Chapter 6

STAGE: Extensions

The preceding chapters have demonstrated the effectiveness of a simple idea: learned predictions of search outcomes can be used to improve future search outcomes. In this chapter, I consider two independent extensions to the basic STAGE algorithm:

- Section 6.1 investigates the use of the TD(λ) family of algorithms, including a new least-squares formulation, for making more efficient use of memory and data while learning V^π.
- Section 6.2 illustrates how STAGE's training time on a problem instance may be reduced by *transferring* learned $\tilde{V}^\pi$ functions from previously solved, similar problem instances.

6.1 Using TD(λ) to learn V^π

In almost all of the experimental results of Chapters 4 and 5, STAGE learned to approximate V^π for a particularly simple choice of π: stochastic hillclimbing with equi-cost moves rejected. This procedure is proper, Markovian, and monotonic, and therefore—as shown earlier in Section 3.4.1—induces a Markov chain over the configuration space X. $V^\pi(x)$ is precisely the value function of the chain.

To approximate V^π, STAGE uses simple Monte-Carlo simulation and linear least-squares function approximation, as described in [Bertsekas and Tsitsiklis 96, §6.2.1]. However, a reinforcement learning technique, TD(λ) or *temporal difference learning*, also applies. TD(λ) is a family of incremental gradient-descent algorithms for approximating V^π, parametrized by $\lambda \in [0, 1]$ [Sutton 88]. For the case of $\lambda = 0$, Bradtke and Barto [96] demonstrated improved data efficiency with a least-squares formulation of the algorithm, which he called LSTD(0). LSTD(0) also eliminates all stepsize parameters from the TD procedure.

In this section, I generalize Bradtke and Barto's results to arbitrary values of $\lambda \in [0, 1]$, drawing on the analyses of TD(λ) in [Tsitsiklis and Roy 96, Bertsekas and Tsitsiklis 96]. I show that LSTD(1) produces the same coefficients as Monte-Carlo simulation with linear regression, but requires less memory (and no extra computation). I also explain how LSTD(λ) bridges the gap between model-free and model-based RL algorithms. Finally, I demonstrate empirical results with STAGE in the Bayes net structure-finding domain, showing that LSTD(λ) with λ set to values less than 1 can sometimes learn an effective approximation of V^π from less simulation data than supervised linear regression.

6.1.1 TD(λ): Background

TD(λ) addresses the problem of computing the value function V^π of a Markov chain, or equivalently, of a fixed policy π in a Markov Decision Problem. This is an important subproblem of several algorithms for sequential decision making, including policy iteration [Bertsekas and Tsitsiklis 96] and STAGE. $V^\pi(x)$ simply predicts the expected long-term sum of future rewards obtained when starting from state x and following policy π until termination. This function is well-defined as long as π is proper, i.e., guaranteed to terminate.[1]

For small Markov chains whose transition probabilities are all explicitly known, computing V^π is a trivial matter of solving a system of linear equations; TD(λ) is not needed. However, in many practical applications, the transition probabilities of the chain are available only implicitly: either in the form of a *simulation model* or in the form of an agent's actual experience executing π in its environment. In either case, we must compute V^π or an approximation to V^π solely from a collection of trajectories sampled from the chain. This is where the TD(λ) family of algorithms applies.

TD(λ) was introduced in [Sutton 88]; excellent summaries may now be found in several books [Bertsekas and Tsitsiklis 96, Sutton and Barto 98]. For each state on an observed trajectory $(x_0, x_1, \ldots, x_L, \text{END})$, TD($\lambda$) incrementally adjusts the coefficients of $\tilde{V}^\pi$ to more closely satisfy

$$\begin{aligned}\tilde{V}^\pi(x_t) \doteq (1-\lambda)\sum_{k=t}^{L-1} \lambda^{k-t}\big(\tilde{V}^\pi(x_{k+1}) + \textstyle\sum_{j=t}^{k} R_j\big) + \\ \lambda^{L-t}\big(\textstyle\sum_{j=t}^{L} R_j\big)\end{aligned} \tag{6.1}$$

where R_j is a shorthand for the one-step reward $R(x_j, x_{j+1})$. The right hand side of Equation 6.1 can be interpreted as computing the weighted average of $L-t$ different lookahead-based estimates for $\tilde{V}^\pi(x_t)$. The different estimates are the 1-step truncated return $(R_t + \tilde{V}^\pi(x_{t+1}))$, 2-step truncated return $R_t + R_{t+1} + \tilde{V}^\pi(x_{t+2})$, and so

[1] For improper policies, V^π may be made well-defined by the use of a discount factor that exponentially reduces future rewards; the TD(λ) and LSTD(λ) algorithms both extend straightforwardly to that case. However, for simplicity I will assume here that V^π is undiscounted and that π is proper.

forth up to the total Monte-Carlo return $(R_t + R_{t+1} + \cdots + R_L)$. The relative weights of these estimates are determined by the terms involving λ, which sum to unity. The λ parameter smoothly interpolates between two extremes:

- **TD(1)**: adjust $\tilde{V}^\pi$ based only on the Monte-Carlo return. This gives rise to an incremental form of supervised learning.

- **TD(0)**: adjust $\tilde{V}^\pi$ based only on the 1-step lookahead $R_t + \tilde{V}^\pi(x_{t+1})$. This gives rise to an incremental, sampled form of Value Iteration.

The target values assigned by TD(1) are unbiased samples of $\tilde{V}^\pi$, but may have significant variance since each depends on a long sequence of rewards from stochastic transitions. By contrast, TD(0)'s target values have low variance—the only random component is the reward of a single state transition—but are biased by the potential inaccuracy of the current estimate of $\tilde{V}^\pi$. The parameter λ trades off between bias and variance. Empirically, intermediate values of λ seem to perform best [Sutton 88, Singh and Sutton 96, Sutton 96].

A convenient form of the TD(λ) algorithm is given in Table 6.1.1. This version of the algorithm assumes that the policy π is proper, that the approximation architecture is linear (as described in Section 3.4.3), and that updates are offline (i.e., the coefficients of $\tilde{V}^\pi$ are modified only at the end of each trajectory). On each transition, the algorithm computes the scalar one-step TD error $R_t + (\boldsymbol{\phi}(x_{t+1}) - \boldsymbol{\phi}(x_t))^\mathsf{T}\boldsymbol{\beta}$, and apportions that error among all state features according to their respective *eligibilities* $\mathbf{z}_t$. The eligibility vector may be seen as an algebraic trick by which TD(λ) propagates rewards backward over the current trajectory without having to remember the trajectory explicitly. Each feature's eligibility at time t depends on the trajectory's history and on λ:

$$\mathbf{z}_t = \sum_{k=t_0}^{t} \lambda^{t-k} \boldsymbol{\phi}(x_k)$$

where t_0 is the time when the current trajectory started. In the case of TD(0), only the current state's features are eligible to be updated, so $\mathbf{z}_t = \boldsymbol{\phi}(x_t)$; whereas in TD(1), the features of all states seen so far on the current trajectory are eligible, so $\mathbf{z}_t = \sum_{k=t_0}^{t} \boldsymbol{\phi}(x_k)$.

The reason for the restriction to linear approximation architectures is that TD(λ) provably converges when such architectures are used, under a few mild additional assumptions detailed below [Tsitsiklis and Roy 96]. The currently available convergence results may be summarized as follows [Bertsekas and Tsitsiklis 96]:

TD($\boldsymbol{\lambda}$) for approximate policy evaluation:

Given:

- a *simulation model* for a proper policy π in MDP X;
- a *featurizer* $\boldsymbol{\phi} : X \to \Re^K$ mapping states to feature vectors, $\boldsymbol{\phi}(\text{END}) = \mathbf{0}$;
- a parameter $\lambda \in [0, 1]$; and
- a sequence of *stepsizes* $\alpha_1, \alpha_2, \ldots$ for incremental coefficient updating.

Output: a coefficient vector $\boldsymbol{\beta}$ for which $V^\pi(x) \approx \boldsymbol{\beta} \cdot \boldsymbol{\phi}(x)$.

Set $\boldsymbol{\beta} := \mathbf{0}$ *(arbitrary initial estimate)*, $t := 0$.
for $n := 1, 2, \ldots$ **do**: {
  Set $\boldsymbol{\delta} := 0$.
  Choose a start state $x_t \in X$.
  Set $\mathbf{z}_t := \boldsymbol{\phi}(x_t)$.
  while $x_t \neq \text{END}$, **do**: {
    Simulate one step of the chain, producing a reward R_t and next state x_{t+1}.
    Set $\boldsymbol{\delta} := \boldsymbol{\delta} + \mathbf{z}_t\big(R_t + (\boldsymbol{\phi}(x_{t+1}) - \boldsymbol{\phi}(x_t))^\mathsf{T}\boldsymbol{\beta}\big)$.
    Set $\mathbf{z}_{t+1} := \lambda\mathbf{z}_t + \boldsymbol{\phi}(x_{t+1})$.
    Set $t := t + 1$.
  }
  Set $\boldsymbol{\beta} := \boldsymbol{\beta} + \alpha_n\boldsymbol{\delta}$.
}

LSTD($\boldsymbol{\lambda}$) for approximate policy evaluation:

Given: a *simulation model*, *featurizer*, and λ as above; no stepsizes.

Output: a coefficient vector $\boldsymbol{\beta}$ for which $V^\pi(x) \approx \boldsymbol{\beta} \cdot \boldsymbol{\phi}(x)$.

Set $\mathbf{A} := \mathbf{0}$, $\mathbf{b} := \mathbf{0}$, $t := 0$.
for $n := 1, 2, \ldots$ **do**: {
  Choose a start state $x_t \in X$.
  Set $\mathbf{z}_t := \boldsymbol{\phi}(x_t)$.
  while $x_t \neq \text{END}$, **do**: {
    Simulate one step of the chain, producing a reward R_t and next state x_{t+1}.
    Set $\mathbf{A} := \mathbf{A} + \mathbf{z}_t(\boldsymbol{\phi}(x_t) - \boldsymbol{\phi}(x_{t+1}))^\mathsf{T}$.
    Set $\mathbf{b} := \mathbf{b} + \mathbf{z}_t R_t$.
    Set $\mathbf{z}_{t+1} := \lambda\mathbf{z}_t + \boldsymbol{\phi}(x_{t+1})$.
    Set $t := t + 1$.
  }
  Whenever updated coefficients are desired: Set $\boldsymbol{\beta} := \mathbf{A}^{-1}\mathbf{b}$. *(Use SVD.)*
}

TABLE 6.1. Gradient-descent and least-squares versions of trial-based TD(λ) for approximating the undiscounted value function of a fixed proper policy. Note that $\mathbf{A}$ has dimension $K \times K$, and $\mathbf{b}$, $\boldsymbol{\beta}$, $\boldsymbol{\delta}$, $\mathbf{z}$, and $\boldsymbol{\phi}(x)$ all have dimension $K \times 1$.

- If the approximator is nonlinear, TD(λ) may diverge [Bertsekas and Tsitsiklis 96], though it has been successful with nonlinear neural networks in practice.

- If the function approximator is linear and can represent the optimal V^π exactly, then TD(λ) will converge to V^π. This result applies primarily to representations with one independent feature per state, i.e., lookup tables.

- If the approximator is linear but cannot represent V^π exactly, then TD(1) will converge to the best least-squares fit of V^π. For values of $\lambda < 1$, TD(λ) will also converge, but possibly to a suboptimal fit of V^π; the error bound worsens as λ decreases toward zero [Bertsekas and Tsitsiklis 96]. In practice, though, smaller values of λ introduce less variance and may enable TD(λ) to converge to an acceptable approximation of V^π using less data.

Sufficient conditions for the convergence, with probability 1, of TD(λ) are as follows:

1. The stepsizes α_n are nonnegative, satisfy $\sum_{n=1}^{\infty} \alpha_n = \infty$, and satisfy $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$. (The stepsizes $\alpha_n = c/n$ satisfy this condition, though in practice a small constant stepsize is often used.)

2. All states $x \in X$ have positive probability of being visited given the start state distribution chosen. (This is a technicality: unvisited states may simply be deleted from the chain.)

3. The features $\boldsymbol{\phi}(x)$ are linearly independent of one another over X. That is, the feature set is not redundant.

What does TD(λ) converge to? Examining the update rule for $\boldsymbol{\delta}$ in Table 6.1.1, it is not difficult to see that the coefficient changes made by TD(λ) after an observed trajectory $(x_0, x_1, \ldots, x_L, \text{END})$ have the form

$$\boldsymbol{\beta} := \boldsymbol{\beta} + \alpha_n(\mathbf{d} + \mathbf{C}\boldsymbol{\beta} + \boldsymbol{\omega})$$

where

$$\begin{aligned}
\mathbf{d} &= \mathrm{E}\Big\{\sum_{i=0}^{L} \mathbf{z}_i R(x_i, x_{i+1})\Big\} \\
\mathbf{C} &= \mathrm{E}\Big\{\sum_{i=0}^{L} \mathbf{z}_i \big(\boldsymbol{\phi}(x_{i+1}) - \boldsymbol{\phi}(x_i)\big)^{\mathsf{T}}\Big\} \\
\boldsymbol{\omega} &= \text{zero-mean noise.}
\end{aligned}$$

The expectations are taken with respect to the distribution of trajectories through the Markov chain. It is shown in [Bertsekas and Tsitsiklis 96] that $\mathbf{C}$ is negative definite and that the noise $\boldsymbol{\omega}$ has sufficiently small variance, which together with the stepsize conditions given above, imply that $\boldsymbol{\beta}$ converges to a fixed point $\boldsymbol{\beta}_\lambda$ satisfying

$$\mathbf{d} + \mathbf{C}\boldsymbol{\beta}_\lambda = \mathbf{0}.$$

In effect, TD(λ) solves this system of equations by performing stochastic gradient descent on the potential function $\|\boldsymbol{\beta} - \boldsymbol{\beta}_\lambda\|^2$. It never explicitly represents $\mathbf{d}$ or $\mathbf{C}$. The changes to $\boldsymbol{\beta}$ depend only on the most recent trajectory, and after those changes are made, the trajectory and its rewards are simply forgotten. This approach, while requiring little computation time per iteration, is wasteful with data and may require sampling many trajectories to reach convergence.

One technique for using data more efficiently is "experience replay" [Lin 93]: explicitly remember all trajectories ever seen, and whenever asked to produce an updated set of coefficients, perform repeated passes of TD(λ) over all the saved trajectories until convergence. This technique is similar to the batch training methods commonly used to train neural networks. However, in the case of linear function approximators, there is another way.

6.1.2 The Least-Squares TD(λ) Algorithm

The Least-Squares TD(λ) algorithm, or LSTD(λ), converges to the same coefficients $\boldsymbol{\beta}_\lambda$ that TD(λ) does. However, instead of performing gradient descent, LSTD(λ) builds explicit estimates of the $\mathbf{C}$ matrix and $\mathbf{d}$ vector (actually, estimates of a constant multiple of $\mathbf{C}$ and $\mathbf{d}$), and then effectively solves $\mathbf{d} + \mathbf{C}\boldsymbol{\beta}_\lambda = \mathbf{0}$ directly. The actual data structures that LSTD(λ) builds from experience are the matrix $\mathbf{A}$ (of dimension $K \times K$, where K is the number of features) and the vector $\mathbf{b}$ (of dimension K):

$$\mathbf{b} = \sum_{i=0}^{t} \mathbf{z}_i R(x_i, x_{i+1})$$
$$\mathbf{A} = \sum_{i=0}^{t} \mathbf{z}_i \big(\boldsymbol{\phi}(x_i) - \boldsymbol{\phi}(x_{i+1})\big)^{\mathsf{T}}$$

After n independent trajectories have been observed, $\mathbf{b}$ is an unbiased estimate of $n\mathbf{d}$, and $\mathbf{A}$ is an unbiased estimate of $-n\mathbf{C}$. Thus, $\boldsymbol{\beta}_\lambda$ can be estimated as $\mathbf{A}^{-1}\mathbf{b}$. As in the least-squares linear regression technique of Section 3.4.3, I use Singular Value Decomposition to invert $\mathbf{A}$ robustly [Press *et al.* 92]. The complete LSTD(λ) algorithm is specified in the bottom half of Table 6.1.1.

LSTD(λ) is a generalization of the LSTD(0) algorithm [Bradtke and Barto 96] to the case of arbitrary λ. When $\lambda = 0$, the equations reduce to

$$\begin{aligned} \mathbf{b} &= \sum_{i=0}^{t} \boldsymbol{\phi}(x_i) R(x_i, x_{i+1}) \\ \mathbf{A} &= \sum_{i=0}^{t} \boldsymbol{\phi}(x_i) \big(\boldsymbol{\phi}(x_i) - \boldsymbol{\phi}(x_{i+1})\big)^{\mathsf{T}}, \end{aligned} \tag{6.2}$$

the same as those derived by Bradtke and Barto using a different approach based on regression with instrumental variables [Bradtke and Barto 96].

At the other extreme, when $\lambda = 1$, LSTD(1) produces precisely the same $\mathbf{A}$ and $\mathbf{b}$ as would be produced by supervised linear regression on training pairs of (state features $\mapsto$ observed eventual outcomes), as described for STAGE in Chapter 3 (Equation 3.8, page 67). The proof of this equivalence is given in Appendix A.2. Thanks to the algebraic trick of the eligibility vectors, LSTD(1) builds the regression matrices *fully incrementally*—without having to store the trajectory while waiting to observe the eventual outcome. When trajectories through the chain are long, this provides significant memory savings over linear regression.

The computation per timestep required to update $\mathbf{A}$ and $\mathbf{b}$ is the same as least-squares linear regression: $O(K^2)$, where K is the number of features. LSTD(λ) must also perform a matrix inversion at a cost of $O(K^3)$ whenever $\boldsymbol{\beta}$'s coefficients are needed—in the case of STAGE, once per complete trajectory. (If updated coefficients are required more frequently, then the $O(K^3)$ cost can be avoided by *recursive least-squares* [Bradtke and Barto 96] or Kalman-filtering techniques [Bertsekas and Tsitsiklis 96, §3.2.2], which update $\boldsymbol{\beta}$ on each timestep at a cost of only $O(K^2)$.) LSTD(λ) is more computationally expensive than incremental TD(λ), which updates the coefficients using only $O(K)$ computation per timestep. However, LSTD(λ) offers several significant advantages, as pointed out by Bradtke and Barto in their discussion of LSTD(0) [96]:

- Least-squares algorithms are "more efficient estimators in the statistical sense" because "they extract more information from each additional observation."

- TD(λ)'s convergence can be slowed dramatically by a poor choice of the stepsize parameters α_n. LSTD(λ) eliminates these parameters.

- TD(λ)'s performance is sensitive to $\|\boldsymbol{\beta}_\lambda - \boldsymbol{\beta}_{\text{init}}\|$, the distance between $\boldsymbol{\beta}_\lambda$ and the initial estimate for $\boldsymbol{\beta}_\lambda$. LSTD($\lambda$) requires no arbitrary initial estimate.

- TD(λ) is also sensitive to the ranges of the individual features. LSTD(λ) is not.

Section 6.1.4 below presents experimental results comparing TD(λ) with LSTD(λ) in terms of data efficiency and time efficiency.

6.1.3 LSTD(λ) as Model-Based TD(λ)

Before giving experimental results with LSTD(λ), I would like to point out an interesting connection between LSTD(λ) and model-based reinforcement learning. To begin, let us restrict our attention to the case of a small discrete state space X, over which V^π can be represented and learned exactly by a lookup table. A classical model-based algorithm for learning V^π from simulated trajectory data would proceed as follows:

1. From the state transitions and rewards observed so far, build in memory an *empirical model* of the Markov chain. The sufficient statistics of this model consist of, for each state $x \in X$:
   - $n(x)$, the number of times state x was visited;
   - $c(x'|x)$, the count of how many times x' followed x for each state $x' \in X$. We do not need to track the absorption frequency $c(\text{END}|x)$ separately, since $c(\text{END}|x) = n(x) - \sum_{x' \in X} c(x'|x)$.
   - $s(x)$, the sum of all observed one-step rewards from state x. (The expected reward at x is then given simply by $s(x)/n(x)$.)
2. Whenever a new estimate of the value function V^π is desired, solve the current empirical model. Solving the model means solving the linear system of Bellman equations (Eq. 2.3), which can be written in the above notation as, $\forall x \in X$:

$$V^\pi(x) = \frac{s(x)}{n(x)} + \sum_{x' \in X} \frac{c(x'|x)}{n(x)} V^\pi(x') \tag{6.3}$$

   In matrix notation, if we let $\mathbf{N}$ be the diagonal matrix of visitation frequencies $n(x)$, let $\mathbf{C}$ be the matrix of counts where $\mathbf{C}_{ij} = c(x_j|x_i)$, let $\mathbf{s}$ be the vector of summed-rewards $s(x)$, and let $\mathbf{v}$ be the vector of V^π values, then the above equations become simply

$$\mathbf{N}\mathbf{v} = \mathbf{s} + \mathbf{C}\mathbf{v},$$

   whose solution is given by

$$\mathbf{v} = (\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}. \tag{6.4}$$

This model-based technique contrasts with TD(λ), a model-free approach to the same problem. TD(λ) does not maintain any statistics on observed transitions and rewards; it simply updates the components of $\mathbf{v}$ incrementally after each observed trajectory. In the limit, assuming a lookup-table representation, both converge to the optimal V^π. The advantage of TD(λ) is its low computational burden per step; the advantage of the classical model-based method is that it makes the most of the available training data. The empirical advantages of model-based and model-free reinforcement learning methods have been investigated in, e.g., [Sutton 90, Moore and Atkeson 93, Atkeson and Santamaria 97, Kuvayev and Sutton 97].

Where does LSTD(λ) fit in? The answer is that, when $\lambda = 0$, it precisely duplicates the classical model-based method sketched above. When $\lambda > 0$, it does something else sensible, which I describe below; but let us first consider LSTD(0). The assumed lookup-table representation for $\tilde{V}^\pi$ means that we have one independent feature per state: the feature vector $\boldsymbol{\phi}$ corresponding to state 1 is $(1, 0, 0, \ldots, 0)$; corresponding to state 2 is $(0, 1, 0, \ldots, 0)$; etc. LSTD(0) performs the following operations upon each observed transition (cf. Equation 6.2):

$$\begin{aligned} \mathbf{b} &:= \mathbf{b} + \boldsymbol{\phi}(x_t) R_t \\ \mathbf{A} &:= \mathbf{A} + \boldsymbol{\phi}(x_t)(\boldsymbol{\phi}(x_t) - \boldsymbol{\phi}(x_{t+1}))^\mathsf{T} \end{aligned}$$

Clearly, the role of $\mathbf{b}$ is to sum all the rewards observed at each state, exactly as the vector $\mathbf{s}$ does in the classical technique. $\mathbf{A}$, meanwhile, accumulates the statistics $(\mathbf{N} - \mathbf{C})$. To see this, note that the outer product given above is a matrix consisting of an entry of $+1$ on the single diagonal element corresponding to state x_t; an entry of -1 on the element in row x_t, column x_{t+1}; and all the rest zeroes. Summing one such sparse matrix for each observed transition gives $\mathbf{A} \equiv \mathbf{N} - \mathbf{C}$. Finally, LSTD(0) performs the inversion $\boldsymbol{\beta} := \mathbf{A}^{-1}\mathbf{b} = (\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}$, giving the same solution as in Equation 6.4.

Thus, when $\lambda = 0$, the $\mathbf{A}$ and $\mathbf{b}$ matrices built by LSTD(λ) effectively record a model of all the observed transitions. What about when $\lambda > 0$? Again, $\mathbf{A}$ and $\mathbf{b}$ record the sufficient statistics of an empirical Markov model—but in this case, the model being captured is one whose single-step transition probabilities directly encode the multi-step TD(λ) backup operations, as defined by Eq. 6.1. That is, the model links each state x to *all* the downstream states that follow x on any trajectory, and records how much influence each has on estimating $\tilde{V}^\pi(x)$ according to TD(λ). In the case of $\lambda = 0$, the TD(λ) backups correspond to the one-step transitions, resulting in the equivalence described above. The opposite extreme, the case of $\lambda = 1$, is also interesting: the empirical Markov model corresponding to TD(1)'s backups is the chain where each state x leads directly to absorption (i.e., all counts are zero: $\mathbf{C}_1 = \mathbf{0}$). The values $s_1(x)$ equal the sum, over all visits to x on all trajectories, of all the observed rewards between x and termination. LSTD(1) then solves for $\tilde{V}^\pi$ as follows:

$$\boldsymbol{\beta} := \mathbf{A}^{-1}\mathbf{b} = (\mathbf{N} - \mathbf{C}_1)^{-1}\mathbf{s}_1 = \mathbf{N}^{-1}\mathbf{s}_1,$$

which simply computes the average of the Monte-Carlo returns at each state x. In short, if we assume a lookup-table representation for the function $\tilde{V}^\pi$, we can view the LSTD(λ) algorithm as doing these two steps:

1. It implicitly uses the observed simulation data to build a Markov chain. This chain compactly models all the backups that TD(λ) would perform on the data.
2. It solves the chain by performing a matrix inversion.

The lookup-table representation for $\tilde{V}^\pi$ is intractable in practical problems; in practice, LSTD(λ) operates on states only via their (linearly *dependent*) feature representations $\boldsymbol{\phi}(x)$. In this case, we can view LSTD(λ) as implicitly building a *compressed* version of the empirical model's transition matrix $\mathbf{N}-\mathbf{C}$ and summed-reward vector $\mathbf{s}$:

$$\begin{aligned} \mathbf{A} &= \mathbf{\Phi}^\mathsf{T}(\mathbf{N} - \mathbf{C})\mathbf{\Phi} \\ \mathbf{b} &= \mathbf{\Phi}^\mathsf{T}\mathbf{s}, \end{aligned}$$

where $\mathbf{\Phi}$ is the $|X| \times K$ matrix representation of the function $\boldsymbol{\phi} : X \to \Re^K$. From the compressed empirical model, LSTD(λ) computes the following coefficients for $\tilde{V}^\pi$:

$$\begin{aligned} \boldsymbol{\beta}_\lambda &= \mathbf{A}^{-1}\mathbf{b} \\ &= (\mathbf{\Phi}^\mathsf{T}(\mathbf{N} - \mathbf{C})\mathbf{\Phi})^{-1}(\mathbf{\Phi}^\mathsf{T}\mathbf{s}). \end{aligned} \tag{6.5}$$

Ideally, these coefficients $\boldsymbol{\beta}_\lambda$ would be equivalent to the *empirical optimal* coefficients $\boldsymbol{\beta}^*_\lambda$. The empirical optimal coefficients are those that would be found by building the full uncompressed empirical model (represented by $\mathbf{N} - \mathbf{C}$ and $\mathbf{s}$), using a lookup table to solve for that model's value function ($\mathbf{v} = (\mathbf{N}-\mathbf{C})^{-1}\mathbf{s}$), and then performing a least-squares linear fit from the state features $\mathbf{\Phi}$ to the lookup-table value function:

$$\begin{aligned} \boldsymbol{\beta}^*_\lambda &= (\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi})^{-1}(\mathbf{\Phi}^\mathsf{T}\mathbf{v}) \\ &= (\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi})^{-1}\mathbf{\Phi}^\mathsf{T}(\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}. \end{aligned} \tag{6.6}$$

It can be seen that Equations 6.5 and 6.6 are indeed equivalent for the case of $\lambda = 1$, because that setting of λ implies that $\mathbf{C} = \mathbf{0}$ so $(\mathbf{N}-\mathbf{C})$ is diagonal. However, for the case of $\lambda < 1$, solving the compressed empirical model does not in general produce the optimal least-squares fit to the solution of the uncompressed model.

6.1.4 Empirical Comparison of TD(λ) and LSTD(λ)

I experimentally compared the performance of TD(λ) and LSTD(λ) on the simple "Hopworld" Markov chain described in Section 2.3.1. The chain consists of 13 states, as illustrated in Figure 2.2 (p. 31). We seek to represent the value function of this chain compactly—as a linear function of four state features. In fact, this domain has been contrived so that the optimal V^π function is exactly linear in these features: the optimal coefficients $\boldsymbol{\beta}^*_\lambda$ are $(-24, -16, -8, 0)$. This condition guarantees that LSTD(λ) will converge with probability 1 to the optimal $\boldsymbol{\beta}^*_\lambda$ for any setting of λ.

TD(λ) is also guaranteed convergence to the optimal V^π, under the additional condition that an appropriate schedule of stepsizes is chosen. As mentioned in Section 6.1.1 above, the sequence of stepsizes (α_n) must satisfy three criteria: $\alpha_n \geq 0 \ \ \forall n$; $\sum_{n=1}^{\infty} \alpha_n = \infty$; and $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$. For example, all three criteria are satisfied by schedules of the following form:

$$\alpha_n \stackrel{\text{def}}{=} \alpha_0 \frac{n_0 + 1}{n_0 + n} \qquad n = 1, 2, \ldots \tag{6.7}$$

The parameter α_0 determines the initial stepsize, and n_0 determines how gradually the stepsize decreases over time. I ran each TD(λ) experiment with six different stepsize schedules, corresponding to the six combinations of $\alpha_0 \in \{0.1, 0.01\}$ and $n_0 \in \{10^2, 10^3, 10^6\}$. These six schedules are plotted in Figure 6.1.

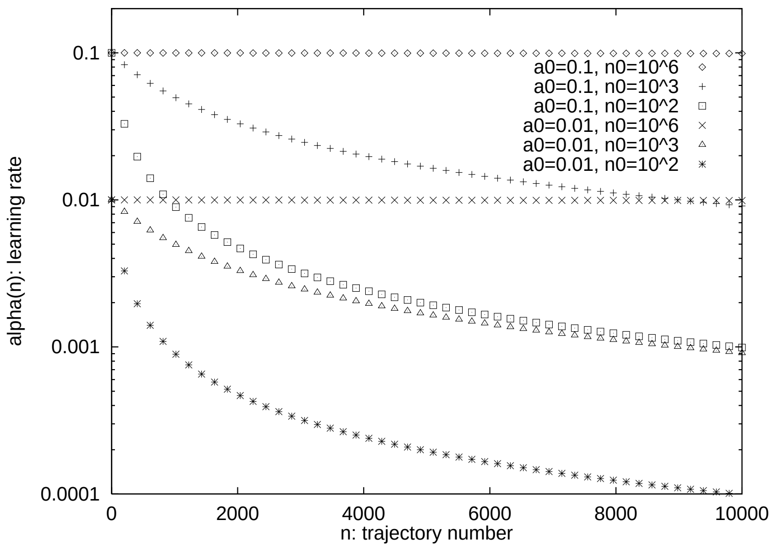

FIGURE 6.1. The six different *stepsize schedules* used in the experiments with TD(λ). The schedules are determined by Equation 6.7 with various settings for α_0 and n_0.

On the Hopworld domain, I ran both TD(λ) and LSTD(λ) for a variety of settings of λ. Figure 6.2 plots the results for the case of $\lambda = 0.4$. The plot compares the performance of six variants of TD(0.4)—corresponding to the six different stepsize schedules—and LSTD(0.4). The x-axis counts the number of trajectories sampled, up to a limit of 10,000; and the y-axis measures the RMS error of the approximated value function $\tilde{V}^\pi$, defined by

$$\|\tilde{V}^\pi - V^\pi\| = \left(\frac{1}{|X|}\sum_{x \in X}\left(\tilde{V}^\pi(x) - V^\pi(x)\right)^2\right)^{\frac{1}{2}} . \tag{6.8}$$

Each point on the plot represents the average of 10 trials.

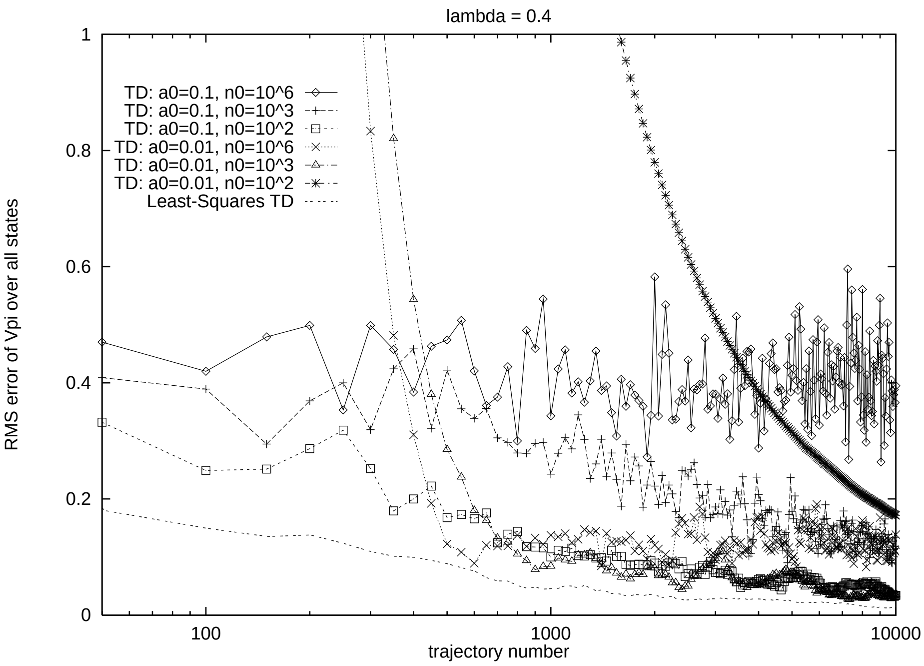

FIGURE 6.2. Performance of TD(0.4) and LSTD(0.4) on the Hopworld domain. Note the logarithmic scale of the x-axis.

The plot shows clearly that for $\lambda = 0.4$, LSTD(λ) learns a good approximation to V^π in fewer trials than any of the TD(λ) experiments, and performs better asymptotically as well. These results held uniformly across all values of λ. Table 6.2 gives the results for $\lambda = 0$, $\lambda = 0.4$, and $\lambda = 1$. The results may be summarized as follows:

- For all values of λ, the convergence rate of LSTD(λ) exceeded that of TD(λ).

Algorithm	Fit error after 100 trajectories	Fit error after 10,000 trajectories
TD(0), $\alpha_0 = 0.1, n_0 = 10^6$	0.49± 0.15	0.37± 0.05
TD(0), $\alpha_0 = 0.1, n_0 = 10^3$	0.38± 0.10	0.09± 0.02
TD(0), $\alpha_0 = 0.1, n_0 = 10^2$	0.32± 0.07	0.04± 0.02
TD(0), $\alpha_0 = 0.01, n_0 = 10^6$	9.08± 0.04	0.10± 0.02
TD(0), $\alpha_0 = 0.01, n_0 = 10^3$	9.29± 0.04	0.04± 0.02
TD(0), $\alpha_0 = 0.01, n_0 = 10^2$	10.66± 0.02	0.70± 0.01
LSTD(0)	**0.19**± 0.09	**0.01**± 0.01
TD(0.4), $\alpha_0 = 0.1, n_0 = 10^6$	0.42± 0.15	0.40± 0.09
TD(0.4), $\alpha_0 = 0.1, n_0 = 10^3$	0.39± 0.16	0.12± 0.04
TD(0.4), $\alpha_0 = 0.1, n_0 = 10^2$	0.25± 0.08	0.03± 0.01
TD(0.4), $\alpha_0 = 0.01, n_0 = 10^6$	6.73± 0.07	0.14± 0.04
TD(0.4), $\alpha_0 = 0.01, n_0 = 10^3$	6.97± 0.06	0.04± 0.01
TD(0.4), $\alpha_0 = 0.01, n_0 = 10^2$	8.73± 0.04	0.17± 0.01
LSTD(0.4)	**0.15**± 0.04	**0.01**± 0.00
TD(1), $\alpha_0 = 0.1, n_0 = 10^6$	0.73± 0.27	0.54± 0.15
TD(1), $\alpha_0 = 0.1, n_0 = 10^3$	0.48± 0.20	0.17± 0.06
TD(1), $\alpha_0 = 0.1, n_0 = 10^2$	0.30± 0.10	0.06± 0.02
TD(1), $\alpha_0 = 0.01, n_0 = 10^6$	1.86± 0.14	0.13± 0.03
TD(1), $\alpha_0 = 0.01, n_0 = 10^3$	2.05± 0.14	0.05± 0.02
TD(1), $\alpha_0 = 0.01, n_0 = 10^2$	3.31± 0.12	0.03± 0.01
LSTD(1)	**0.14**± 0.04	**0.01**± 0.00

TABLE 6.2. Summary of results with TD(λ) and LSTD(λ) on the Hopworld domain for $\lambda = 0$, 0.4, and 1.0. Fit errors are measured according to Equation 6.8; the mean over 10 trials and 95% confidence interval of the mean are displayed. Results for other values of λ were similar.

- The performance of TD(λ) depends critically on the stepsize schedule chosen. LSTD(λ) has no tunable parameters other than λ itself.

- Varying λ had no discernible effect on LSTD(λ)'s performance.

Because the Hopworld domain is so small and the optimal value function is exactly linear over the available features, these results are not necessarily representative of how TD(λ) and LSTD(λ) will perform on practical problems. For example, if a domain has many features and simulation data is available cheaply, then incremental methods will have better real-time performance than least-squares methods [Sutton 92]. On the other hand, some reinforcement-learning applications have been successful with very small numbers of features (e.g., [Singh and Bertsekas 97]). STAGE's results certainly meet this description. I investigate the performance of LSTD(λ) in the context of STAGE in the following section.

One exciting possibility for future work is to apply LSTD(λ) in the context of Markov decision problems—that is, for the purpose of approximating not V^π but V^*. LSTD(λ) could provide an efficient alternative to TD(λ) in the inner loop of optimistic policy iteration [Bertsekas and Tsitsiklis 96].

6.1.5 Applying LSTD(λ) in STAGE

The application of LSTD(λ) within STAGE is straightforward. We are given a local search procedure π that is assumed to be proper, Markovian and monotonic, which implies that π induces a Markov chain over the configuration space X. (To use LSTD(λ) with a nonmonotonic local search procedure such as WALKSAT, the best-so-far abstraction must be applied; see Appendix A.1.) Note that the one-step rewards in the induced Markov chain are all zero except at termination (see Eq. 3.4, p. 59). This means that the update step $\mathbf{b} := \mathbf{b} + \mathbf{z}_t R_t$ may be moved outside the inner loop of LSTD(λ).

The interesting empirical question concerns the best value of λ. The STAGE experiments of Chapters 4 and 5 all used supervised least-squares regression, which is equivalent to $\lambda = 1$. Is that value of λ best, since the predictions it generates are unbiased estimates of the true V^π function? Or will a lower setting of λ enable learning in fewer trials, since LSTD(λ)'s implicit transition model enables it to treat noisy training data more informedly?

I performed experiments with LSTD(λ) on three optimization problem instances: cartogram design instance US49 and Bayes net structure-finding instances SYNTH125K and ADULT2. Other than λ, all STAGE parameters were fixed at the same settings specified earlier in Sections 4.4 and 4.6. The λ parameter was varied over six evenly-spaced settings between 0 and 1.

The experimental results, shown in Table 6.3 and the accompanying three figures, were inconclusive. On the cartogram design problem, LSTD(1) decisively outperformed all the smaller values of λ. By contrast, on SYNTH125K, all values of $\lambda \leq 0.8$ produced significantly faster learning than LSTD(1), although Figure 6.4 shows that LSTD(1) nearly catches up by the time 100,000 moves have been considered. Finally, on instance ADULT2, there were no significant differences between any of the different settings for λ. From this set of results, the most that can be said is that values of $\lambda < 1$ sometimes help, sometimes hurt, and sometimes have no effect on the performance of STAGE in practice.

Instance	Algorithm	Performance (N runs each)		
		mean	best	worst
Cartogram	LSTD(0.0)	0.083±0.008	0.040	0.178
(US49)	LSTD(0.2)	0.078±0.009	0.040	0.183
$N = 50, M = 10^6$	LSTD(0.4)	0.074±0.007	0.041	0.170
	LSTD(0.6)	0.079±0.010	0.038	0.204
	LSTD(0.8)	0.076±0.007	0.042	0.175
	LSTD(1.0)	**0.057**±0.004	**0.037**	**0.105**
Bayes net	LSTD(0.0)	**736241**± 1721	720244	784874
(SYNTH125K)	LSTD(0.2)	**734806**± 1722	719431	777711
$N = 200, M = 4 \cdot 10^4$	LSTD(0.4)	**734548**± 1674	719187	779267
	LSTD(0.6)	**736164**± 1938	719261	796555
	LSTD(0.8)	**736094**± 1796	**719068**	**776308**
	LSTD(1.0)	741111± 1871	719748	790014
Bayes net	LSTD(0.0)	**440511**± 59	439372	441052
(ADULT2)	LSTD(0.2)	**440531**± 62	439460	441247
$N = 100, M = 10^5$	LSTD(0.4)	**440540**± 60	439761	441168
	LSTD(0.6)	**440490**± 52	439767	441208
	LSTD(0.8)	**440484**± 67	**439267**	441152
	LSTD(1.0)	**440461**± 61	439715	**441005**

TABLE 6.3. STAGE performance with LSTD(λ) on three optimization instances. Each line summarizes the performance of N trials, each limited to considering M total search moves.

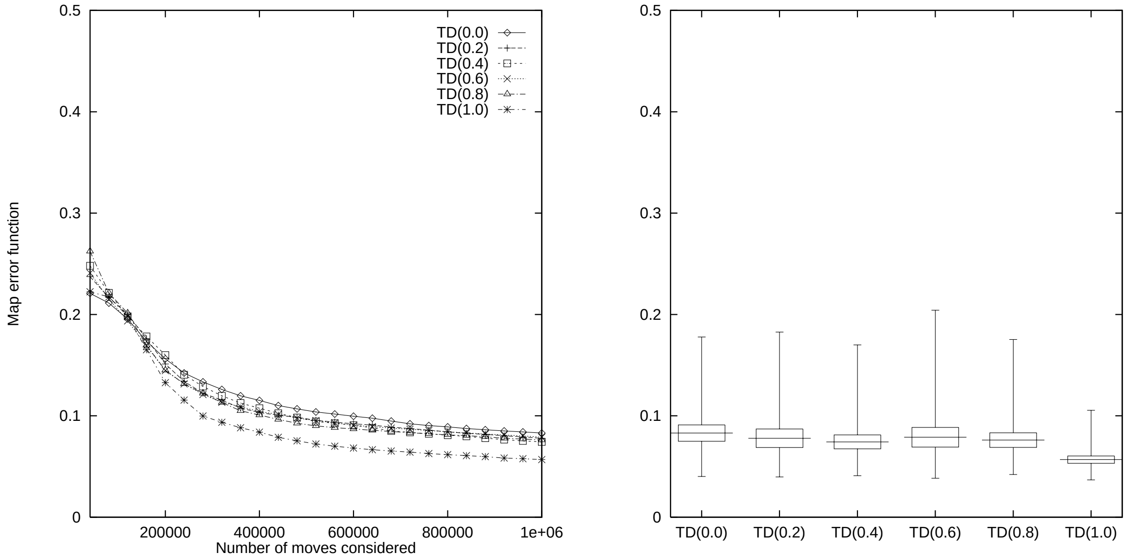

FIGURE 6.3. Cartogram performance with LSTD(λ)

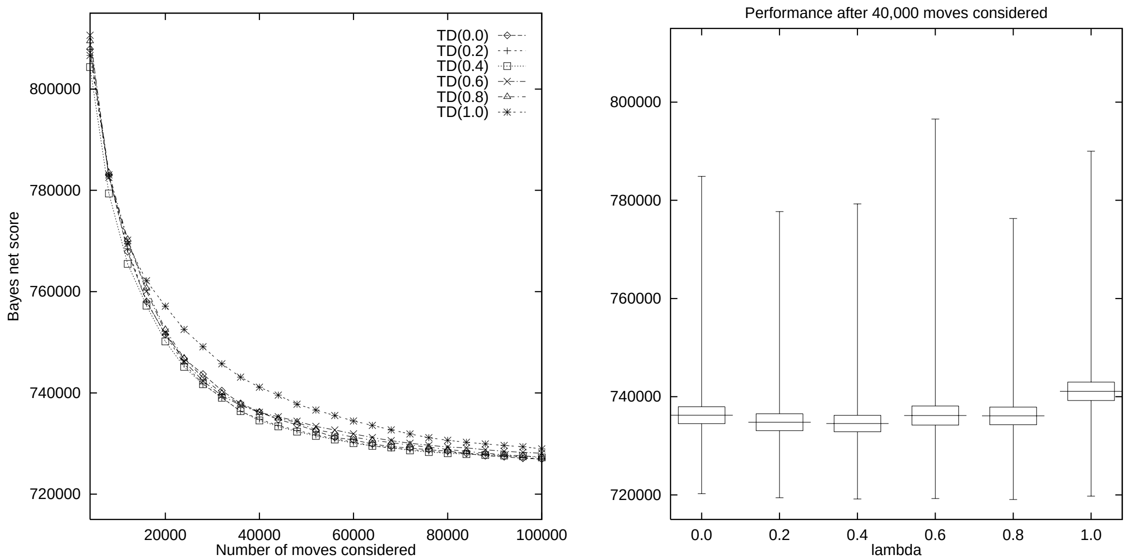

FIGURE 6.4. Performance of STAGE with LSTD(λ) on Bayes net structure-finding instance SYNTH125K. After 40,000 moves, LSTD(1.0) significantly lagged all other values for λ.

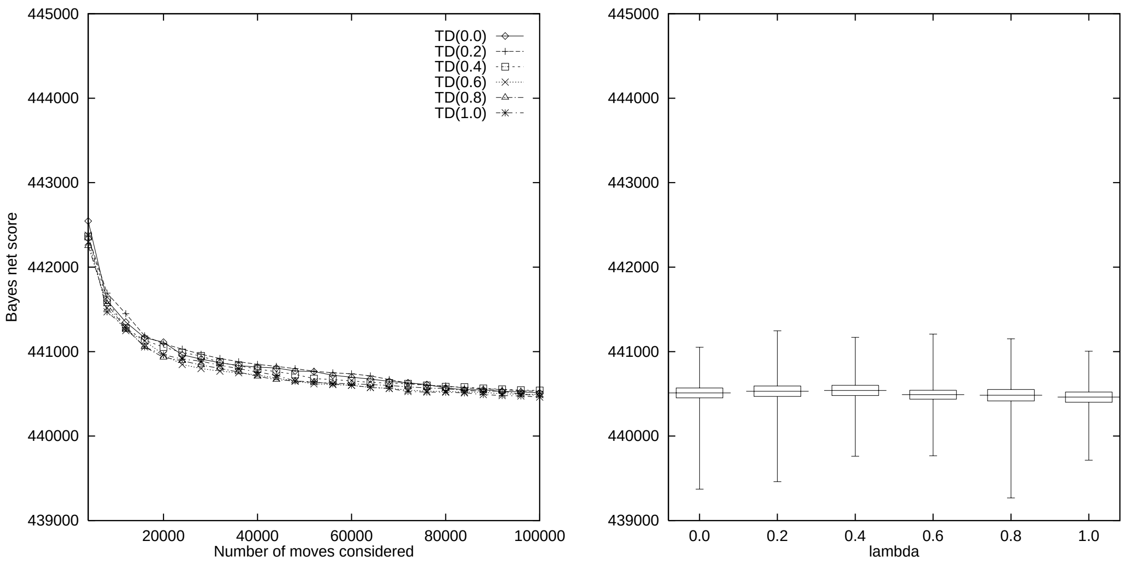

FIGURE 6.5. Performance of STAGE with LSTD(λ) on Bayes net structure-finding instance ADULT2

6.2 Transfer

This section concerns a second significant extension to STAGE: enabling *transfer* of learned knowledge between related problem instances. I motivate and describe an algorithm for transfer, X-STAGE, and present empirical results on two problem families.

6.2.1 Motivation

There is a computational cost to training a function approximator on V^π. Learning from a π-trajectory of length L, with either linear regression or LSTD(λ) over D features, costs STAGE $O(D^2L + D^3)$ per iteration; quadratic regression costs $O(D^4L + D^6)$. In the experiments of Chapter 4, these costs were minimal—typically, 0–10% of total execution time. However, STAGE's extra overhead would become significant if many more features or more sophisticated function approximators were used. Furthermore, even if the function approximation is inexpensive, STAGE may require many trajectories to be sampled in order to obtain sufficient data to fit V^π effectively.

For some problems such costs are worth it in comparison with a non-learning method, because a better or equally good solution is obtained with overall less computation. But in those cases where we use more computation, the STAGE method may nevertheless be preferable if we are then asked to solve further similar problems (e.g., a new channel routing problem with different pin assignments). Then we can hope that the computation we invested in solving the first problem will pay off in the second, and future, problems because we will already have a $\tilde{V}^{\pi}$ estimate. This effect is called *transfer*; the extent to which it occurs is largely an empirical question.

To investigate the potential for transfer, I re-ran STAGE on a suite of eight problems from the channel routing literature [Chao and Harper 96]. Table 6.4 summarizes the results and gives the coefficients of the linear evaluation function, learned independently for each problem. To make the similarities easier to see in the table, I have normalized the coefficients so that their squares sum to one; note that the search behavior of an evaluation function is invariant under positive linear transformations.

Problem instance	lower bound	best-of-3 hillclimbing	best-of-3 STAGE	learned coefficients $< \beta_w, \beta_p, \beta_U >$
YK4	10	22	12	$< 0.71, 0.05, -0.70 >$
HYC1	8	8	8	$< 0.52, 0.83, -0.19 >$
HYC2	9	9	9	$< 0.71, 0.21, -0.67 >$
HYC3	11	12	12	$< 0.72, 0.30, -0.62 >$
HYC4	20	27	23	$< 0.71, 0.03, -0.71 >$
HYC5	35	39	38	$< 0.69, 0.14, -0.71 >$
HYC6	50	56	51	$< 0.70, 0.05, -0.71 >$
HYC7	39	54	42	$< 0.71, 0.13, -0.69 >$
HYC8	21	29	25	$< 0.71, 0.03, -0.70 >$

TABLE 6.4. STAGE results on eight problems from [Chao and Harper 96]. The coefficients have been normalized so that their squares sum to one.

The similarities among the learned evaluation functions are striking. Except on the trivially small instance HYC1, all the STAGE-learned functions assign a relatively large positive weight to feature $w(x)$, a similarly large negative weight to feature $U(x)$, and a small positive weight to feature $p(x)$. In Section 5.1.3 (see Eq. 5.1, page 116), I explained the coefficients found on instance YK4 as follows: STAGE has learned that good hillclimbing performance is predicted by an uneven distribution of track fullness levels $(w(x) - U(x))$ and by a low analytical bound on the effect of further merging of tracks $(p(x))$. From Table 6.4, we can conclude that this explanation holds generally for many channel routing instances. Thus, transfer between instances should be fruitful.

6.2.2 X-STAGE: A Voting Algorithm for Transfer

Many sensible methods for transferring the knowledge learned by STAGE from "training" problem instances to new instances can be imagined. This section presents one such method. STAGE's learned knowledge, of course, is represented by the approximated value function $\tilde{V}^\pi$. We would like to take the $\tilde{V}^\pi$ information learned on a set of training instances $\{I_1, I_2, \ldots, I_N\}$ and use it to guide search on a given new instance I'. But how can we ensure that $\tilde{V}^\pi$ is meaningful across multiple problem instances simultaneously, when the various instances may differ markedly in size, shape, and attainable objective-function value?

The first crucial step is to impose an instance-independent representation on the features $F(x)$, which comprise the input to $\tilde{V}^\pi(F(x))$. In their algorithm for transfer between job-shop scheduling instances (which I will discuss in detail in Section 7.2), Zhang and Dietterich recognize this: they define "a fixed set of summary statistics describing each state, and use these statistics as inputs to the function approximator" [Zhang and Dietterich 98]. As it so happens, almost all the feature sets used with STAGE in this thesis are naturally instance-independent, or can easily be made so by normalization. For example, in Bayes-net structure-finding problems (§4.4), the feature that counts the number of "orphan" nodes can be made instance-independent simply by changing it to the *percentage* of total nodes that are orphans.

The second question concerns normalization of the *outputs* of $\tilde{V}^\pi(F(x))$, which are predictions of objective-function values. In Table 6.4 above, the nine channel routing instances all have quite different solution qualities, ranging from 8 tracks in the case of instance HYC1 to more than 50 tracks in the case of instance HYC6. If we wish to train a single function approximator to make meaningful predictions about the expected solution quality on both instances HYC1 and HYC6, then we must normalize the objective function itself. For example, $\tilde{V}^\pi$ could be trained to predict not the expected reachable Obj value, but the expected reachable percentage above a known lower bound for each instance. Zhang and Dietterich adopt this approach: they heuristically normalize each instance's final job-shop schedule length by dividing it by the difficulty level of the starting state [Zhang and Dietterich 98]. This enables them to train a single neural network over all problem instances.

However, if tight lower bounds are not available, such normalization can be problematic. Consider the following concrete example:

- There are two similar instances, I_1 and I_2, which both have the same true optimal solution, say, $\mathrm{Obj}(x^*) = 130$.

- A single set of features $\mathbf{f}$ is equally good in both instances, promising to lead search to a solution of quality 132 on either.

- The only available lower bounds for the two instances are $b_1 = 110$ and $b_2 = 120$, respectively.

In this example, normalizing the objective functions to report a percentage above the available lower bound would result in a target value of 20% for $\tilde{V}^\pi(\mathbf{f})$ on I_1 and a target value of 10% for $\tilde{V}^\pi(\mathbf{f})$ on I_2. At best, these disparate training values add noise to the training set for $\tilde{V}^\pi$. At worst, they could interact with other inaccurate training set values and make the non-instance-specific $\tilde{V}^\pi$ function useless for guiding search on new instances.

I adopt here a different approach, which eliminates the need to normalize the objective function across instances. The essential idea is to recognize that each individually learned $\tilde{V}^\pi_{I_k}$ function, *unnormalized*, is already suitable for guiding search on the new problem I': the search behavior of an evaluation function is scale- and translation-invariant. The **X-STAGE** algorithm, specified in Table 6.5, combines the knowledge of multiple $\tilde{V}^\pi_{I_k}$ functions not by merging them into a single new evaluation function, but by having them *vote* on move decisions for the new problem I'. Note that after the initial set of value functions has been trained, X-STAGE performs no further learning when given a new optimization problem I' to solve.

Combining $\tilde{V}^\pi_{I_k}$ decisions by voting rather than, say, averaging, ensures that each training instance carries equal weight in the decision-making process, regardless of the range of that instance's objective function. Voting is also robust to "outlier" functions, such as the one learned on instance HYC1 in Table 6.4 above. Such a function's move recommendations will simply be outvoted. A drawback to the voting scheme is that, in theory, loops are possible in which a majority prefers x over x', x' over x'', and x'' over x. However, I have not seen such a loop in practice, and if one did occur, the patience counter PAT would at least prevent X-STAGE from getting permanently stuck.

6.2.3 Experiments

I applied the X-STAGE algorithm to the domains of bin-packing and channel routing. For the bin-packing experiment, I gathered a set of 20 instances from the OR-Library—the same 20 instances studied in Section 4.2. Using the same STAGE parameters given in that section (p. 75), I trained $\tilde{V}^\pi$ functions for all of the 20 except u250_13, and then applied X-STAGE to test performance on the held-out instance. The performance curves of X-STAGE and, for comparison, ordinary STAGE are shown in Figure 6.6. The semilog scale of the plot clearly shows that X-STAGE

X-STAGE$(I_1, I_2, \ldots, I_N, I')$:

Given:

- a set of training problem instances $\{I_1, \ldots, I_N\}$ and a test instance I'.

  Each instance has its own objective function and all other STAGE parameters (see p. 52). It is assumed that each instance's featurizer $F : X \to \Re^D$ maps states to the same number D of real-valued features.

1. **Run STAGE independently on each of the N training instances.**

   This produces a set of learned value functions $\{\tilde{V}^\pi_{I_1}, \tilde{V}^\pi_{I_2}, \ldots, \tilde{V}^\pi_{I_N}\}$.

2. **Run STAGE on the new instance I',** but with STAGE's Step 2c—the step that searches for a promising new starting state for π (see p. 52)—modified as follows: instead of performing hillclimbing on a newly learned $\tilde{V}^\pi$, perform *voting-hillclimbing* on the set of previously learned $\tilde{V}^\pi$ functions. Voting-hillclimbing means simply:

   > Accept a proposed move from state x to state x' if and only if, for a majority of the learned value functions, $\tilde{V}^\pi_{I_k}(F(x')) \leq \tilde{V}^\pi_{I_k}(F(x))$.

   Return the best state found.

TABLE 6.5. The X-STAGE algorithm for transferring learned knowledge to a new optimization instance

reaches good performance levels more quickly than STAGE. However, after only about 10 learning iterations and 10,000 evaluations, the average performance of STAGE exceeds that of X-STAGE. STAGE's $\tilde{V}^{\pi}$ function, finely tuned for the particular instance under consideration, ultimately outperforms the voting-based restart policy generated from 19 related instances.

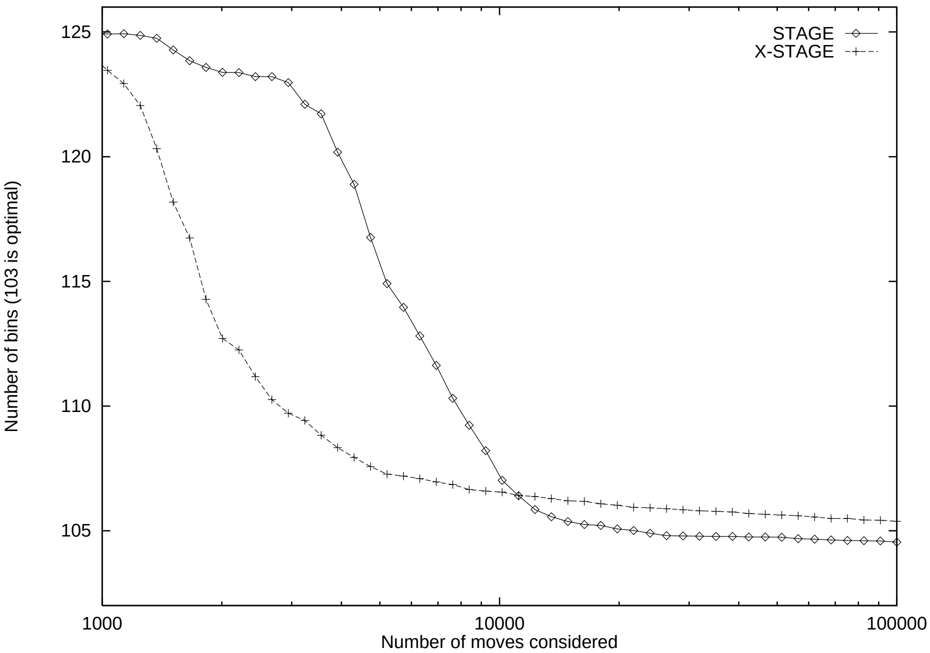

FIGURE 6.6. Bin-packing performance on instance u250_13 with transfer (X-STAGE) and without transfer (STAGE). Note the logarithmic scale of the x-axis.

The channel routing experiment was conducted with the set of 9 instances shown in Table 6.4 above. Again, all STAGE parameters were set as in the experiments of Chapter 4 (see p. 81). I trained $\tilde{V}^{\pi}$ functions for the instances HYC1 ... HYC8, and applied X-STAGE to test performance on instance YK4. The performance curves of X-STAGE and ordinary STAGE are shown in Figure 6.7. Again, X-STAGE reaches good performance levels more quickly than does STAGE. This time, the voting-based restart policy maintains its superiority over the instance-specific learned policy for the duration of the run.

These preliminary experiments indicate that the knowledge STAGE learns during problem-solving can indeed be profitably transferred to novel problem instances. Future work will consider ways of combining previously learned knowledge with new knowledge learned during a run, so as to have the best of both worlds: exploiting

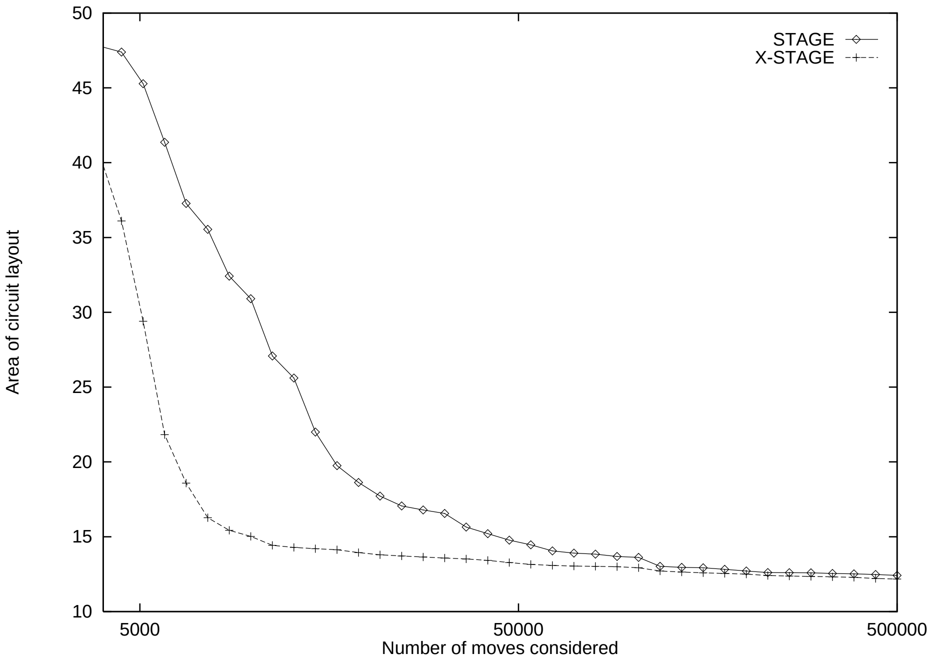

FIGURE 6.7. Channel routing performance on instance YK4 with transfer (X-STAGE) and without transfer (STAGE). Note the logarithmic scale of the x-axis.

general knowledge about a family of instances to reach good solutions quickly, and exploiting instance-specific knowledge to reach the best possible solutions.

6.3 Discussion

This chapter has presented two significant extensions to STAGE: the Least-Squares TD(λ) algorithm for efficient reinforcement learning of $\tilde{V}^\pi$; and the X-STAGE algorithm for transferring $\tilde{V}^\pi$ functions learned on a set of problem instances to new, similar instances. Many further interesting extensions to STAGE are possible. After giving a survey of related work in Chapter 7, I will present a number of as-yet unexplored ideas for STAGE in the concluding chapter.