# Learning Evaluation Functions

Justin A. Boyan

THESIS PROPOSAL

June 17, 1996

*Thesis Committee*
Andrew W. Moore, co-chair
Scott E. Fahlman, co-chair
Tom Mitchell
Thomas G. Dietterich, Oregon Graduate Institute

**Abstract**

Evaluation functions are an essential component of practical search algorithms for optimization, planning and control. Examples of such algorithms include hillclimbing, simulated annealing, best-first search, A*, and alpha-beta. In all of these, the evaluation functions are typically built manually by domain experts, and may require considerable tweaking to work well.

I will investigate the thesis that statistical machine learning can be used to automatically generate high-quality evaluation functions for practical combinatorial problems. The data for such learning is gathered by running trajectories through the search space. The learned evaluation function may be applied either to guide further exploration of the same space, or to improve performance in new problem spaces which share similar features.

Two general families of learning algorithms apply here: reinforcement learning and meta-optimization. The reinforcement learning approach, dating back to Samuel's checkers player [1959] but with more recent successes on backgammon [Tesauro, 1992] and job-shop scheduling [Zhang and Dietterich, 1995], is based on asynchronous dynamic programming with value function approximation. The currently-popular algorithms, Q-learning and TD($\lambda$) with neural networks, run slowly and may even be unstable. I will evaluate several original value-function-approximation algorithms tailored for combinatorial optimization domains.

The meta-optimization approach, which has also been applied to game-playing [Pollack *et al.*, 1996] and combinatorial optimization [Ochotta, 1994], is conceptually simpler: we assume a fixed parametric form for the evaluation function and optimize it directly. These methods, lacking the theoretical advantages of dynamic programming, have been ignored by the reinforcement-learning community; however, recent advances

in local optimization [Moore and Schneider, 1996] may help make them superior to reinforcement learning in practice. My research will develop both methods and evaluate them empirically.

I will apply these techniques to all or most of the following large-scale combinatorial domains: VLSI channel routing; treatment planning for radiation therapy; scheduling a factory production line; configuring the subsystems of the NASA Outer Planet Orbiter; and move selection in the game of backgammon. I will also validate my new algorithms on standard problems from the optimization and control literatures. Finally, a byproduct of this research will be a software package for public use.

# Contents

# 1 Introduction

## 1.1 Background

The use of heuristic *evaluation functions* to guide search through large state spaces is fundamental to the field of Artificial Intelligence. Which state should be visited next in the search for a better, nearer, cheaper goal state? The evaluation function maps features of each state to a real value which assesses the state's promise. For example, in the domain of chess, a classic evaluation function is obtained by summing material advantage weighted by 1 for pawns, 3 for bishops and knights, 5 for rooks, and 9 for queens. The function values provide a basis for decision-making in search. The choice of evaluation function "critically determines search results" [Nilsson, 1980, p.74] in popular algorithms for planning and control ($A^*$), game-playing (alpha-beta), and combinatorial optimization (hillclimbing, simulated annealing).

Evaluation functions have generally been designed by human domain experts. The weights {1,3,5,9} in the chess evaluation function given above summarize the judgment of generations of chess players. In combinatorial optimization domains, such as the Traveling Salesperson Problem [Lin and Kernighan, 1973], the state space consists of legal candidate solutions, and the domain's *objective function* (the function which evaluates the quality of a final solution) can itself serve as a natural evaluation function to guide search. However, to get good optimization results, engineers often spend considerable effort tweaking the coefficients of penalty terms and other additions to their objective function. This excerpt, from a book on VLSI layout by simulated annealing [Wong *et al.*, 1988], is typical:

> Clearly, the objective function to be minimized is the channel width $w$. However, $w$ is too crude a measure of the quality of intermediate solutions. Instead, for any valid partition $\pi$, the following cost function is used:
>
> $$C = w^2 + \lambda_p \cdot p^2 + \lambda_U \cdot U \tag{1}$$
>
> where ... $\lambda_p$ and $\lambda_U$ are constants, ...

In this application, the authors hand-tuned the coefficients and set $\lambda_p = 0.5$, $\lambda_U = 10$. (I will later show a machine-learning procedure which achieved much better performance on this task by assigning, counterintuitively, a *negative* value to $\lambda_U$.) The simulated annealing literature is full of similar examples of evaluation functions being engineered for good performance (e.g. [Cohn, 1992, Szykman and Cagan, 1995]).

## 1.2 Proposal

This thesis will investigate new machine learning algorithms for generating evaluation functions automatically. The algorithms are *simulation-based*: they gather data by simulating trajectories through the search space, and use the outcomes of those trajectories to improve the evaluation function. The learned evaluation function may be applied either to guide further exploration of the same space, or to improve performance in new problem spaces which share similar features.

Two general families of learning algorithms apply here: reinforcement learning and meta-optimization. The reinforcement learning approach, dating back to Samuel's checkers player [Samuel, 1959] but with more recent successes on backgammon [Tesauro, 1992] and job-shop scheduling [Zhang and Dietterich, 1995], is based on approximating a special evaluation function known as the *value function*. The value function is defined to predict the best possible long-term outcome expected from each state; a greedy search guided by it will produce an optimal result. Learning the value function in combinatorially large domains, however, is quite challenging. The currently-popular approximation algorithms, Q-learning and TD($\lambda$) with neural networks, run very slowly and may even be unstable [Boyan and Moore, 1995]. Section 2 of this proposal reviews the literature on value function approximation and describes an original algorithm for more efficient learning. Section 3 focuses specifically on the application of value function approximation to combinatorial optimization problems, and presents another new algorithm.

The second approach, meta-optimization, is conceptually simpler: we assume a fixed parametric form for the evaluation function and optimize it directly. This approach was also used by Samuel [1959] and has been applied in the simulated annealing community [Ochotta, 1994]. Meta-optimization methods, lacking the theoretical advantages of dynamic programming, have been largely ignored by the reinforcement-learning community; however, recent advances in local optimization [Moore and Schneider, 1996] may help make them superior to reinforcement learning in practice. My research will lay out a framework for understanding both methods more formally; investigate improvements in both; and evaluate empirically which approach works better. Section 4 reviews the literature in meta-optimization and outlines the experiments I plan to carry out.

I will apply these techniques to all or most of the following large-scale combinatorial domains: VLSI channel routing; treatment planning for radiation therapy; scheduling a factory production line; configuring the subsystems of the NASA Outer Planet Orbiter; and move selection in the game of backgammon. These applications are described in Section 5. I will also validate my new algorithms on standard problems from the optimization and control literatures. Finally, a byproduct of this research will be a software package for public use.

# 2 Value Function Approximation for Prediction and Control

The optimal *value function* is an evaluation function which encapsulates complete knowledge of the best expected search outcome attainable from each state:

$$V^*(x) = \text{the expected long-term reward starting from } x, \text{ assuming optimal decisions.} \quad (2)$$

Such an evaluation function is ideal in that a greedy local search with respect to $V^*$ will always make the globally optimal move. This section reviews the literature on computing

$V^*(x)$, and motivates and describes our new approximation algorithms for this problem.

## 2.1 Markov Decision Problems

Formally, let our search space be represented as a Markov decision problem (MDP), defined by

- a finite set of states $X$, including a set of start states $S \subset X$;
- a finite set of actions $A$;
- a reward function $R : X \times A \to \Re$, where $R(x,a)$ is the expected immediate reward (or negative cost) for taking action $a$ in state $x$; and
- an action model $P : X \times X \times A \to \Re$, where $P(x'|x,a)$ gives the probability that executing action $a$ in state $x$ will lead to state $x'$.

An agent in an MDP environment observes its current state $x_t$, selects an action $a_t$, and as a result receives a reward and moves probabilistically to another state $x_{t+1}$. The model is flexible enough to represent AI planning problems, stochastic games (e.g. backgammon) against a fixed opponent, and combinatorial optimization search spaces. With natural extensions, it can also represent continuous stochastic control domains, two-player games, and many other problem formulations [Littman and Szepesvári, 1996, Harmon *et al.*, 1995].

Specifying a fixed deterministic policy $\pi : X \to A$ reduces an MDP to a Markov chain. The *policy value function* $V^\pi(x)$ is the expected long-term reward accumulated over trajectories through the chain:

$$V^\pi(x) = \mathop{\mathrm{E}}_{\substack{\text{all trajectories} \\ x, x_1, x_2, \ldots}} \left[ \sum_{t=0}^{\infty} \gamma^t R(x_t, \pi(x_t)) \right].$$

Here, $\gamma \in [0,1]$ is a *discount factor* which determines how important long-term rewards are; my work in this thesis predominantly uses the *undiscounted* case of $\gamma = 1$. The form of $V^\pi$ above is convenient because it satisfies this linear system of *Bellman equations for prediction*:

$$\forall x, \quad V^\pi(x) = R(x, \pi(x)) + \gamma \sum_{y \in X} P(y|x, \pi(x)) V^\pi(y) \tag{3}$$

The optimal solution to an MDP is a policy $\pi^*$ which maximizes $V^\pi(x)$. The policy value function of $\pi^*$ is the optimal value function $V^*$. It satisfies the *Bellman equations for control*:

$$\forall x, \quad V^*(x) = \max_{a \in A} \left[ R(x,a) + \gamma \sum_{y \in X} P(y|x,a) V^*(y) \right] \tag{4}$$

From the value function $V^*$, it is easy to infer the optimal policy: at any state $x$, any action which instantiates the max in Equation 4 is an optimal choice [Bellman, 1957]. This formalizes the notion that $V^*$ is an ideal evaluation function.

For small problems of up to $10^7$ or so discrete states, the value function can be stored in a lookup table and computed by any of a variety of algorithms, including *linear programming*

[Denardo, 1982], *policy iteration* [Howard, 1960], and *value iteration* [Bellman, 1957]. Value iteration (VI) has been the basis for the most important simulation-based algorithms for learning $V^*$, so I focus on it here.

VI computes $V^*$ by repeatedly sweeping over the state space, applying Equation 4 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 the $i^{\text{th}}$ sweep 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 main 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; combinatorial optimization domains as formulated in Section 3 below; 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, for small discrete problems 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 per 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.

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 quantized high-dimensional state spaces characteristic of real-world control tasks, the *curse of dimensionality* makes such enumeration intractable. Computing $V^*$ requires generalization: one natural technique is to encode the states as real-valued feature

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

vectors and to use a function approximator to fit $V^*$ over this feature space. This technique goes by the names *Neuro-Dynamic Programming* [Bertsekas and Tsitsiklis, 1996] and *Value Function Approximation* (VFA) [Boyan *et al.*, 1995].

## 2.2 Literature Review

Any review of the literature on reinforcement learning and evaluation functions must begin with the pioneering work of Arthur Samuel on the game of checkers [Samuel, 1959, Samuel, 1967]. Samuel recognized the worth of the value function, saying that

> ... 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 will occur during actual play. Of course, if one could develop a perfect system of this sort it would be the equivalent of always looking ahead to the end of the game. [Samuel, 1959, p. 219]

Samuel's program incrementally changed the coefficients of an evaluation polynomial so as to make each visited state's value closer to the value obtained from lookahead search.

After Samuel, work in evaluation function learning was sporadic until the 1980s. Christensen [1986] tried replacing Samuel's coefficient-tweaking procedure with least-squares regression, and was able to learn reasonable weights for the chess material-advantage function. Lee and Mahajan [1988] trained a nonlinear evaluation function on expertly-played games, and it played high-quality Othello. But the biggest advance in the state of the art came more recently, when the reinforcement-learning community elaborated the connection between AI search and the Bellman equations [Barto *et al.*, 1989, Watkins, 1989, Sutton, 1990]. This connection had been unexplored despite the publication of an AI textbook by Richard Bellman himself [Bellman, 1978]!

Reinforcement learning's most celebrated success has also been in a game domain, the game of backgammon [Tesauro, 1992, Tesauro, 1994, Boyan, 1992]. Tesauro modified Sutton's TD($\lambda$) 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. An implementation is given in Appendix A.1. When $\lambda = 0$, this algorithm reduces to 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 MDP, rather than along sweeps of the entire state space. Applying TD($\lambda$) with a standard multi-layer perceptron function approximator, Tesauro's program learned an evaluation function which produced expert-level backgammon play. Boyan [1992] replicated Tesauro's results and demonstrated improvements using modular neural networks.

Tesauro's combination of TD($\lambda$) and neural networks has been applied to other domains, including elevator control [Crites and Barto, 1996] and job-shop scheduling [Zhang and Dietterich, 1995]. (I will discuss the scheduling application in detail in Section 3.2.) Nevertheless, it is important to note that when function approximators are used, TD($\lambda$) provides no guarantees of optimality. In the case of uncontrolled Markov chains and linear function approximators, online TD($\lambda$) does converge [Sutton, 1988, Dayan, 1992, Tsitsiklis and 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 Markov Decision Problems, repeatedly applying one-step backups may propagate and enlarge approximation errors, leading to instability [Thrun and Schwartz, 1993, Boyan and Moore, 1995, Gordon, 1995].

## 2.3 VFA in Acyclic Domains: "ROUT"

I 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. I have therefore investigated generalizations of the Dijkstra and DAG-SP algorithms specifically modified to accommodate huge state spaces and value function approximation. My variant of Dijkstra's algorithm, called Grow-Support, was presented in [Boyan and Moore, 1995] and will not be explored further in this thesis. My variant of DAG-SP is an algorithm called ROUT [Boyan and Moore, 1996], which this thesis will develop further.

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$\rightarrow$value} pair is added to a training set for a function approximator. Thus, ROUT's main loop consists of identifying a frontier state; determining its $V^*$ value; and retraining the approximator (see Appendix A.2). 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 a starting state of the MDP.

The HuntFrontierState operation returns a state which with high probability satisfies these properties. It begins with some state $x$ and generates a number of trajectories from $x$, each time checking to see whether all states along the trajectory are self-consistent (i.e., satisfy Equation 4 to some tolerance $\epsilon$). 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 2.3 illustrates how the routine works.

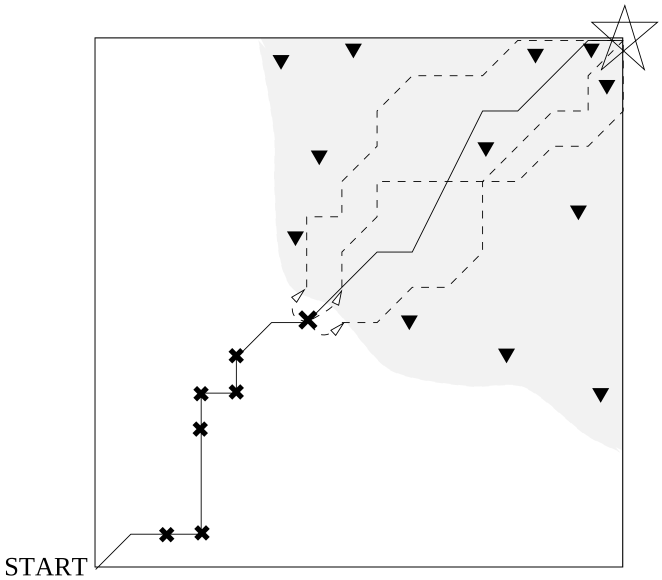


Figure 1: A schematic of ROUT working on an acyclic two-dimensional navigation domain, where the allowable actions are only $\rightarrow$, $\nearrow$, and $\uparrow$. Suppose that ROUT has thus far established training values for $V^*$ at the triangles, and that the function approximator has successfully 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.

The parameters of the ROUT algorithm are $H$, the number of trajectories generated to certify a state's readiness, and $\epsilon$, 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 I used were $H = 20$ and $\epsilon =$ roughly 5% of the range of $V^*$).

In experiments on two medium-sized domains ($\approx 10^6$ states), ROUT learned evaluation functions which were as good or better than those learned by TD($\lambda$) with the same neural-net approximator, and used an order of magnitude less training data to do so. These results are detailed in Appendix B.1.

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. Some function approximators may be especially well-suited to ROUT's required extrapolation from accurate training data, and I plan to explore this. I will also explore algorithmic refinements to make ROUT more robust in general. One such refinement is to adapt the tolerance level $\epsilon$, thereby guaranteeing progress at the expense of accuracy.

With such improvements, I hope to develop ROUT into the algorithm of choice for learning evaluation functions for large-scale acyclic domains. I will validate it empirically on the control problem of factory scheduling (Section 5.3), the game of backgammon[2], and a variety of combinatorial optimization problems formulated as acyclic MDPs as described below.

# 3 Value Function Approximation for Combinatorial Optimization

## 3.1 Combinatorial Optimization

Combinatorial optimization domains, in addition to being more economically important than game domains, have several nice properties that make them amenable to evaluation function learning. First, the simple search techniques in wide use (hillclimbing, simulated annealing) only ever use a single ply of lookahead, making them especially dependent on the evaluation function. Second, the widespread use of manual objective-function tuning clearly indicates the need for and potential benefits of efficient automatic procedures. Finally, like games but unlike most robotic control problems, the complete action model for optimization domains is known. This means we can apply value function approximation algorithms without the added complication of having to learn a model and deal with its inevitable inaccuracies.

Formally, the problem of combinatorial optimization is simply stated: given a finite state space $X$ and an objective function $f : X \to \Re$, find an optimal state $x^* = \mathrm{argmin}_{x \in X} f(x)$. Typically, $X$ is huge, and finding an optimal $x^*$ is intractable. However, there are many

[2]Though backgammon is not strictly acyclic, loops are quite rare, so ROUT may work well anyway.

heuristic algorithms that attempt to exploit $f$'s structure to locate good optima, including hillclimbing, simulated annealing, tabu search, and genetic algorithms. Loosely speaking, these work by imposing a neighborhood relation on the states of $X$ and then searching the graph that results, guided by $f$.

The effectiveness of these search algorithms clearly depends on the shape of $f$ with respect to the graph structure. If $f$ is riddled with local minima or has large regions of constant value (plateaus), then search-based algorithms are likely to fail. For this reason, the most natural $f$ for a problem—the one which evaluates the actual quantity a human would like to minimize—is often *not* a good evaluation function to use for guiding search. Looked at another way, $f$ is most useful in search if it indicates not only which states are good, but also *which states predictably lead to good states.*

In current practice, more complex evaluation functions than $f$, based on features of the state that help predict the search quality, are manually built and tuned. This part of my thesis considers the possibility of automating this task using value function approximation. I consider two methods: Section 3.2 extends the approach of [Zhang and Dietterich, 1995], and Section 3.3 introduces a simpler approach based on predicting search outcomes.

## 3.2 Optimization as an MDP

To my knowledge, the work by Zhang and Dietterich [Zhang and Dietterich, 1995, Zhang, 1996] represents the only application of value function approximation to a combinatorial optimization problem. Since this work is the most closely related to what I propose to do, I will summarize it here in some detail while giving my own analysis of the issues involved.

Zhang and Dietterich attacked the problem of *job-shop scheduling*: given a partially-ordered set of jobs and the resources required by each, assign them start times so as to respect the partial order, meet constraints on simultaneous resource usage, and minimize the total execution time. Following the "repair-based scheduling" paradigm of [Zweben and Fox, 1994], they defined a search over the space of fully-specified schedules which meet the ordering constraints but not necessarily the resource constraints. Search begins at a fixed start state (the "critical path schedule") and proceeds by the application of two types of deterministic operators: Reassign-Pool and Move. To keep the number of instantiated operators small, they only considered operations which would repair the schedule's earliest constraint violation. Search terminates as soon as a violation-free schedule is reached.

The intent of Zhang and Dietterich's work was to learn a value function which captured knowledge about not a single instance of a scheduling problem, but rather a large family of related scheduling instances. Thus, the input to their value function approximator (a neural network) consisted of abstract problem-independent features, such as the percentage of the schedule containing a violation and the mean and standard deviation of certain slack times. Likewise, the ultimate measure of schedule quality which the neural network was learning to predict had to be normalized so that it spanned the same range regardless of problem difficulty. These abstractions are imperfect, since the true $V^*$ certainly differs from instance to instance, and the abstracted problem features lack enough information to actually produce the correct $V^*$ for any instance! Nevertheless, if a reasonable instance-independent approximation can be learned, it is beneficial because the cost of learning can be amortized over the whole family of applicable problems.

In formulating a general combinatorial optimization problem as a Markov decision problem, several issues arise. The first is that, since only the quality of the final solution reached really matters, and (unlike games) the search agent can choose to eventually reach any state from any other state, the undiscounted $V^*$ value of every state should be the same—an agent can eventually reach the global optimum no matter what decision it makes. Figure 2 illustrates this issue on a trivial 5-state domain. Zhang and Dietterich addressed this by adding a tiny penalty for each move; a similar fix is to use a discount factor $\gamma$ slightly less than 1. Another possible solution, which allows undiscounted return to be used safely, is to set an upper limit $L$ on the number of steps in a search trajectory. This gives rise to a second MDP formulation of the optimization problem, where the number of steps remaining is included in the state space as illustrated in Figure 3. Despite the expanded state space, this alternative formulation is appealing:

- The extra steps-to-go feature may be useful for learning evaluation functions that change the search strategy over time.
- Although Zhang and Dietterich's search space for the scheduling domain has natural terminating states—those for which all constraints are satisfied—many problems offer no such distinguished states. The time-bounded formulation provides a simple termination criterion.
- The expanded space allows the MDP transition probabilities to represent simulated-annealing-like approaches in which move choices are stochastic at first, and then gradually become closer and closer to deterministic. Such "jitter" could help make $V^*$ smoother and easier to approximate.
- It makes the search space acyclic, so our ROUT algorithm applies.

One other issue should be noted. In optimization domains where each state has hundreds or thousands of possible neighbors, choosing each greedy action would involve significant computation time to evaluate $V^*$ for each neighbor. A remedy for this is to make only a small random subset of neighbors available from each state. In the extreme case, a single random neighboring state $y$ is chosen with probability $Pr(y|x)$, and the choice becomes simply whether to accept the move or to stay put. With this change, the optimization process is no longer deterministic nor strictly an MDP, but is still a generalized-MDP [Littman and Szepesvári, 1996] with associated Bellman equations:

$$\forall x \in X, \quad V^*(x) = \sum_{y \in X} Pr(y|x) \max_{x' \in \{x,y\}} (R(x \to x') + \gamma V^*(x')) \tag{5}$$

This equation has the same form as the Bellman equation used in Tesauro's backgammon experiments, where a position's value is defined as an expectation (over dice rolls) of a max (over legal moves). And as in the backgammon case, the TD($\lambda$) algorithm may be used to learn this value function with a neural network.

Following Tesauro's methodology, Zhang and Dietterich applied TD($\lambda$) to the scheduling domain. Unlike backgammon, getting things to work required substantial tuning. Experience

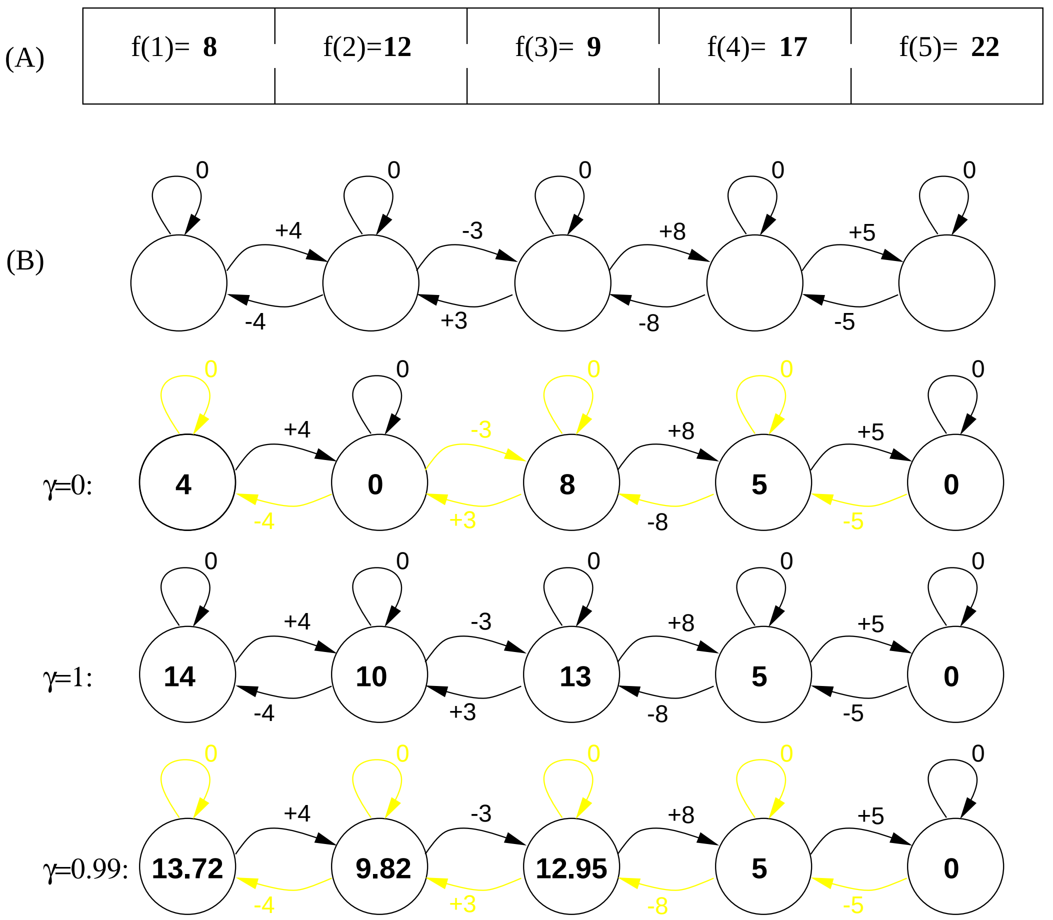


Figure 2: One formulation of combinatorial optimization as an MDP. Part (A) shows a trivial 5-state domain with its objective function $f(x)$ and neighborhood relationship. The rewards of the MDP (B) are constructed such that following a greedy trajectory on the MDP is equivalent to hillclimbing on (A). The lower three diagrams illustrate the optimal value function (bold numbers) and policy (dark arrows) for different values of $\gamma$. $\gamma = 0$ produces the greedy-hillclimbing policy; $\gamma = 1$ produces a policy where all actions appear equally good; $\gamma = 0.99$ produces a policy that leads directly to the global optimum from all states.

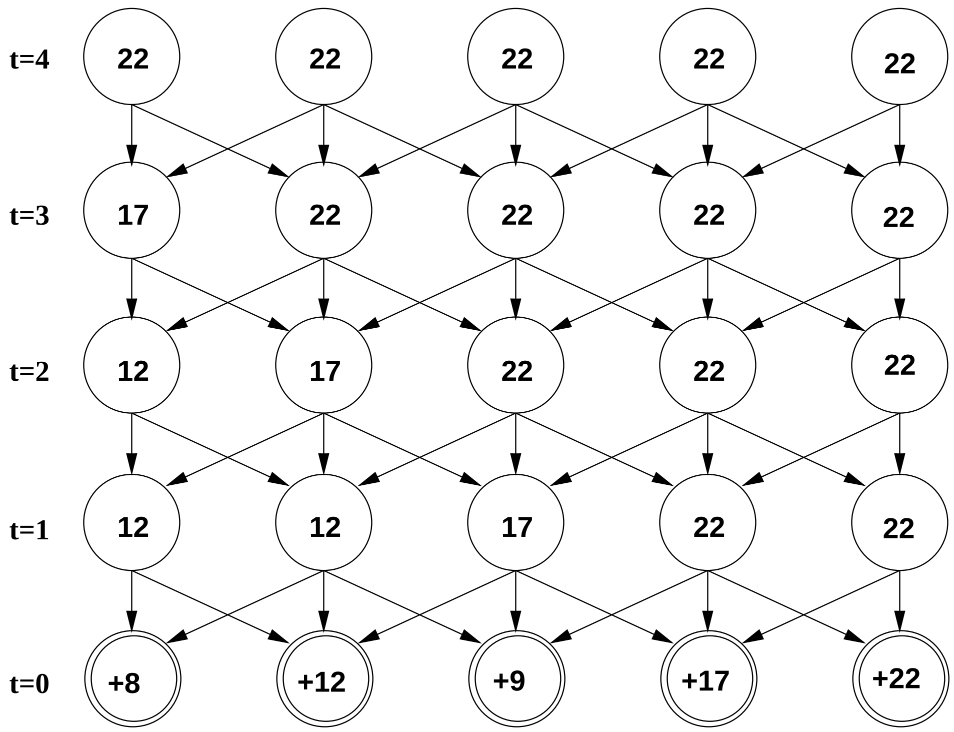


Figure 3: A second MDP formulation of the optimization problem. The state space expands in order to keep track of the number of steps remaining. All rewards are 0 except at the terminal states. A state's $V^*$ value simply predicts the objective function value of the best state that can be reached from $x$ in the amount of time remaining. Greedily following the $V^*$ values (depicted) always leads to the globally optimal $f(x^*) = 22$.

replay [Lin, 1993], Gaussian output representation [Pomerleau, 1991], exploration, and loop-avoidance techniques all had to be carefully calibrated. They trained the value function on a set of small problem instances. Training continued until performance stopped improving on a validation set of other problem instances. The $N$ best-performing networks were saved and used for the comparisons against Zweben's iterative-repair system [Zweben and Fox, 1994], the previously best scheduler. The results showed that searches with the learned evaluation functions produced schedules as good as Zweben's in about half the CPU time.

My thesis will investigate two types of methods based on Value Function Approximation for improving local search in combinatorial optimization problems. One type, like Zhang and Dietterich's, attempts to learn from scratch an approximate value function that can steer a greedy search to a global optimum. I hope to extend Zhang and Dietterich's framework to apply to other domains, using the formulation described above, and to improve learning efficiency by using ROUT in place of TD($\lambda$). The second type of VFA-based method, a simpler approach, is based on predicting and then bootstrapping the performance of a given search algorithm. My experiments to date have centered on this latter approach, and I present those ideas in the following section.

## 3.3 Prediction for Optimization: "STAGE"

**Learning to Predict**

This part of the research begins with a simple observation: the performance of a graph-search-based optimization algorithm depends on the state from which search starts. We can express this dependence in a mapping from starting states $x$ to expected search result:

$$V_A(x) = \sum_{z \in X} P(x \xrightarrow{A} z) f(z) \tag{6}$$

where $A$ refers to the search algorithm being used, and $P(x \xrightarrow{A} z)$ is the probability that a search starting from $x$ will terminate in state $z$. $V_A(x)$ evaluates $x$'s promise as a starting state for $A$. We seek to learn this function using a statistical regression model such as linear regression or multi-layer perceptrons, where states $x$ are represented as real-valued feature vectors. These input features may encode any relevant properties of the domain, including the original objective function $f(x)$ itself.

Training data for supervised learning of $V_A(x)$ may be readily obtained by performing Monte Carlo simulations of $A$ from random starting points. In addition, if the algorithm $A$ behaves as a Markov chain, then all the intermediate states of each simulated trajectory may also be considered alternate "starting points" for that search, and thus used as training data as well. Stochastic hillclimbing trivially meets the Markov requirement: each state $x$ leads stochastically to a neighboring state $y$ for which $f(y) < f(x)$, regardless of the past history of the search; local minima are terminal states of the chain. Simulated annealing, too, is Markovian in the expanded state space $X \times \{\text{temperature}\}$ [van Laarhoven, 1987].

In the Markov-chain cases, $V_A$ is simply the value function of the chain: it predicts the eventual expected outcome from every state. Since value functions satisfy the Bellman equations (Equation 3), algorithms more sophisticated than Monte-Carlo simulation with supervised learning are applicable: in particular, TD($\lambda$) and ROUT both apply.

The state space $X$ is huge, so we cannot expect our simulations to explore any significant fraction of it. Instead, we must depend on good extrapolation from the function approximator if we are to learn $V_A$ accurately. Specifically, we hope that the function approximator will predict good results for unexplored states which share many features with training states that performed well. If $V_A(x)$ is fairly smooth, this hope is reasonable.

### Using the Predictions

The function $V_A(x)$ is designed to reward states which are good starting places for algorithm $A$. We can optimize $V_A$ using any method; most straightforward is to apply the same graph-search procedure as $A$, but to use $V_A$ instead of $f$ as the evaluation function. Whatever optimization procedure is chosen (call it $B$), performing it should produce a state $y$ which is a good starting state for $A$; we can then run $A$ starting from $y$ to complete our search. We use the notation "$B; A$" to represent this concatenation of the two searches. Our "STAGE" algorithm, given in Appendix A.3, interleaves function-approximator training (by either batch fitting or TD($\lambda$)) with running the two-stage learned optimization procedure $B; A$.

The following example illustrates how STAGE can produce better performance than the optimization method which it is learning about. Consider minimizing the one-dimensional function $f(x) = (|x| - 10)\cos(2\pi x)$ over the domain $X = [-10, 10]$, as depicted in Figure 4. Assuming a graph structure on this domain where tiny moves to the left or right are allowed, hillclimbing search clearly leads to a suboptimal local minimum for all but the luckiest of starting points. However, the quality of the local minima reached does correlate strongly with the starting position of $x$, making it possible to learn useful predictions.

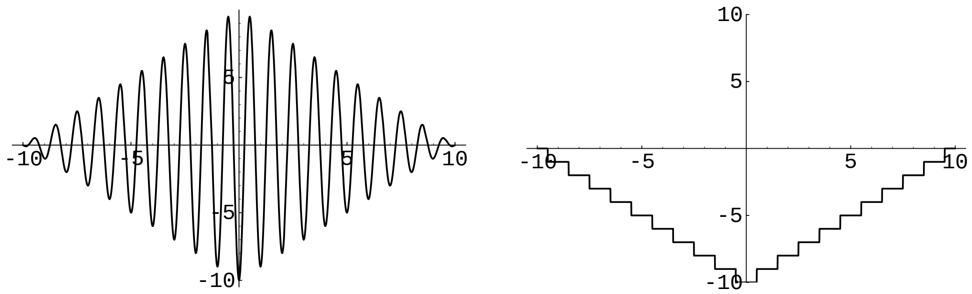


Figure 4: A one-dimensional function minimization domain (left) and the value function which predicts hillclimbing's performance on that domain (right). The value function is approximately linear in the feature $|x|$. TD-learning easily identifies that structure, resulting in an improved evaluation function for guiding search to a global minimum.

We encode this problem for STAGE using a single input feature $|x|$, which makes the value function $V_A$ approximately linear. Not surprisingly, running STAGE with a linear function approximator performs efficiently on this problem: after only a few trajectories through the space, it learns $V_A(x) \approx |x| - 10$. Hillclimbing on $V_A(x)$ rather than $f(x)$, then, leads directly to the global optimum.

This problem is contrived, but we think its essential property—that features of the state help to predict the performance of an optimizer—does indeed hold in many practical domains.

We have gotten excellent preliminary results in the domain of VLSI channel routing. The learned evaluation function soundly outperforms the hand-tuned function mentioned above in Equation 1 (page 3). Details of these results are given in Appendix B.2.

Unlike the work of Zhang and Dietterich, which learns a search strategy from scratch using online value iteration, this method assumes an already-given search strategy for the domain and uses prediction to learn to improve on it. This simpler approach may serve as a better starting point for analysis.

My thesis will develop the STAGE algorithm further and apply it to the optimization domains of Section 5. One interesting direction for exploration is to apply STAGE recursively to itself, resulting in an automatically-generated multi-stage optimization algorithm. These stages, interleaving policy evaluation with execution, are reminiscent of the iterations of policy iteration, and my thesis will elaborate this relationship.

# 4 Direct Meta-Optimization of Evaluation Functions

Standing in contrast to evaluation function learning methods based on approximating the theoretically-optimal $V^*$ are what I call *direct meta-optimization* methods. Such methods assume a fixed parametric form for the evaluation function and optimize it directly with respect to the ultimate objective, sampled by Monte Carlo simulation. In symbols, given an evaluation function $\tilde{V}(x|\vec{w})$ parametrized by weights $\vec{w}$, we seek to learn $\vec{w}$ by directly optimizing the *meta-objective function*

$$M(\vec{w}) = \text{the expected performance of search using evaluation function } \tilde{V}(x|\vec{w}) \; .$$

The functions $\tilde{V}$ learned by such methods are not constrained by the Bellman equations: the evaluations they produce for any given state have no semantic interpretation akin to the definition of $V^*$ in Equation 2 (page 4). The lack of such constraints means that less information for training the function can be gleaned from a simulation run. The temporal-difference goal of explicitly caching values from lookahead search into the static evaluation function is discarded; only the final costs of completed simulation runs are available. For these reasons, the reinforcement-learning community has largely ignored the direct meta-optimization approach.

Nevertheless, I want to give this approach a fair comparison against VFA methods, for several reasons:

- The approach is simple to implement and understand.
- Not having to meet the Bellman constraints may actually make learning easier. For example, even if a domain's correct $V^*$ is very jagged and discontinuous, meta-optimization may discover a quite different, smooth $\tilde{V}$ that performs well. (This point was also made in [Utgoff and Clouse, 1991].)
- Although this approach promises to require a heavy computational burden, it must be admitted that the temporal-difference approaches have not set a high standard in this regard.

- Finally, new stochastic optimization algorithms designed to be data-efficient may make this approach more competitive.

## 4.1 Literature Review

As in the VFA literature, game-playing programs have provided the most common testbed for this approach. Indeed, Samuel's checkers program should be seen as a hybrid VFA/meta-optimization method. In Samuel's algorithm, after the scoring function was tweaked to more closely predict the lookahead values, the new function ("Alpha") was tested in a head-to-head match against the old one ("Beta"). Only if Alpha won were the TD-based tweaks allowed to be kept, replacing Beta. Furthermore, if several consecutive Alpha functions lost, then the biggest coefficient was simply reset to zero—in effect, diversifying the meta-optimization search at the expense of accurate value function approximation.

Genetic-algorithm approaches to game learning generally fall into this category (e.g. [Tunstall-Pedoe, 1991]). Most recently, Pollack et. al. attacked backgammon with a simple hillclimbing approach [Pollack *et al.*, 1996]. Like Samuels, they used Alpha (challenger) and Beta (champion) evaluation functions. Their optimization scheme was purposely simplistic: generate a new Alpha by randomly tweaking Beta's coefficients (the 3980 weights of a neural network), and if Alpha beats Beta in a match, move Beta's coefficients uniformly 5% toward Alpha's. Surprisingly, this procedure developed an excellent backgammon player!

Meta-optimization has also been applied successfully to aid combinatorial optimization. Ochotta's simulated annealing system for synthesizing analog circuit cells made use of a sophisticated cost function parametrized by 46 real numbers [Ochotta, 1994]. These and 10 other parameters of the annealer were optimized using Powell's method as described in [Press *et al.*, 1992]. Each parameter setting was evaluated by summing the mean, median and minimum final results of 200 annealing runs on a small representative problem instance. After several months of real time and four years of CPU time (!), Powell's method produced an evaluation function which performed well and generalized robustly to larger instances.

## 4.2 Meta-Optimization Algorithms

I believe that the computational requirements of the meta-optimization approach can be significantly reduced by the use of stochastic optimization techniques being developed in our group [Moore and Schneider, 1996]. These techniques are designed to optimize functions for which samples are both expensive to gather and potentially very noisy. The meta-objective function $M$ certainly fits this characterization, since sampling $M$ means performing a complete run of simulated annealing (or whatever search algorithm is being optimized) and reporting the final result.

Memory-based stochastic optimization uses the results of all searches conducted thus far to build a global model of $M$. This model is trained with Bayesian locally-weighted polynomial regression, a flexible function approximator which also behaves well while the data set is still small. Whereas Powell's method requires noise-free samples (and thus forced Ochotta to do 200 annealing runs for each parameter setting), Bayesian regression can capture meaningful gradients even without high confidence in the model accuracy.

The learned model can be used in several ways to recommend the next parameter setting to sample. The specific algorithms I will explore are PMAX and IEMAX [Moore and Schneider, 1996]. PMAX's approach is simple: it always recommends sampling the current optimum of the model. IEMAX, patterned after Kaelbling's IE algorithm [Kaelbling, 1990], recommends sampling the parameter setting which optimizes the "optimistic model," i.e. the 95th-percentile values of the model's predicted confidence intervals. In an empirical test on synthetic unimodal five-dimensional optimization problems, PMAX converged to the optimum more quickly than IEMAX, but IEMAX's more persistent exploration will be helpful if $M$ has several local optima.

The approaches based on locally-weighted regression are best-suited for optimizing evaluation functions with a relatively small number of parameters. For learning more complicated evaluation functions, one possibility is to use a cascaded neural network architecture [Fahlman and Lebiere, 1990], which can build a $\tilde{V}$ function of $p$ parameters without ever having to optimize more than $O(\sqrt{p})$ of them at a time. Another possibility is to replace the locally-weighted regression model of $M$ with a global polynomial regression better suited to high-dimensional inputs. Time permitting, I will investigate both these approaches.

# 5 Application Domains

The new algorithms which are the central focus of this thesis will be validated empirically on a broad range of large-scale search problems. This section briefly describes the major domains. I will also report results on testbed domains such as the the children's game of Pig (see Appendix B.1) and small Traveling Salesperson problems.

## 5.1 VLSI Channel Routing

The problem of "Manhattan channel routing" is an important subtask of VLSI circuit design [Wong *et al.*, 1988]. Given two rows of labelled pins across a rectangular channel, we must connect like-labelled pins to one another by placing wire segments into vertical and horizontal tracks (see Figure 5). Segments may cross but not otherwise overlap. The objective is to minimize the area of the channel's rectangular bounding box—or equivalently, to minimize the number of different horizontal tracks needed.

Channel routing is known to be NP-hard, though good heuristics now make optimal solutions obtainable for commercial-grade problems. Nevertheless, it makes a good testbed for optimization algorithms. We follow the development of [Wong *et al.*, 1988, Chapter 4], which attacks the problem with simulated annealing. The operator set is sophisticated, involving manipulations to a partitioning of vertices in an acyclic constraint graph. If the partitioning meets certain additional constraints, then it corresponds to a legal routing, and the number of partitions corresponds to the channel size we are trying to minimize. Appendix B.2 presents our preliminary results with STAGE on this domain.

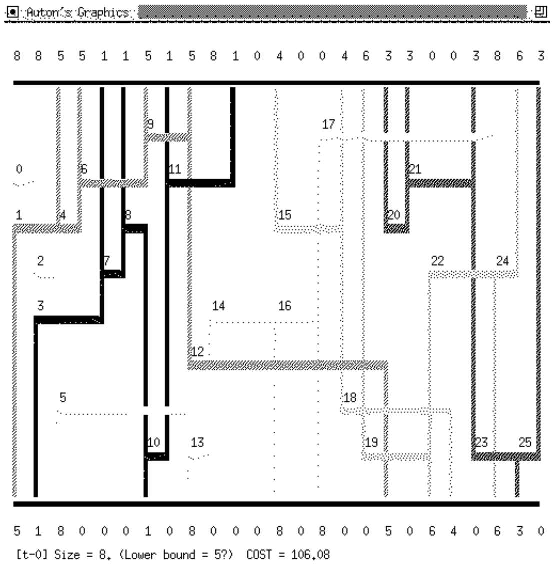


Figure 5: A small channel routing problem.

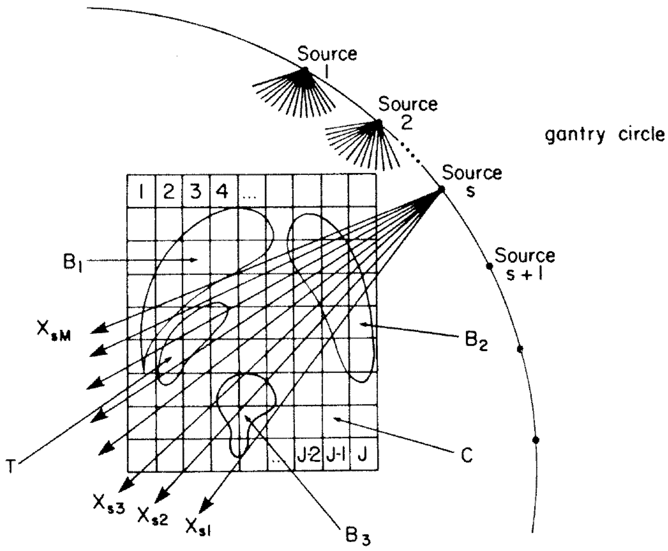


Figure 6: Radiotherapy treatment planning (from [Censor *et al.*, 1988])

## 5.2 Radiotherapy Treatment Planning

Radiation therapy is a widely-used method of treating tumors [Censor *et al.*, 1988]. A linear accelerator which produces a radioactive beam is mounted on a rotating gantry, and the patient is placed so that the tumor is at the center of the beam's rotation. Depending on the exact equipment being used, the beam can be shaped in various ways as it rotates around the patient. A *radiotherapy treatment plan* specifies the beam's shape and intensity at a fixed number of source angles.

A map of the relevant part of the patient's body, with the tumor and all important structures labelled, is available. Also known are good clinical forward models for calculating, from a treatment plan, the distribution of radiation that will be delivered to the patient's tissues. The optimization problem, then, is to produce a treatment plan which meets target radiation doses for the tumor while minimizing damage to sensitive nearby structures.

The current practice is to use simulated annealing for this problem. We will apply the methods of Sections 3 and 4 to learn to improve the performance of simulated annealing over a distribution of problem instances (maps of tumors and surrounding structures). We have also investigated a linear-programming solution to a modified formulation of this problem. The methods of Section 4 may be applicable for optimizing the parameters of the linear program, as well.

## 5.3 Scheduling a Factory Production Line

The "Factory Coating Problem" is a commercially important problem which several members of our lab group are working on. It involves a huge production line that produces about 100 products per second which need to be coated by a special process. Depending on the desired final product, there are about twenty different possible coating processes. The line can be configured with a small number of different coaters. Each coater, when it is not busy and when it is working properly, greedily grabs products from the line. If too many coaters are busy, or too many are temporarily out of service, products reach the end of the line and (literally) fall into an enormous waste bin.

The problem is to find a good schedule for a month of production. The start state is a list of demands for how many of each product must be manufactured (numbers are typically in the tens or hundreds of millions). The task is to find a sequence of configurations over the month to meet production requirements. The objective function combines minimizing wasted products, minimizing the number of configuration changes, and completing the month's production as quickly as possible. Given an approximate model of the performance of individual coaters and their conditional probabilities of temporary breakdowns, the problem can be specified as an MDP with a state space formed as the Cartesian product of $\{$ a 20-dimensional demand space $\} \times \{$ approximately 85,000 coater configurations $\} \times$ $\{$time to end of month$\}$. This MDP is acyclic and thus makes an excellent practical application for ROUT.

## 5.4 Spacecraft Configuration Optimization

A spacecraft can be designed and manufactured more quickly if it is built from a mix-and-match set of subsystems. A spacecraft needs several dozen subsystems, including a propulsion system, an attitude controller, a power bus, a power source, solar panels, flight computers and scientific instruments. NASA's Pluto Express spacecraft project will allow each subsystem to be chosen from an existing supply of pre-designed, pre-manufactured modules, thereby reducing mission costs and increasing mission reliability. The optimization problem is: given a set of mission constraints, choose the best set of modules to implement the subsystems.

This is an interesting combinatorial space because:

- each module decision affects many other module decisions;
- the mission constraints are complex non-linear functions;
- many module decisions may be refined numerically, e.g. the size of a solar panel and bandwidth of a communication system; and
- evaluating a given configuration can be computationally expensive.

It is also an immensely worthwhile problem to solve because of the difficulty and expense of making these decisions manually, and the large cost impact of a successful system.

In conjunction with Steve Chien and researchers at the NASA Jet Propulsion Laboratory, I will formulate this optimization problem as a state-space search and generate evaluation functions that improve search performance. I will also have the opportunity to work on another JPL project, the Integrated Probe Dynamics Tool, which includes a similar component for optimizing the design of a penetrator (a needle-nosed projectile which is sent crashing into a planet to analyze soil).

## 5.5 Backgammon Move Selection

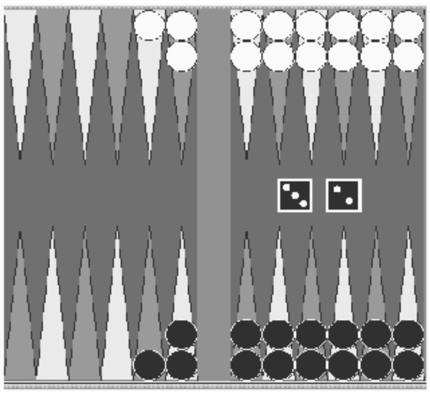

Figure 7: A backgammon endgame position (from [Pollack *et al.*, 1996])

The domain of backgammon fits naturally in this research for several reasons: it was the subject of my Master's thesis [Boyan, 1992], it is the most celebrated and most successful application of value function approximation [Tesauro, 1992], and it also appears to be suitable for direct meta-optimization approaches [Pollack *et al.*, 1996]. Backgammon provides a perfect setting for developing both approaches and putting them in head-to-head competition.

# 6 Summary

## 6.1 Contributions

This thesis will contribute to the state-of-the-art in machine learning, reinforcement learning, heuristic search and combinatorial optimization. Its specific contributions will include:

1. A source reference for researchers interested in automatic learning and tuning of evaluation functions.
2. A new algorithm for learning approximate value functions in large, acyclic Markov decision problems.
3. Formulations for posing general combinatorial optimization problems as Value Function Approximation tasks, extending [Zhang, 1996].
4. A new algorithm for using prediction learning to bootstrap search algorithms.
5. Development of new techniques, based on memory-based stochastic optimization, for direct meta-optimization of evaluation functions.
6. An empirical comparison, on several large-scale applications, of the VFA and direct-optimization approaches.
7. A software product for public distribution that implements the major algorithms for evaluation function learning.

The following research topics are closely related to this thesis but nonetheless *outside* the scope of what I plan to explore:

- Combining learning of the world transition model with learning the evaluation function. Rather, all our algorithms assume that the dynamics of the world are fully known in advance.
- Learning approaches based on analyzing the problem-space operators, such as [Prieditis, 1993], or based on human-labelled training data, such as [Tesauro and Sejnowski, 1989]. All our algorithms learn from simulations and function approximation.
- Theoretical results and analysis. The new algorithms presented here make no assumptions about the function-approximation scheme used, so strong theoretical results are unlikely. My focus will instead be on designing well-motivated algorithms and validating them empirically.

- Direct policy optimization. A reasonable approach for some domains is to bypass the evaluation function altogether, and instead learn a direct mapping from states to actions [Moriarty and Miikkulainen, 1995]. I will not consider this approach.

## 6.2 Timeline

I hope to complete this thesis in a little over a year. Here is my target schedule:

`Jan 96` Basic research: ROUT (done)

`Mar 96` Basic research: optimization as MDP; STAGE (done)

`May 96` Develop initial codebase with applications: VLSI channel routing; 2-D radiotherapy treatment planning (done)

`Jun 96` Propose

`Jul 96` Application focus: factory scheduling

`Aug 96` Work on algorithm refinements

`Sep 96` Application focus: 3-D radiotherapy treatment planning

`Oct 96` Application focus: spacecraft configuration (at JPL)

`Nov 96` Finalize algorithms

`Dec 96` Write release-quality software

`Feb 97` Gather experimental results

`May 97` Write dissertation

`Aug 97` Defend

This thesis and timeline face several risks. First, real applications inevitably introduce thorny issues and software complexity. Fortunately, the simulation software for all of the major domains but one (NASA) is already implemented, and the NASA implementation is to be done by a colleague at JPL. Second, some of the algorithms I have proposed to investigate are quite CPU-intensive. Thoroughly evaluating the cross-product of algorithms $\times$ domains is out of the question; I will have to judiciously choose the subset of experiments that answers the most interesting questions.

# A Algorithms

## A.1 TD($\lambda$) for Control

This implementation of TD($\lambda$) is trajectory-based. For a version of TD($\lambda$) that performs updates after each move, refer to [Sutton, 1987].

**TD**($\lambda$, 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:

$$\text{targ}_i := \begin{cases} r_T \text{ (the terminal reward)} & \text{if } i = T \\ r_i + \lambda\gamma \cdot \text{targ}_{i+1} + (1-\lambda)F(x_{i+1}) & \text{otherwise} \end{cases}$$

   update $F$'s weights by delta rule: $\Delta w := \alpha(\text{targ}_i - F(x_i))\nabla_w F(x_i)$ ;

   end

## A.2 ROUT

Here are the ROUT algorithm and HuntFrontierState subroutine, as described in Section 2.3.

**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 $x$.

## A.3 STAGE

Here is the STAGE optimization algorithm main loop, along with two alternative subroutines for training $V_A$, as described in Section 3.3.

---

**STAGE**(Algorithm $A$, Objective-function $f(x)$):
  /* *Assumption: $A(v)$ is a graph-search algorithm which, given any evaluation function $v(x)$, acts as a Markov chain over the graph.* */
  repeat:
    run $A(f)$, producing a trajectory $T = \{x_0 \to x_1 \to x_2 \to \cdots \to x_{\mathrm{length(T)}}\}$;
    Update_Fitter_from_Traj($V_{A(f)}(x), T, f(x_{\mathrm{length(T)}})$);
    run the two-stage optimization procedure: $A(V_{A(f)})$; $A(f)$, and print result.
  until results stop improving.

---

**Update_Fitter_From_Traj_by_TD**$\lambda$(Fitter $V$, Trajectory $T$, result-value $z$):
  /* *Assumes that function approximator $V$ is parametrized by weight vector $w$.* */
  for $i$ := length(T) downto 0, do:

$$\mathrm{targ}_i := \begin{cases} z \text{ (the terminal reward)} & \text{if } i = \mathrm{length(T)} \\ \lambda \cdot \mathrm{targ}_{i+1} + (1-\lambda)V(x_{i+1}) & \text{otherwise} \end{cases}$$

    update $V$'s weights by delta rule: $\Delta w := \alpha(\mathrm{targ}_i - V(x_i))\nabla_w V(x_i)$ ;
  end.

---

**Update_Fitter_From_Traj_by_Batch_Fit**(Fitter $V$, Trajectory $T$, result-value $z$):
  /* *Assumes that $V$ stores all data it has ever been trained on.* */
  for $i$ := 0 to length(T), do:
    Add training pair to $V$'s memory: $\{x_i \mapsto z\}$;
  Re-train $V$ from the updated training set.

---

# B Preliminary Results

## B.1 ROUT

I present here results with ROUT on three domains: a prediction task, a two-player dice game, and a $k$-armed bandit problem. For all problems, I compare ROUT's performance with that of TD($\lambda$) given the equivalent function approximator. I 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 8.

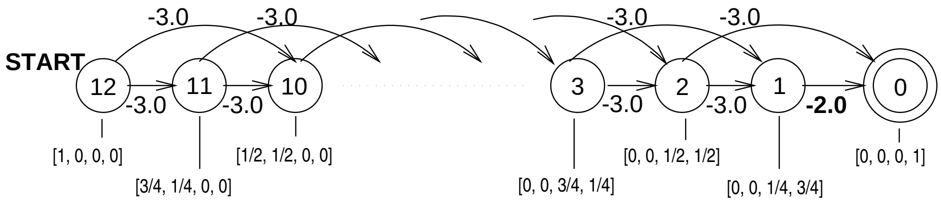


Figure 8: The Hopworld Markov chain. Each state is represented by a four-element feature vector as shown. The function approximator is linear.

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 1. 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 generalized Bellman equations [Littman and Szepesvári, 1996] similar to Equation 4. 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]

| 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 1: 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.)

The Pig results are charted in Table 1 and graphed in Figure 9. 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 vertical axis measures performance in expected points per game against the minimax optimal player, where +1 point is awarded for a win and $-1$ for a loss.) 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 Fristedt, 1985]. While an optimal solution in the infinite-horizon case can be

[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 these comparative experiments, however, we used linear or neural net fits for both algorithms.

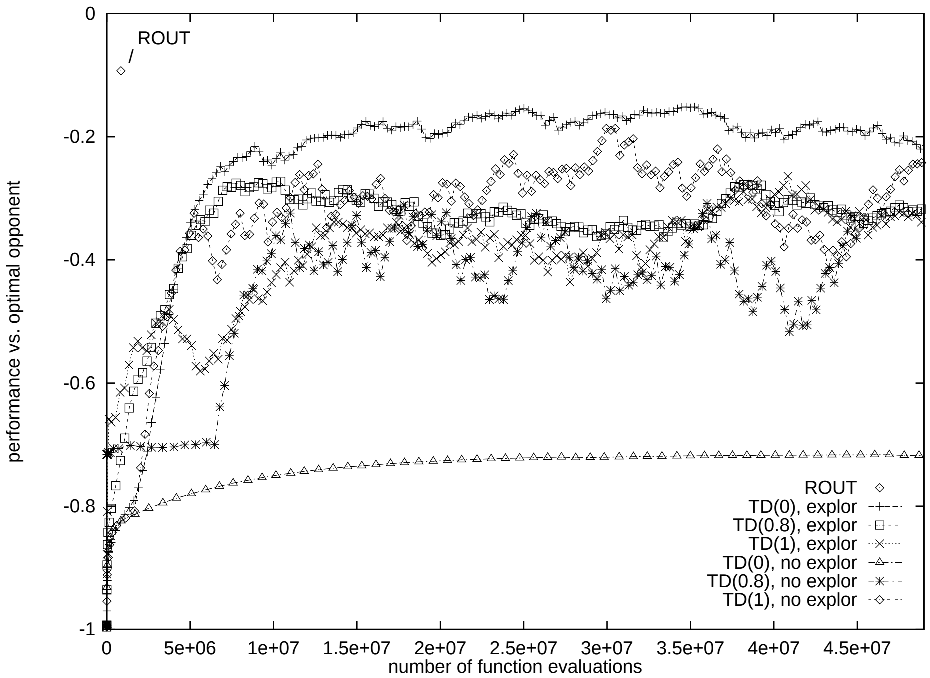


Figure 9: Performance of Pig policies learned by TD and ROUT. ROUT's performance is marked by a single diamond at the top left of the graph.

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 [#succ$_{arm1}$, #fail$_{arm1}$, #succ$_{arm2}$, #fail$_{arm2}$, #succ$_{arm3}$, #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 1. 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.

## B.2 STAGE

In their simulated-annealing approach to the channel-routing problem (described in Section 5.1, Wong et. al. say,

> Clearly, the objective function to be minimized is the channel width $w$. However, $w$ is too crude a measure of the quality of intermediate solutions. Instead,

for any valid partition $\pi$, the following cost function is used:

$$C = w^2 + \lambda_p \cdot p^2 + \lambda_U \cdot U \tag{7}$$

> where $p$ is the longest path length of $G_\pi$ [a graph induced by the partitioning], both $\lambda_p$ and $\lambda_U$ are constants, and ... $U = \sum_{i=1}^{w} u_i^2$, where $u_i$ is the fraction of track $i$ that is unoccupied. [Wong *et al.*, 1988]

They hand-tuned the coefficients and set $\lambda_p = 0.5, \quad \lambda_U = 10$.

To apply STAGE to this problem, we began with not the contrived function $C$ but the natural objective function $f(x) = w$. The additional objective function terms used in Eq. 7, $p$ and $U$, along with $w$ itself, were given as the three input features to STAGE's function approximator.

Table 2 summarizes the results we have obtained thus far, using STAGE to predict and improve upon the results of stochastic hillclimbing optimization on a large channel-routing problem (YK4). For prediction, we have used simple models: linear regression in the batch-update case, and backprop on MLPs (2 hidden units) in the TD($\lambda$) case.

| ALGORITHM | Performance best, mean | typical traj. len taken / considered |
|---|---|---|
| (*) Global optimum for problem | 10 | |
| (A) Hillclimbing, patience=100 (N=30) | 44, 52.4 | 93 / 696 |
| (B) Hillclimbing, patience=1000 (N=30) | 43, 47.9 | 99 / 2591 |
| (C) Simulated annealing, $f(x) = w^2 + 0.5p^2 + 10U$ (N=3) | 35, 35 | 3836 / 126,202 |
| (D) Simulated annealing, $f(x) = w$ (N=3) | 14, 14.7 | 163,401 / 279,073 |
| (E) Modified hillclimbing, patience=1000 (N=5) | 19, 19.8 | 7765 / 11,342 |
| (F) STAGE, TD(0.6) + backprop (N=3) | 14, 22.7 | 388 / 2779 |
| (G) STAGE, batch linear regression (N=28) | 13, 17.1 | 340 / 2293 |
| (H) STAGE, random fit (for validation) | 50, 55 | 190 / 1704 |

Table 2: Results on routing problem YK4. The first pair of columns gives the best and mean channel size attained over $N$ runs of each algorithm. The second pair counts the steps taken in a typical search trajectory from the starting state to a local optimum (in the case of STAGE, *after* learning).

Our results provide some interesting insights. Experiments (A) and (B) show that hillclimbing easily gets stuck in very poor local optima, even when the "patience" parameter is set high (allowing it to evaluate more random moves before terminating). Experiment (C) shows that simulated annealing, as used in [Wong *et al.*, 1988], does considerably better. Very surprisingly, when given plenty of computing time, the annealer of experiment (D) does better still. It seems that the "crude" evaluation function $f(x) = w$ allows simulated annealing to effectively do a random walk along the ridge of all solutions of equal cost $w$, and given enough time it will fortuitously find a hole in the ridge. Hillclimbing modified to accept equi-cost moves performed nearly as well (E).

The hillclimber we gave to STAGE for learning was the fast but weakly-performing one of experiment (A). The results show STAGE learned to optimize superbly, not only bettering the performance of (A) but quickly finding solutions as good as those found by the long simulated annealing runs. The batch fits with linear regression (G) were especially impressive, often producing evaluation functions that performed well after only two or three trajectories of training. This seems too good to be true; did STAGE work according to its design or did it somehow "cheat"?

We considered and eliminated several hypotheses. (1) Since STAGE concatenates two hillclimbing trajectories, perhaps it simply has twice as much time to explore. This explanation is rejected by experiment (B). (2) The function approximator may simply be smoothing $f(x)$, which helps eliminate local minima and plateaus. No, we tried training $V(x)$ to smooth $f(x)$ directly and running $A(V); A(f)$, but this failed to produce improvement. We also trained $V(x)$ to fit randomly-generated training values from the same range as those generated by STAGE; again, this failed to improve over plain one-stage hillclimbing (Experiment H). (3) The input features happen to be excellent, providing an easy answer. This explanation turns out to be correct; I discuss how at the end of the next section.

**Transfer**

When linear generalization from the input features does not suffice to produce good performance, the cost of generating training data is likely to make STAGE inferior to non-learning optimization algorithms on any given problem instance. However, if the input features can be made suitably *instance-independent*, then the cost of training an evaluation function can be amortized over the whole family of applicable instances. This was exactly the tradeoff found to be profitable in [Zhang, 1996], as discussed in Section 3.2.

To investigate how problem-dependent learned evaluation functions are in channel routing, we re-ran experiment (G) on a suite of eight problems from the literature [Chao and Harper, 1996]. Table 3 summarizes the results and gives the coefficients of the linear evaluation function learned (independently) for each problem.

| Problem instance | lower bound | best hillcl. | best STAGE | learned coefficients $< 1, U, p, w >$ |
|---|---|---|---|---|
| YK4 | 10 | 41 | 14 | $< -7.55, -141.2, 7.94, 141.7 >$ |
| HYC1 | 8 | 8 | 8 | $< -0.19, -2.12, 0.86, 2.36 >$ |
| HYC2 | 9 | 9 | 9 | $< 0.05, -2.78, 0.34, 2.66 >$ |
| HYC3 | 11 | 14 | 12 | $< -0.09, -5.06, 0.38, 5.02 >$ |
| HYC4 | 20 | 49 | 30 | $< -3.96, -49.62, 0.33, 49.14 >$ |
| HYC5 | 35 | 47 | 41 | $< -1.68, -7.30, 1.18, 7.08 >$ |
| HYC6 | 50 | 75 | 59 | $< -1.38, -9.59, 0.72, 9.48 >$ |
| HYC7 | 39 | 87 | 83 | $< -5.35, -49.26, -1.38, 48.03 >$ |
| HYC8 | 21 | 55 | 33 | $< -0.79, -32.96, 0.11, 32.78 >$ |

Table 3: STAGE results on eight problems from [Chao and Harper, 1996].

The similarities among the learned evaluation functions are striking. Like the hand-tuned cost function $C$ of [Wong *et al.*, 1988] (Equation 7), all the STAGE-learned cost functions assigned a relatively small positive weight to feature $p$, except on instance HYC7 (where STAGE's learning helped least). Unlike function $C$, they all assigned a large *negative* weight to feature $U$. Since the behavior of a linear evaluation function is invariant under linear transformations, the similarity of these functions suggests that transfer between problem instances will be fruitful. As a first test of transfer, I ran STAGE on HYC7 using the value function learned from HYC6. This did indeed help, producing a solution of size 71 instead of 83.

The assignment of a negative coefficient to $U$ is surprising, because $U$ measures the sparsity of the horizontal tracks. $U$ correlates strongly positively with the objective function to be minimized; a term of $-U$ in the evaluation function ought to pull the search toward terrible solutions in which each subnet occupies its own track. However, the positive coefficient on $w$ cancels out this bias, and in fact a proper balance between the two terms can be shown to bias search toward solutions with an *uneven* distribution of track sparsities. Although this characteristic is not itself the mark of a high-quality solution, it does help lead hillclimbing search to high-quality solutions. STAGE successfully discovered and exploited this predictive combination of features.

# References


[Barto *et al.*, 1989] A. Barto, R. Sutton, and C. Watkins. Learning and sequential decision making. Technical Report COINS 89-95, University of Massachusetts, 1989.

[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.

[Bellman, 1978] R. Bellman. *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company, 1978.

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

[Bertsekas and Tsitsiklis, 1996] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[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.

[Boyan and Moore, 1996] J. A. Boyan and A. W. Moore. Learning evaluation functions for large acyclic domains. In L. Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference*. Morgan Kaufmann, 1996. (To appear).

[Boyan *et al.*, 1995] J. A. Boyan, A. W. Moore, and R. S. Sutton, editors. *Proceedings of the Workshop on Value Function Approximation*, Machine Learning Conference, July 1995. CMU-CS-95-206. Web: http://www.cs.cmu.edu/~reinf/ml95/.

[Boyan, 1992] J. A. Boyan. Modular neural networks for learning context-dependent game strategies. Master's thesis, Engineering Department, Cambridge University, 1992.

[Censor *et al.*, 1988] Y. Censor, M. D. Altschuler, and W. D. Powlis. A computational solution of the inverse problem in radiation-therapy treatment planning. *Applied Mathematics and Computation*, 25:57–87, 1988.

[Chao and Harper, 1996] Heng-Yi Chao and Mary P. Harper. An efficient lower bound algorithm for channel routing. *Integration: The VLSI Journal*, 1996. (to appear).

[Christensen, 1986] J. Christensen. Learning static evaluation functions by linear regression. In T. Mitchell, J. Carbonell, and R. Michalski, editors, *Machine learning: A guide to current research*, pages 39–42. Kluwer, Boston, 1986.

[Cohn, 1992] J. M. Cohn. *Automatic Device Placement for Analog Cells in KOAN*. PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, February 1992.

[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($\lambda$) for general $\lambda$. *Machine Learning*, 8(3/4), May 1992.

[Denardo, 1982] E. Denardo. *Dynamic Programming: Models and Applications*. Prentice-Hall, Inc., 1982.

[Fahlman and Lebiere, 1990] S. Fahlman and C. Lebiere. The Cascade-Correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

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

[Harmon *et al.*, 1995] M. Harmon, L. Baird, and A. H. Klopf. Advantage updating applied to a differential game. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances In Neural Information Processing Systems 7*. MIT Press, 1995.

[Howard, 1960] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press and John Wiley & Sons, 1960.

[Kaelbling, 1990] L. P. Kaelbling. *Learning in Embedded Systems.* PhD thesis, Stanford University, Department of Computer Science, 1990.

[Lee and Mahajan, 1988] K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36, 1988.

[Lin and Kernighan, 1973] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[Lin, 1993] L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993.

[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.* Morgan Kaufmann, 1996. (To Appear).

[Moore and Schneider, 1996] A. W. Moore and J. Schneider. Memory-based stochastic optimization. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Neural Information Processing Systems 8.* MIT Press, 1996.

[Moriarty and Miikkulainen, 1995] D. Moriarty and R. Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3–4):195–209, 1995.

[Nilsson, 1980] N.J. Nilsson. *Principles of Artificial Intelligence.* McGraw-Hill, 1980.

[Ochotta, 1994] E. Ochotta. *Synthesis of High-Performance Analog Cells in ASTRX/OBLX.* PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, April 1994.

[Pollack *et al.*, 1996] J. Pollack, A. Blair, and M. Land. Coevolution of a backgammon player. In C.G. Langton, editor, *Proceedings of Artificial Life 5.* MIT Press, 1996. (to appear).

[Pomerleau, 1991] D. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3, January 1991.

[Press *et al.*, 1992] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[Prieditis, 1993] A. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[Samuel, 1959] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[Samuel, 1967] A. L. Samuel. Some studies in machine learning using the game of checkers II—Recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[Sutton, 1987] R. S. Sutton. Implementation details of the TD($\lambda$) procedure for the case of vector predictions and backpropagation. Technical Note TN87-509.1, GTE Laboratories, May 1987.

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

[Sutton, 1990] 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*. Morgan Kaufmann, 1990.

[Szykman and Cagan, 1995] S. Szykman and J. Cagan. A simulated annealing-based approach to three-dimensional component packing. *ASME Journal of Mechanical Design*, 117, June 1995.

[Tesauro and Sejnowski, 1989] G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39, 1989.

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

[Tesauro, 1994] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[Thrun and Schwartz, 1993] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, 1993.

[Tsitsiklis and 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.

[Tunstall-Pedoe, 1991] W. Tunstall-Pedoe. Genetic algorithms optimizing evaluation functions. *ICCA Journal*, 14(3):119–128, 1991.

[Utgoff and Clouse, 1991] P. Utgoff and J. Clouse. Two kinds of training information for evaluation function learning. In *Proceedings of AAAI*, 1991.

[van Laarhoven, 1987] P. van Laarhoven. *Simulated Annealing: Theory and Applications*. Kluwer Academic, 1987.

[Watkins, 1989] C. Watkins. *Learning From Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[Wong *et al.*, 1988] D. F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic, 1988.

[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.

[Zhang, 1996] W. Zhang. *Reinforcement Learning for Job-Shop Scheduling*. PhD thesis, Oregon State University, 1996.

[Zweben and Fox, 1994] M. Zweben and M.S. Fox. *Scheduling and Rescheduling with Iterative Repair*. Morgan Kaufmann, 1994.