shown in Table 4. ROUT's performance is efficient and predictable on this contrived problem. At the start, HUNTFRONTIERSTATE finds F is inconsistent and trains $F(1)$ and $F(2)$ to be -2 and -4, respectively. Linear extrapolation then forces states 3 and 4 to be correct. On the third iteration, $F(5)$ is spotted as inconsistent and added to the training set, and beneficial extrapolation continues. By comparison, TD also has no trouble learning V^*, but requires many more evaluations. This is because TD trains blindly on all transitions, not only the useful ones; and because its updates must be done with a fairly small learning rate, since the domain is stochastic. TD could be improved by an adaptive learning rate, but even the most baroque scheme for adaptation would have a hard time making the direct least-squares fits that ROUT is able to do.

Task 2: The Game of Pig

"Pig" is a two-player children's dice game. Each player starts with a total score of zero, which is increased on each turn by dice rolling. The first to 100 wins. On her turn, a player accumulates a subtotal by repeatedly rolling a 6-sided die. If at any time she rolls a 1, however, she loses the subtotal and gets only 1 added to her total. Thus, before each roll, she must decide whether to (a) add her currently-accumulated subtotal to her permanent total and pass the turn to the other player; or (b) continue rolling, risking an unlucky 1.

Pig belongs to the class of symmetric, alternating, Markov games. This means that the minimax-optimal value function can be formulated as the unique solution to a system of Bellman equations like Equation 1.[2] The state space, with two-player symmetry factored out, has 515,000 positions—large enough to be interesting, but small enough that computing the exact V^* is tractable.

For input to the function approximator, we represent states by their natural 3-dimensional feature representation: X's total, O's total, and X's current subtotal. The approximator is a standard MLP with two hidden units. In ROUT, the network is retrained to convergence (at most 1000 epochs) each time the training set is augmented. Note that this extra cost of ROUT is not reflected in the results table, but for practical applications, a far faster approximator than backprop would be used with ROUT.[3]

The Pig results are charted in Table 4 and graphed in Figure 3. The graph shows the learning curves for the best single trial of each of six classes of runs: TD(0), TD(0.8) and TD(1), with and without exploration. The best TD run, TD(0) with exploration, required about 30 million evaluations to reach its best performance of about -0.15. By contrast, ROUT completed successfully in under 1 million evaluations, and performed at the significantly higher level of -0.09. ROUT's adaptively-generated training set contained only 133 states.

Task 3: Multi-armed Bandit Problem

Our third test problem is to compute the optimal policy for a finite-horizon k-armed bandit [Berry and

[2]The only difference is that some of the "probabilities" $\mathrm{Prob}(x \xrightarrow{a} y)$ will be negative, reflecting the minimax nature of the game. Some MDP-solving methods (e.g. linear programming) can no longer be used for this class of problems; however, VI and DAG-SP do still apply, as do their function-approximation counterparts, TD and ROUT [Littman and Szepesvári, 1996].

[3]Unlike TD, which works only with parametric function approximators for which $\nabla_w F(x)$ can be calculated, ROUT can work with arbitrary function approximators, including batch methods such as projection-pursuit and locally weighted regression. For this paper's comparative experiments, however, we used linear or neural net fits for both algorithms.

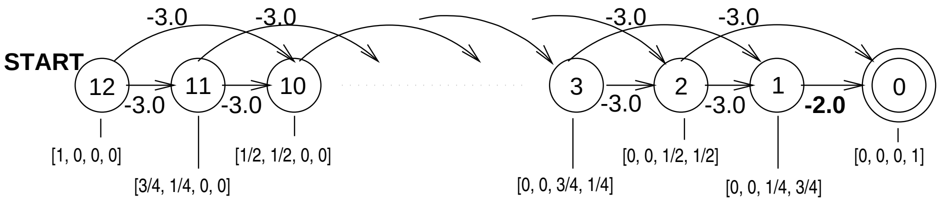

Figure 2: The Hopworld Markov chain. Each state is represented by a four-element feature vector as shown. The function approximator is linear.

Problem	Method	# training samples	total evaluations	RMS Bellman	RMS $\|\|V^* - F\|\|$	Policy Quality
HOP	Discrete*	12	21	0	0	-24 *
	ROUT	4	158	0.	0.	-24
	TD(0)	5000	10,000	0.03	0.1	-24
	TD(1)	5000	10,000	0.03	0.1	-24
PIG	Discrete*	515,000	3.6M	0	0	0 *
	ROUT	133	0.8M	0.09	0.14	-0.093
	TD(0) + explore	5 M	30 M	0.23	0.29	-0.151
	TD(0.8) + explore	9 M	60 M	0.23	0.33	-0.228
	TD(1) + explore	6 M	40 M	0.22	0.30	-0.264
	TD(0) no explore	8+ M	50+ M	0.12	0.54	-0.717
	TD(0.8) no explore	5 M	35 M	0.33	0.44	-0.308
	TD(1) no explore	5 M	30 M	0.23	0.32	-0.186
BAND	Discrete*	736,281	4 M	0	0	0.682 *
	ROUT	30	15,850	0.01	0.05	0.668
	TD(0)	150,000	900,000	0.07	0.14	0.666
	TD(1)	100,000	600,000	0.02	0.04	0.669

Table 4: Summary of results. For each algorithm on each problem, we list two measurements of time to quiescence followed by three measurements of the solution quality. The measurements for TD were taken at the time when, roughly, best performance was first consistently reached. (Key: M=10^6; * denotes optimal performance for each task.)

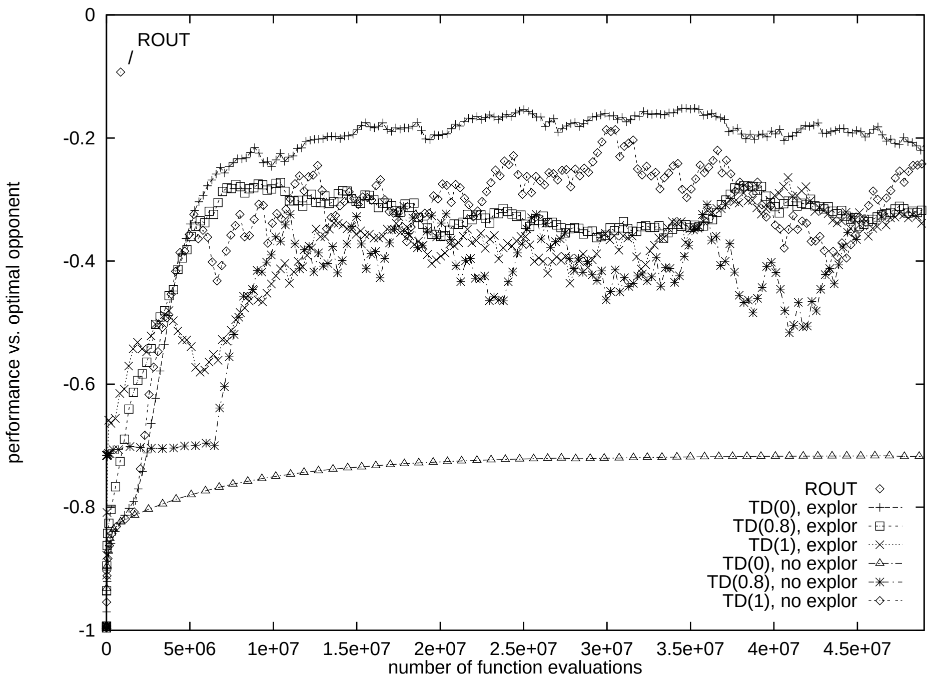

Figure 3: Performance of Pig policies learned by TD and ROUT.
ROUT's performance is marked by a single diamond at the top left of the graph.

Fristedt, 1985, Duff, 1995]. While an optimal solution in the infinite-horizon case can be found efficiently using Gittins indices, solving the finite-horizon problem is equivalent to solving a large acyclic, stochastic MDP in belief space [Berry and Fristedt, 1985]. We show results for $k = 3$ arms and a horizon of $n = 25$ pulls, where the resulting MDP has 736,281 states. Solving this MDP by DAG-SP produces the optimal exploration policy, which has an expected reward of 0.6821 per pull.

We encoded each state as a six-dimensional feature vector of [$\#\text{succ}_{arm1}$, $\#\text{fail}_{arm1}$, $\#\text{succ}_{arm2}$, $\#\text{fail}_{arm2}$, $\#\text{succ}_{arm3}$, $\#\text{fail}_{arm3}$] and attempted to learn a neural network approximation to V^* with TD(0), TD(1), and ROUT. Again, the parameters for all algorithms were tuned by hand.

The results are shown in Table 4. All methods do spectacularly well, although the TD methods again require more trajectories and more evaluations. Careful inspection of the problem reveals that a globally linear value function, extrapolated from the states close to the end, has low Bellman residual and performs very nearly optimally. Both ROUT and TD successfully exploit this linearity.

DISCUSSION

When a function approximator is capable of fitting V^*, ROUT will, in the limit, find it. However, for ROUT to be efficient, the frontier must grow backward from the goal quickly; and this depends on good extrapolation from the training set. When good extrapolation does not occur, ROUT may become stuck, repeatedly adding points near the goal region and never progressing backwards. One possible solution to this would be to adapt ROUT's tolerance level ϵ, thereby guaranteeing progress at the expense of accuracy. Another possibility would be to investigate the use of function approximators especially well-suited to extrapolation from noiseless training data.

Still, in some cases the approximator may not even be adequate for fitting V^* at all. In this case, we wish to find the best *evaluation function*—a fittable function that produces a good policy, regardless of the Bellman equations. At that point all our intuitions about how to derive the best function break down, and the behaviors of ROUT, TD(0) and even TD(1) become ill-understood. For example, in preliminary experiments on the game of Connect-4, we found that ROUT was unable to represent V^* near the goal region and became stuck, whereas TD learned to play well despite the approximator's inadequacy. Understanding how TD manages this is an important open question for reinforcement learning.

4 CONCLUSIONS

Graph-theoretic algorithms which work backwards from the goal have been important for efficiency in many areas of computer science (planning, Grassfire algorithm, endgame databases), so it is natural to ask whether they can similarly benefit function-approximation-based methods for learning control. The ROUT algorithm addresses this question. A key consideration was to avoid sampling all states in order to work backwards; the HUNTFRONTIERSTATE method provides a basis for sampling adaptively. On our test domains, ROUT learned excellent evaluation functions using much less data than TD(λ), and we believe it indicates many interesting directions for future research.

Acknowledgments

We would like to thank Michael Littman for many helpful discussions, and the anonymous reviewers for excellent feedback. We also acknowledge the support of the NDSEG/ONR fellowship program and NSF Grant IRI-9214873.

References

[Barto *et al.*, 1995] A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time learning and control using asynchronous dynamic programming. *AI Journal*, 1995.

[Bellman, 1957] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Berry and Fristedt, 1985] D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, 1985.

[Bertsekas, 1995] D. Bertsekas. A counterexample to temporal differences learning. *Neural Computation*, 7:270–9, 1995.

[Boyan and Moore, 1995] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*. MIT Press, 1995.

[Cormen *et al.*, 1990] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Crites and Barto, 1996] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselno, editors, *Advances in Neural Information Processing Systems 8*, 1996.

[Dayan, 1992] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8(3/4), May 1992.

[Duff, 1995] M. O. Duff. Q-learning for bandit problems. Technical Report CMPSCI 95-26, University of Massachusetts, 1995.

[Gordon, 1995] G. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufmann, 1995.

[Littman and Szepesvári, 1996] M. L. Littman and C. Szepesvári. A generalized reinforcement-learning model: Convergence and applications. In L. Saitta, editor, *Machine Learning: Proceedings Of The Thirteenth International Conference (this volume)*. Morgan Kaufmann, 1996.

[Sutton, 1988] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 1988.

[Tesauro, 1992] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4), May 1992.

[Tsitsiklis and Van Roy, 1996] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, MIT, 1996.

[Watkins and Dayan, 1992] C. Watkins and P. Dayan. Technical note: Q-Learning. *Machine Learning*, 8(3/4), May 1992.

[Zhang and Dietterich, 1995] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of IJCAI-95*, pages 1114–1120, 1995.