Learning Evaluation Functions for Global Optimization

by

Justin Andrew Boyan

A Dissertation Submitted to the Faculty of the

Computer Science Department

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the School of Computer Science

Carnegie Mellon University

August 1, 1998

Dedication

To my grandfather,
Ara Boyan, 1906–93

Abstract

In complex sequential decision problems such as scheduling factory production, planning medical treatments, and playing backgammon, optimal decision policies are in general unknown, and it is often difficult, even for human domain experts, to manually encode good decision policies in software. The reinforcement-learning methodology of "value function approximation" (VFA) offers an alternative: systems can learn effective decision policies autonomously, simply by simulating the task and keeping statistics on which decisions lead to good ultimate performance and which do not. This thesis advances the state of the art in VFA in two ways.

First, it presents three new VFA algorithms, which apply to three different restricted classes of sequential decision problems: Grow-Support for deterministic problems, ROUT for acyclic stochastic problems, and Least-Squares TD(λ) for fixed-policy prediction problems. Each is designed to gain robustness and efficiency over current approaches by exploiting the restricted problem structure to which it applies.

Second, it introduces STAGE, a new search algorithm for general combinatorial optimization tasks. STAGE learns a problem-specific heuristic evaluation function as it searches. The heuristic is trained by supervised linear regression or Least-Squares TD(λ) to predict, from features of states along the search trajectory, how well a fast local search method such as hillclimbing will perform starting from each state. Search proceeds by alternating between two stages: performing the fast search to gather new training data, and following the learned heuristic to identify a promising new start state.

STAGE has produced good results (in some cases, the best results known) on a variety of combinatorial optimization domains, including VLSI channel routing, Bayes net structure-finding, bin-packing, Boolean satisfiability, radiotherapy treatment planning, and geographic cartogram design. This thesis describes the results in detail, analyzes the reasons for and conditions of STAGE's success, and places STAGE in the context of four decades of research in local search and evaluation function learning. It provides strong evidence that reinforcement learning methods can be efficient and effective on large-scale decision problems.

Acknowledgments

This dissertation represents the culmination of my six years of graduate study at CMU and of my formal education as a whole. Many, many people helped me achieve this goal, and I would like to thank a few of them in writing. Readers who disdain sentimentality may wish to turn the page now!

First, I thank my primary advisor, Andrew Moore. Since his arrival at CMU one year after mine, Andrew has been a mentor, role model, collaborator, and friend. No one else I know has such a clear grasp of what really matters in machine learning and in computer science as a whole. His long-term vision has helped keep my research focused, and his technical brilliance has helped keep it sound. Moreover, his patience, ability to listen, and innate personal generosity will always inspire me. Unlike the stereotypical professor who has his graduate students do the work and then helps himself to the credit, Andrew has a way of listening to your half-baked brainstorming, reformulating it into a sensible idea, and then convincing you the idea was yours all along. I feel very lucky indeed to have been his first Ph.D. student!

My other three committee members have also been extremely helpful. Scott Fahlman, my co-advisor, helped teach me how to cut to the heart of an idea and how to present it clearly. When my progress was slow, Scott always encouraged me with the right blend of carrots and sticks. Tom Mitchell enthusiastically supported my ideas while at the same time asking good, tough questions. Finally, Tom Dietterich has been a wonderful external committee member—from the early stages, when his doubts about my approach served as a great motivating challenge, until the very end, when his comments on this document were especially thorough and helpful.

Three friends at CMU—Michael Littman, Marc Ringuette, and Jeff Schneider—also contributed immeasurably to my graduate education by being my regular partners for brainstorming and arguing about research ideas. I credit Michael with introducing me to the culture of computer science research, and also thank him for his extensive comments on a thesis draft.

Only in the past year or two have I come to appreciate how amazing a place CMU is to do research. I would like to thank all my friends in the Reinforcement Learning group here, especially Leemon Baird, Shumeet Baluja, Rich Caruana, Scott Davies, Frank Dellaert, Kan Deng, Geoff Gordon, Thorsten Joachims, Andrew McCallum, Peter Stone, Astro Teller, and Belinda Thom. Special thanks also go to my officemates over the years, including Lin Chase, Fabio Cozman, John Hancock, and Jennie Kay.

The CMU faculty, too, could not have been more supportive. Besides my committee members, the following professors all made time to discuss my thesis and assist my research: Avrim Blum, Jon Cagan, Paul Heckbert, Jack Mostow, Steven Rudich, Rob Rutenbar, Sebastian Thrun, Mike Trick, and Manuela Veloso. I would also like to thank the entire Robotics Institute for allowing me access to over one hundred of their workstations for my 40,000+ computational experiments!

The third major part of what makes the CMU environment so amazing is the excellence of the department administration. I would particularly like to thank Sharon Burks, Catherine Copetas, Jean Harpley, and Karen Olack, who made every piece of paperwork a pleasure (well, nearly), and who manage to run a big department with a personal touch.

Outside CMU, I would like to thank the greater reinforcement-learning and machine-learning communities—especially Chris Atkeson, Andy Barto, Wray Buntine, Tom Dietterich, Leslie Kaelbling, Pat Langley, Sridhar Mahadevan, Satinder Singh, Rich Sutton, and Gerry Tesauro. These are leaders of the field, yet they are all wonderfully down-to-earth and treat even greenhorns like myself as colleagues.

Finally, I thank the following institutions that provided financial support for my graduate studies: the Winston Churchill Foundation of the United States (1991–92); the National Defense Science and Engineering Graduate fellowship (1992–95); National Science Foundation grant IRI-9214873 (1992–94); the Engineering Design Research Center (EDRC), an NSF Engineering Research Center (1995–96); the Pennsylvania Space Grant fellowship (1995–96); and the NASA Graduate Student Researchers Program (GSRP) fellowship (1996–98). I thank Steve Chien at JPL for sponsoring my GSRP fellowship.

* * *

On a more personal level, I would like to thank my closest friends during graduate school, who got me through all manner of trials and tribulations: Marc Ringuette, Michael Littman, Kelly Amienne, Zoran Popović, Lisa Falenski, and Jeff Schneider. Peter Stone deserves special thanks for making possible my trips to Brazil and Japan in 1997. I also thank Erik Winfree and all my CTY friends in the Mattababy Group for their constant friendship.

Second, I would like to thank the teachers and professors who have guided and inspired me throughout my education, including Barbara Jewett, Paul Sally, Stuart Kurtz, John MacAloon, and Charles Martin.

Finally, I would like to thank the two best teachers I know: my parents, Steve and Kitty Boyan. They taught me the values of education and thoughtfulness, of moderation and balance, and of love for all people and life. To thank them properly—for all the opportunities and freedoms they provided for me, and for all their wisdom and love—would fill the pages of another book as long as this one.

Table of Contents

1 Introduction **13**
- 1.1 Motivation: Learning Evaluation Functions 13
- 1.2 The Promise of Reinforcement Learning 14
- 1.3 Outline of the Dissertation 16

2 Learning Evaluation Functions for Sequential Decision Making **19**
- 2.1 Value Function Approximation (VFA) 19
  - 2.1.1 Markov Decision Processes 19
  - 2.1.2 VFA Literature Review 22
  - 2.1.3 Working Backwards 26
- 2.2 VFA in Deterministic Domains: "Grow-Support" 27
- 2.3 VFA in Acyclic Domains: "ROUT" 29
  - 2.3.1 Task 1: Stochastic Path Length Prediction 33
  - 2.3.2 Task 2: A Two-Player Dice Game 33
  - 2.3.3 Task 3: Multi-armed Bandit Problem 34
  - 2.3.4 Task 4: Scheduling a Factory Production Line 37
- 2.4 Discussion 40

3 Learning Evaluation Functions for Global Optimization **43**
- 3.1 Introduction 43
  - 3.1.1 Global Optimization 43
  - 3.1.2 Local Search 45
  - 3.1.3 Using Additional State Features 46
- 3.2 The "STAGE" Algorithm 49
  - 3.2.1 Learning to Predict 49
  - 3.2.2 Using the Predictions 51
- 3.3 Illustrative Examples 53
  - 3.3.1 1-D Wave Function 53
  - 3.3.2 Bin-packing 56
- 3.4 Theoretical and Computational Issues 57
  - 3.4.1 Choosing π 59
  - 3.4.2 Choosing the Features 65
  - 3.4.3 Choosing the Fitter 66
  - 3.4.4 Discussion 70

4 STAGE: EMPIRICAL RESULTS . . . 71
- 4.1 Experimental Methodology . . . 72
  - 4.1.1 Reference Algorithms . . . 72
  - 4.1.2 How the Results are Tabulated . . . 73
- 4.2 Bin-packing . . . 74
- 4.3 VLSI Channel Routing . . . 81
- 4.4 Bayes Network Learning . . . 86
- 4.5 Radiotherapy Treatment Planning . . . 92
- 4.6 Cartogram Design . . . 95
- 4.7 Boolean Satisfiability . . . 97
  - 4.7.1 WALKSAT . . . 97
  - 4.7.2 Experimental Setup . . . 99
  - 4.7.3 Main Results . . . 101
  - 4.7.4 Follow-up Experiments . . . 103
- 4.8 Boggle Board Setup . . . 104
- 4.9 Discussion . . . 108

5 STAGE: ANALYSIS . . . 109
- 5.1 Explaining STAGE's Success . . . 109
  - 5.1.1 $\tilde{V}^\pi$ versus Other Secondary Policies . . . 110
  - 5.1.2 $\tilde{V}^\pi$ versus Simple Smoothing . . . 112
  - 5.1.3 Learning Curves for Channel Routing . . . 115
  - 5.1.4 STAGE's Failure on Boggle Setup . . . 119
- 5.2 Empirical Studies of Parameter Choices . . . 119
  - 5.2.1 Feature Sets . . . 120
  - 5.2.2 Fitters . . . 126
  - 5.2.3 Policy π . . . 129
  - 5.2.4 Exploration/Exploitation . . . 132
  - 5.2.5 Patience and ObjBound . . . 134
- 5.3 Discussion . . . 135

6 STAGE: EXTENSIONS . . . 139
- 6.1 Using TD(λ) to learn V^π . . . 139
  - 6.1.1 TD(λ): Background . . . 140
  - 6.1.2 The Least-Squares TD(λ) Algorithm . . . 144
  - 6.1.3 LSTD(λ) as Model-Based TD(λ) . . . 146
  - 6.1.4 Empirical Comparison of TD(λ) and LSTD(λ) . . . 149
  - 6.1.5 Applying LSTD(λ) in STAGE . . . 152
- 6.2 Transfer . . . 155
  - 6.2.1 Motivation . . . 155

6.2.2 X-STAGE: A Voting Algorithm for Transfer 157
6.2.3 Experiments 158
6.3 Discussion 161

7 Related Work **163**
7.1 Adaptive Multi-Restart Techniques 163
7.2 Reinforcement Learning for Optimization 166
7.3 Rollouts and Learning for AI Search 169
7.4 Genetic Algorithms 171
7.5 Discussion 173

8 Conclusions **175**
8.1 Contributions 175
8.2 Future Directions 177
8.2.1 Extending STAGE 177
8.2.2 Other Uses of VFA for Optimization 179
8.2.3 Direct Meta-Optimization 180
8.3 Concluding Remarks 182

A Proofs **183**
A.1 The Best-So-Far Procedure Is Markovian 183
A.2 Least-Squares TD(1) Is Equivalent to Linear Regression 187

B Simulated Annealing **189**
B.1 Annealing Schedules 189
B.2 The "Modified Lam" Schedule 190
B.3 Experiments 193

C Implementation Details of Problem Instances **197**
C.1 Bin-packing 197
C.2 VLSI Channel Routing 198
C.3 Bayes Network Learning 199
C.4 Radiotherapy Treatment Planning 199
C.5 Cartogram Design 201
C.6 Boolean Satisfiability 201

References **203**

Chapter 1
INTRODUCTION

In the industrial age, humans delegated physical labor to machines. Now, in the information age, we are increasingly delegating *mental* labor, charging computers with such tasks as controlling traffic signals, scheduling factory production, planning medical treatments, allocating investment portfolios, routing data through communications networks, and even playing expert-level backgammon or chess. Such tasks are difficult *sequential decision problems*:

- the task calls not for a single decision, but rather for a whole series of decisions over time;
- the outcome of any decision may depend on random environmental factors beyond the computer's control; and
- the ultimate objective—measured in terms of traffic flow, patient health, business profit, or game victory—depends in a complicated way on many interacting decisions and their random outcomes.

In such complex problems, optimal decision policies are in general unknown, and it is often difficult, even for human domain experts, to manually encode even reasonably good decision policies in software. A growing body of research in Artificial Intelligence suggests the following alternative methodology:

> **A decision-making algorithm can autonomously *learn* effective policies for sequential decision tasks, simply by simulating the task and keeping statistics on which decisions tend to lead to good ultimate performance and which do not.**

The field of *reinforcement learning*, to which this thesis contributes, defines a principled foundation for this methodology.

1.1 Motivation: Learning Evaluation Functions

In Artificial Intelligence, the fundamental data structure for decision-making in large state spaces is the *evaluation function*. 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 that 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 choice of evaluation function "critically determines search results" [Nilsson 80, 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 {1,3,3,5,9} in the chess evaluation function given above summarize the judgment of generations of chess players. IBM's Deep Blue chess computer, which defeated world champion Garry Kasparov in a 1997 match, used an evaluation function of over 8000 tunable parameters—the values of which were set initially by an automatic procedure, but later carefully hand-tuned under the guidance of a human grandmaster [Hsu *et al.* 90, Campbell 98]. Similar tuning occurs in combinatorial optimization domains such as the Traveling Salesperson Problem [Lin and Kernighan 73] and VLSI circuit design tasks [Wong *et al.* 88]. In such domains the state space consists of legal candidate solutions, and the domain's *objective function*—the function that evaluates the quality of a final solution—can itself serve as an evaluation function to guide search. However, if the objective function has many local optima or regions of constant value (plateaus) with respect to the available search moves, then it will not be effective as an evaluation function. Thus, to get good optimization results, engineers often spend considerable effort tweaking the coefficients of penalty terms and other additions to their objective function; I cite several examples of this in Chapter 3. Clearly, automatic methods for building evaluation functions offer the potential both to save human effort and to optimize search performance more effectively.

1.2 The Promise of Reinforcement Learning

Reinforcement learning (RL) provides a solid foundation for learning evaluation functions for sequential decision problems. Standard RL methods assume that the problem can be formalized as a Markov decision process (MDP), a model of controllable dynamic systems used widely in control theory, artificial intelligence, and operations research [Puterman 94]. I describe the MDP model in detail in Chapter 2. The key fact about this model is that for any MDP, there exists a special evaluation function known as the *optimal value function*. Denoted by $V^*(x)$, the optimal value function predicts the expected long-term reward available from each state x when all future decisions are made optimally. V^* is an ideal evaluation function: a greedy one-step lookahead search based on V^* identifies precisely the optimal long-term decision to make at each state. The problem, then, becomes how to compute V^*.

Algorithms for computing V^* are well understood in the case where the MDP state space is relatively small (say, fewer than 10^7 discrete states), so that V^* can be implemented as a lookup table. In small MDPs, if we have access to the transition model which tells us the distribution of successor states that will result from applying a given action in a given state, then V^* may be calculated exactly by a variety of classical algorithms such as dynamic programming or linear programming [Puterman 94]. In small MDPs where the explicit transition model is not available, we must build V^* from sample trajectories generated by direct interaction with a simulation of the process; in this case, recently discovered reinforcement learning methods such as TD(λ) [Sutton 88], Q-learning [Watkins 89], and Prioritized Sweeping [Moore and Atkeson 93] apply. These algorithms apply dynamic programming in an asynchronous, incremental way, but under suitable conditions can still be shown to converge to V^* [Bertsekas and Tsitsiklis 96, Littman and Szepesvári 96].

The situation is very different for large-scale decision tasks, such as the transportation and medical domains mentioned at the start of this chapter. These tasks have high-dimensional state spaces, so enumerating V^* in a table is intractable—a problem known as the "curse of dimensionality" [Bellman 57]. One approach to escaping this curse is to approximate V^* compactly using a function approximator such as linear regression or a neural network. The combination of reinforcement learning and function approximation, known as *neuro-dynamic programming* [Bertsekas and Tsitsiklis 96] or *value function approximation* [Boyan *et al.* 95], has produced several notable successes on such problems as backgammon [Tesauro 92, Boyan 92], job-shop scheduling [Zhang and Dietterich 95], and elevator control [Crites and Barto 96]. However, these implementations are extremely computationally intensive, requiring many thousands or even millions of simulated trajectories to reach top performance. Furthermore, when general function approximators are used instead of lookup tables, the convergence proofs for nearly all dynamic programming and reinforcement learning algorithms fail to carry through [Boyan and Moore 95, Bertsekas 95, Baird 95, Gordon 95]. Perhaps the strongest convergence result for value function approximation to date is the following [Tsitsiklis and Roy 96]: for an MDP with a *fixed* decision-making policy, the TD(λ) algorithm may be used to calculate an accurate *linear* approximation to the value function. Though its assumption of a fixed policy is quite limiting, this theorem nonetheless applies to the learning done by STAGE, a practical algorithm for global optimization introduced in this dissertation.

1.3 Outline of the Dissertation

This thesis aims to advance the state of the art in value function approximation for large, practical sequential decision tasks. It addresses two questions:

1. Can we devise new methods for value function approximation that are robust and efficient?

2. Can we apply the currently available convergence results to practical problems?

Both questions are answered in the affirmative:

1. I discuss three new algorithms for value function approximation, which apply to three different restricted classes of Markov decision processes: Grow-Support for large deterministic MDPs (§2.2), ROUT for large acyclic MDPs (§2.3), and Least-Squares TD(λ) for large Markov chains (§6.1). Each is designed to gain robustness and efficiency by exploiting the restricted MDP structure to which it applies.

2. I introduce **STAGE**, a new reinforcement learning algorithm designed specifically for large-scale global optimization tasks. In STAGE, commonly applied local optimization algorithms such as stochastic hillclimbing are viewed as inducing fixed decision policies on an MDP. Given that view, TD(λ) or supervised learning may be applied to learn an approximate value function for the policy. STAGE then exploits the learned value function to improve optimization performance in real time.

The thesis is organized as follows:

Chapter 2 presents formal definitions and notation for Markov decision processes and value function approximation. It then summarizes Grow-Support and ROUT, algorithms which learn to approximate V^* in deterministic and acyclic MDPs, respectively. Both these algorithms build V^* strictly backward from the goal, even when given only a forward simulation model, as is usually the case. These algorithms have been presented previously [Boyan and Moore 95, Boyan and Moore 96], but this chapter offers a new unified discussion of both algorithms and new results and analysis for ROUT.

Chapter 3 introduces STAGE, the algorithm which is the main contribution of this dissertation [Boyan and Moore 97, Boyan and Moore 98]. STAGE is a practical method for applying value function approximation to arbitrary large-scale global optimization problems. This chapter motivates and describes the algorithm and discusses issues of theoretical soundness and computational efficiency.

Chapter 4 presents empirical results with STAGE on seven large-scale optimization domains: bin-packing, channel routing, Bayes net structure-finding, radiotherapy treatment planning, cartogram design, Boolean formula satisfiability, and Boggle board setup. The results show that on a wide range of problems, STAGE learns efficiently, effectively, and with minimal need for problem-specific parameter tuning.

Chapter 5 analyzes STAGE's success, giving evidence that reinforcement learning is indeed responsible for the observed improvements in performance. The sensitivity of the algorithm to various user choices, such as the feature representation and function approximator, and to various algorithmic choices, such as when to end a trial and how to begin a new one, is tested empirically.

Chapter 6 offers two significant investigations beyond the basic STAGE algorithm. In Section 6.1, I describe a least-squares implementation of TD(λ), which generalizes both standard supervised linear regression and earlier results on least-squares TD(0) [Bradtke and Barto 96]. In Section 6.2, I discuss ways of transferring knowledge learned by STAGE from already-solved instances to novel similar instances, with the goal of saving training time.

Chapter 7 reviews the relevant work from the optimization and AI literatures, situating STAGE at the confluence of adaptive multi-start local search methods, reinforcement learning methods, genetic algorithms, and evaluation function learning techniques for game-playing and problem-solving search.

Chapter 8 concludes with a summary of the thesis contributions and a discussion of the many directions for future research in value function approximation for optimization.

Chapter 2

Learning Evaluation Functions for Sequential Decision Making

Given only a simulator for a complex task and a measure of overall cumulative performance, how can we efficiently build an evaluation function which enables optimal or near-optimal decisions to be made at every choice point? This chapter discusses approaches based on the formalism of Markov decision processes and value functions. After introducing the notation which will be used throughout this dissertation, I give a review of the literature on value function approximation. I then discuss two original approaches, Grow-Support and ROUT, for approximating value functions robustly in certain restricted problem classes.

2.1 Value Function Approximation (VFA)

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

$$V^*(x) = \text{the expected long-term reward starting from } x, \;\; \text{assuming optimal decisions.} \tag{2.1}$$

Such an evaluation function is ideal in that a greedy local search with respect to V^* will always make the globally optimal move. In this section, I formalize the above definition, review the literature on computing $V^*(x)$, and motivate a new class of approximation algorithms for this problem based on *working backwards*.

2.1.1 Markov Decision Processes

Formally, let our search space be represented as a Markov decision process (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

- a transition model $P : X \times X \times A \rightarrow \Re$, 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 r_t and moves probabilistically to another state x_{t+1}. It is assumed that the agent can fully observe its current state at all times; more general partially observable MDP models [Littman 96] are beyond the scope of this dissertation. The basic MDP 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 94, Harmon *et al.* 95, Littman and Szepesvári 96, Mahadevan *et al.* 97].

Decisions in an MDP are represented by a *policy* $\pi : X \rightarrow A$, which maps each state to a chosen action (or, more generally, a probability distribution over actions). I assume that the policy is *stationary*, that is, unchanging over the course of a simulation. For any stationary policy π, the *policy value function* $V^\pi(x)$ is defined as the expected long-term reward accumulated by starting from state x and following policy π thereafter:

$$V^\pi(x) = \mathrm{E}\Big\{\sum_{t=0}^{\infty} \gamma^t R(x_t, \pi(x_t)) \mid x_0 = x\Big\}. \tag{2.2}$$

Here, $\gamma \in [0,1]$ is a *discount factor* which determines the extent of our preference for short-term rewards over long-term rewards. Assuming bounded rewards, V^π is certainly well-defined for any choice of $\gamma < 1$; in the *undiscounted* case of $\gamma = 1$, V^π remains well-defined under the additional condition that every trajectory is guaranteed to terminate, i.e., reach a special *absorbing state* for which all further rewards are 0. Most of the problems considered in this dissertation have this property naturally; furthermore, an arbitrary MDP evaluated with a discount factor $\gamma < 1$ may be transformed into an absorbing MDP whose undiscounted returns are equivalent to the original problem's discounted returns, simply by introducing a termination probability of $1-\gamma$ at each state. Therefore, I will generally assume $\gamma = 1$ throughout this dissertation, giving equal weight to short-term and long-term rewards.

The policy value function V^π satisfies this linear system of *Bellman equations for prediction*:

$$\forall x, \;\; V^\pi(x) = R(x,\pi(x)) + \gamma \sum_{x' \in X} P(x'|x,\pi(x)) V^\pi(x') \tag{2.3}$$

The solution to an MDP is an *optimal policy* π^* which simultaneously maximizes $V^\pi(x)$ at every state x. A deterministic optimal policy exists for every MDP [Bellman 57]. The policy value function of π^* is the optimal value function V^* of Equation 2.1. It satisfies the *Bellman equations for control*:

$$\forall x, \;\; V^*(x) = \max_{a \in A} \big[R(x,a) + \gamma \sum_{x' \in X} P(x'|x,a) V^*(x') \big] \tag{2.4}$$

From the value function V^*, it is easy to recover the optimal policy: at any state x, any action which instantiates the max in Equation 2.4 is an optimal choice [Bellman 57]. This formalizes the notion that V^* is an ideal evaluation function.

Algorithms for computing V^* are well understood in the case where the MDP state space is relatively small (say, fewer than 10^7 discrete states), so that V^* can be implemented as a lookup table. In small MDPs, if we have explicit knowledge of the transition model $P(x'|x,a)$ and reward function $R(x,a)$, then V^* may be calculated exactly by a variety of classical algorithms such as *linear programming* [D'Epenoux 63], *policy iteration* [Howard 60], *modified policy iteration* [Puterman and Shin 78], or *value iteration* [Bellman 57]. In small MDPs where the transition model is not explicitly available, we must build V^* from sample trajectories generated by direct interaction with a simulation of the process; in this case, reinforcement learning (RL) methods apply. RL methods are either *model-based*, which means they build an empirical transition model from the sample trajectories and then apply one of the aforementioned classical algorithms (e.g., Dyna-Q [Sutton 90], Prioritized Sweeping [Moore and Atkeson 93])—or *model-free*, which means they estimate V^* values directly (e.g., TD(λ) [Sutton 88], Q-learning [Watkins 89], SARSA [Rummery and Niranjan 94, Singh and Sutton 96]). I will have more to say on the issue of model-based versus model-free algorithms in Section 6.1.2. Broadly speaking, all these algorithms may be viewed as applying dynamic programming in an asynchronous, incremental way; and under suitable conditions, all can still be shown to converge to the exact optimal value function [Bertsekas and Tsitsiklis 96, Singh *et al.* 98].

The situation is very different for practical large-scale decision tasks. These tasks have high-dimensional state spaces, so enumerating V^* in a table is intractable—a problem known as the "curse of dimensionality" [Bellman 57]. Computing V^* requires generalization. One natural approach is to encode the states as real-valued feature vectors and to use a function approximator to fit V^* over the feature space. This approach goes by the name *value function approximation* (VFA) [Boyan *et al.* 95].

2.1.2 VFA Literature Review

The current state of the art in value function approximation is surveyed thoroughly in the book *Neuro-Dynamic Programming* [Bertsekas and Tsitsiklis 96]. Here, I briefly review the history of the field and the state of the art, so as to place this chapter's algorithms in context.

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 59, Samuel 67]. Samuel implicitly 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 59, 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.

In the dynamic-programming community, Bellman [63] and others explored polynomial and spline fits for value function approximation in continuous MDPs; reviews of these efforts may be found in [Johnson *et al.* 93, Rust 96]. But Artificial Intelligence research into evaluation function learning was sporadic until the 1980s. In the domain of chess, Christensen [86] tried replacing Samuel's coefficient-tweaking procedure with least-squares regression, and was able to learn reasonable weights for the chess material-advantage function. In Othello, Lee and Mahajan [88] trained a nonlinear evaluation function on expertly played games, and it played at a high level. Christensen and Korf [86] put forth a unified theory of heuristic evaluation functions, advocating the principles of "outcome determination" and "move invariance"; these correspond precisely to the two key properties of MDP value functions, that they represent long-term predictions and that they satisfy the Bellman equations. Finally, in the late 1980s, the reinforcement learning community elaborated the deep connection between AI search and dynamic programming [Barto *et al.* 89, Watkins 89, Sutton 90, Barto *et al.* 95]. This connection had been unexplored despite the publication of an AI textbook by Richard Bellman himself [Bellman 78].

Reinforcement learning's most celebrated success has also been in a game domain: backgammon [Tesauro 92, Tesauro 94]. Tesauro modified Sutton's TD(λ) algorithm

[Sutton 88], which is designed to approximate V^π for a fixed policy π, to the task of learning an optimal value function V^* and optimal policy. The modification is simple: instead of generating sample trajectories by simulating a fixed policy π, generate sample trajectories by simulating the policy μ which is greedy with respect to the current value function approximation $\tilde{V}$:

$$\mu(x) = \operatorname*{argmax}_{a \in A} \big[R(x,a) + \gamma \sum_{x' \in X} P(x'|x,a)\tilde{V}(x')\big] \tag{2.5}$$

(Occasional non-greedy "exploration" moves are also usually performed [Thrun 92, Singh *et al.* 98], but were found unnecessary in backgammon because of the domain's inherent stochasticity [Tesauro 92].) The modified algorithm has been termed *optimistic TD(λ)* [Bertsekas and Tsitsiklis 96], because little is known of its convergence properties. An implementation is sketched in Table 2.1.2. When $\lambda = 0$, the algorithm strongly resembles Real-Time Dynamic Programming (RTDP) [Barto *et al.* 95], except that RTDP assigns target values at each state by a "full backup" (averaging over all possible outcomes, as in value iteration) rather than TD(0)'s "sample backups" (learning from only the single observed outcome). Applying optimistic TD(λ) with a multi-layer perceptron function approximator, Tesauro's program learned an evaluation function which produced expert-level backgammon play. These results have been replicated by myself [Boyan 92] and others.

Tesauro's combination of optimistic TD(λ) and neural networks has been applied to other domains, including elevator control [Crites and Barto 96] and job-shop scheduling [Zhang and Dietterich 95]. (I will discuss the scheduling application in detail in Section 7.2.) Nevertheless, it is important to note that when function approximators are used, optimistic TD(λ) provides no guarantees of optimality. The following paragraphs summarize the current convergence results for value function approximation. For both the prediction learning (approximating V^π) and control learning (approximating V^*) tasks, the relevant questions are (1) do the available algorithms converge, and (2) if so, how good are their resulting approximations?

We first consider the case of approximating the policy value function V^π of a *fixed* policy π. The TD(λ) family of algorithms applies here. When $\lambda = 1$, TD(λ) reduces to performing stochastic gradient descent to minimize the squared difference between the approximated predictions $\tilde{V}^\pi$ and the observed simulation outcomes. Under standard conditions, using any parametric function approximator, this will converge to a local optimum of the squared-error function. For sufficiently small λ, however, TD(λ) may diverge when nonlinear function approximators are used [Bertsekas and Tsitsiklis 96]. Only in the case where the function approximator is a *linear architecture* over state features has TD(λ) been proven to converge for arbitrary λ [Tsitsiklis and Roy 96].

Optimistic TD($\boldsymbol{\lambda}$) for value function approximation:

Given:

- a *simulation model* for MDP X;
- a function approximator $\tilde{V}(x)$ parametrized by weight vector $\mathbf{w}$;
- a sequence of *step sizes* $\alpha_1, \alpha_2, \ldots$ for incremental weight updating; and
- a parameter $\lambda \in [0, 1]$.

Output: a weight vector $\mathbf{w}$ such that $\tilde{V}(x) \approx V^*(x)$.

Set $\mathbf{w} := \mathbf{0}$ *(or an arbitrary initial estimate).*
for $n := 1, 2, \ldots$ **do**: {

1. Using the greedy policy for the current evaluation function $\tilde{V}$ (see Eq. 2.5), generate a trajectory from a start state in X to termination: $x_0 \to x_1 \cdots \to x_T$. Record the rewards $r_0, r_1, \ldots r_T$ received at each step.
2. Update the weights of $\tilde{V}$ from the trajectory as follows:
   for i := T **downto** 0, **do**: {
   $$\mathrm{targ}_i := \begin{cases} r_T \text{ (the terminal reward)} & \text{if } i = T \\ r_i + \lambda \cdot \mathrm{targ}_{i+1} + (1-\lambda) \cdot \tilde{V}(x_{i+1}) & \text{otherwise.} \end{cases}$$
   Update $\tilde{V}$'s weights by delta rule: $\mathbf{w} := \mathbf{w} + \alpha_n(\mathrm{targ}_i - \tilde{V}(x_i))\nabla_w \tilde{V}(x_i)$.
   }

}

TABLE 2.1. Optimistic TD(λ) for undiscounted value function approximation in an absorbing MDP. This easy-to-implement version performs updates after the termination of each trajectory. For an incremental version that performs updates after each transition, refer to [Sutton 87]. In practice, trajectories are often generated using a mixture of greedy moves and exploration moves [Thrun 92, Singh *et al.* 98].

A useful error bound has also been shown in the linear case: the resulting fit is worse than the best possible linear fit by a factor of at most $(1-\gamma\lambda)/(1-\gamma)$, assuming a discount factor of $\gamma < 1$ [Tsitsiklis and Roy 96]. This implies that TD(1) is guaranteed to produce the best fit, but the bound quickly deteriorates as λ decreases. The same qualitative conclusion applies (though the formula for the bound is more complex) for $\gamma = 1$ [Bertsekas and Tsitsiklis 96].

We now proceed to the harder problem of approximating the optimal value function V^*. First, independent of how we construct it, is an approximate value function $\tilde{V}$ useful for deriving a decision-making policy? Singh and Yee [94] show that if $\tilde{V}$ differs from V^* by at most ϵ at any state, then the expected return of the greedy policy for $\tilde{V}$ will be worse than that of the optimal policy by a factor of at most $2\gamma\epsilon/(1-\gamma)$. A similar result holds in the undiscounted case, assuming all policies are proper (γ is then replaced by a contraction factor in a suitably weighted max norm). This bound is not particularly comforting, since $1/(1-\gamma)$ will be large in practical applications, but at least it guarantees that policies cannot be *arbitrarily* bad.

How should we construct V^*? In general, algorithms based on value iteration's one-step-backup operator, such as optimistic TD(λ), use function approximator predictions to assign new training values for that same function approximator—a recursive process that may propagate and enlarge approximation errors, leading to parameter divergence. I have demonstrated empirically that such divergence can indeed happen when offline value iteration is combined with commonly used function approximators, such as polynomial regression and neural networks [Boyan and Moore 95]. Small illustrative examples of divergence have also been demonstrated [Baird 95, Gordon 95]. Sutton has argued that certain of these instabilities may be prevented by sampling states along simulated trajectories, as optimistic TD(λ) does [Sutton 96]; but there are no convergence proofs of this as yet.

Parameter divergence in offline value iteration can provably be prevented by using function approximators belonging to the class of *averagers*, such as k-nearest-neighbor [Gordon 95]. However, this class excludes practical function approximators which extrapolate trends beyond their training data (e.g., global or local polynomial regression, neural networks). *Residual algorithms*, which attempt to blend optimistic TD(λ) with a direct minimization of the residual approximation errors in the Bellman equation, are guaranteed stable with arbitrary parametric function approximators [Baird 95]; these methods are promising but as yet unproven on real-world problems.

2.1.3 Working Backwards

Value iteration (VI) computes V^* by repeatedly sweeping over the state space, applying Equation 2.4 as an assignment statement (this is called a "one-step backup") at each state in parallel. Suppose the lookup table is initialized with all 0's. Then after the i^{th} sweep of VI, the table will store the maximum expected return of a path of length i from each state. For so-called *stochastic shortest path problems* in which every trajectory produced by the optimal policy inevitably terminates in an absorbing state [Bertsekas and Tsitsiklis 96], this corresponds to the intuition that VI works by propagating correct V^* values backwards, by one step per iteration, from the terminal states.

I have explored the efficiency and robustness gains possible when VI is modified to take advantage of the working-backwards intuition. 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 [Bertsekas and Tsitsiklis 96].

2. (possibly stochastic) *acyclic* domains: domains where no trajectory can pass through the same state twice. Many problems naturally have this property (e.g., board-filling games like tic-tac-toe and Connect-Four, industrial scheduling as described in Section 2.3.4 below, and 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 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.* 90].[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.

[1] Although Cormen et al. [90] present DAG-SP only for deterministic acyclic problems, it applies straightforwardly to the stochastic case.

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 distinction is important in the context of generalization and the possibility of approximation error.

In sum, 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 95] and is summarized briefly in Section 2.2. My variant of DAG-SP is an algorithm called ROUT [Boyan and Moore 96], which I describe in more detail and with new results in Section 2.3. Other researchers have also investigated learning control by working backwards, notably Atkeson [94] for the case of deterministic domains with continuous dynamics.

2.2 VFA in Deterministic Domains: "Grow-Support"

This section summarizes Grow-Support, an algorithm for value function approximation in large, deterministic, minimum-cost-to-goal domains [Boyan and Moore 95]. Grow-Support is designed to construct the optimal value function with a generalizing function approximator while remaining robust and stable. It recognizes that function approximators cannot always be relied upon to fit the intermediate value functions produced by value iteration. Instead, it assumes only that the function approximator can represent the final V^* function accurately, if given accurate training values for a prespecified collection of sample states. The specific principles of Grow-Support are as follows:

1. We maintain a "support" subset of sample states whose final V^* values have been computed, starting with terminal states and then growing backward from there. The fitter $\tilde{V}$ is trained only on these values, which we assume it is capable of fitting.

2. Instead of propagating values by one-step backups, we use *rollouts*—simulated trajectories guided by the current greedy policy on $\tilde{V}$. They explicitly verify the achievability of a state's estimated future reward before that state is added to the support set. In a rollout, the new $\tilde{V}$ training value is derived from rewards

along an actual path to the goal, not from the predictions made by the previous iteration's function approximation. This prevents divergence.

3. We take maximum advantage of generalization. On each iteration, we add to the support set any sample state that can, by executing a single action, reach a state that passes the rollout test. In a discrete environment, this would cause the support set to expand in one-step concentric "shells" back from the goal, similar to Dijkstra's algorithm. But in the continuous case, the function approximator may be able to extrapolate correctly well beyond the support region—and when this happens, we can add many points to the support set at once. This leads to the very desirable behavior that the support set grows in big jumps in regions where the value function is smooth.

Grow-Support($\hat{X}, G, A,$ NEXTSTATE, COST, $\tilde{V}$):
Given: • a finite collection of states $\hat{X} = \{x_1, x_2, \ldots x_N\}$ sampled from the continuous state space $X \subset \Re^n$, and goal region $G \subset X$
• a finite set of allowable actions A
• a deterministic transition function NEXTSTATE : $X \times A \to X$
• the 1-step cost function COST : $X \times A \to \Re$
• a smoothing function approximator $\tilde{V}$
• a tolerance level ϵ for value function approximation error
SUPPORT := $\{(x_i \mapsto 0) \mid x_i \in G\}$
repeat
  Train $\tilde{V}$ to approximate the training set SUPPORT
  for each $x_i \notin$ SUPPORT do
    $c := \min_{a \in A} \left[\text{COST}(x_i, a) + \text{ROLLOUTCOST}(\text{NEXTSTATE}(x_i, a), \tilde{V})\right]$
    if $c < \infty$ then
      add $(x_i \mapsto c)$ to the training set SUPPORT
until SUPPORT stops growing or includes all sample points.

ROLLOUTCOST(state x, fitter $\tilde{V}$):
  Starting from x, simulate the greedy policy defined by value function $\tilde{V}$ until either reaching the goal, or exceeding a total path cost of $\tilde{V}(x) + \epsilon$.
  Then return:
  $\longrightarrow$ the actual total cost of the path, if goal is reached with cost $\leq \tilde{V}(x) + \epsilon$;
  $\longrightarrow \infty$, if goal is not reached in cost $\tilde{V}(x) + \epsilon$.

TABLE 2.2. The Grow-Support algorithm and ROLLOUTCOST subroutine

The algorithm is sketched in Table 2.2. In a series of experiments reported in [Boyan and Moore 95], I found that Grow-Support is more robust than value iteration with function approximation. (Several follow-up studies provide additional insight into value iteration's potential for divergence [Gordon 95, Sutton 96].) Grow-Support was also seen to be no more computationally expensive, and often much cheaper, despite the overhead of performing rollouts. Reasons for this include: (1) the rollout test is not expensive; (2) once a state has been added to the support, its value is fixed and it needs no more computation; and most importantly, (3) the aggressive exploitation of generalization enables the algorithm to converge in very few iterations.

It is easy to prove that Grow-Support will always terminate after a finite number of iterations. If the function approximator is inadequate for representing the V^* function, Grow-Support may terminate before adding all sample states to the support set. When this happens, we then know exactly which of the sample states are having trouble and which have been learned. This suggests potential schemes for adaptively adding sample states in problematic regions. The ROUT algorithm, described next, does adaptively generate its own set of sample states for learning.

2.3 VFA in Acyclic Domains: "ROUT"

As Grow-Support scaled up Dijkstra's algorithm for deterministic domains, ROUT aims to scale up DAG-Shortest-Paths (DAG-SP) for stochastic, acyclic domains. In large combinatorial spaces requiring function approximation, 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 $\{\text{state}\rightarrow\text{value}\}$ pair is added to a training set for a function approximator. I determine the training value by a one-step backup rather than rollouts because, unlike the deterministic MDPs to which Grow-Support applies, stochastic MDPs would require performing not one but many rollouts for accurate value determination. However, ROUT still does use an analogue of Grow-Support's "rollout test" to identify the states at which the one-step backup may be safely applied.

In sum, ROUT's main loop consists of identifying a frontier state; determining its V^* value; and retraining the approximator. The training set, constructed adaptively, grows backwards from the goal. HuntFrontierState is the key subroutine ROUT uses to identify a good state to add to the training set. The criteria for such a state x are as follows:

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 2.4 to some tolerance ϵ). If all states after x on all sample trajectories are self-consistent, then x is deemed ready, and ROUT will add x to its training set. If, on the other hand, a trajectory from x reveals any inconsistencies in the approximated value function, then we flag that trajectory's *last* such inconsistent state, and restart HuntFrontierState from there. Table 2.3 specifies the algorithm, and Figure 2.3 illustrates how the routine works.

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

The following sections present experimental 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 optimistic TD(λ) 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). The results show that ROUT learned evaluation functions which were as good or better than those learned by TD(λ), and used an order of magnitude less training data in doing so. I also report preliminary results on a fourth domain, a simplified production scheduling task.

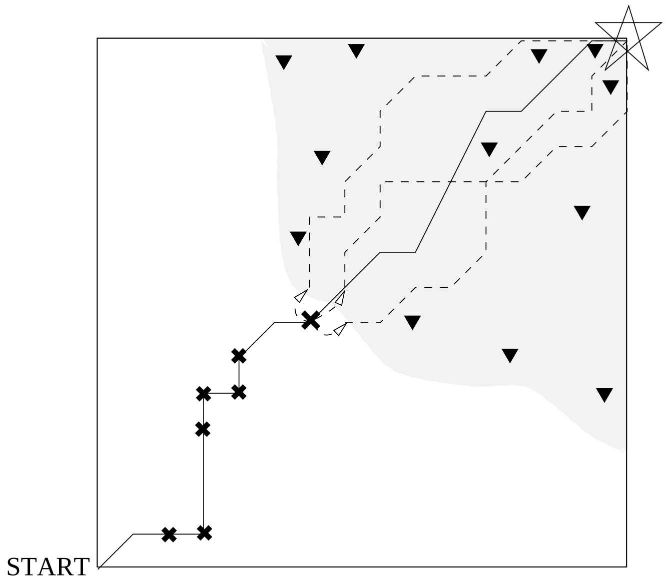


FIGURE 2.1. A schematic of ROUT working on an acyclic two-dimensional navigation domain, where the allowable actions are only $\rightarrow$, $\nearrow$, and $\uparrow$. Suppose that ROUT has thus far established training values for V^* at the triangles, and that the function approximator has successfully 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.

ROUT(start states $\hat{X}$, fitter $\tilde{V}$):
  /* *Assumes that the world model MDP is known and acyclic.* */
  initialize training set $S := \emptyset$, and $\tilde{V} :=$ an arbitrary fit;
  repeat:
    for each start state $x \in \hat{X}$ not yet marked "done", do:
      $s :=$ HUNTFRONTIERSTATE$(x, \tilde{V})$;
      add $\{s \mapsto$ one-step-backup(s)$\}$ to training set S and re-train fitter $\tilde{V}$ 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, fitter $\tilde{V}$):
  /* *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$(y, \tilde{V})$.
  /* *reaching this point, x's subtree is deemed all self-consistent and correct!* */
  return x.

TABLE 2.3. The ROUT algorithm and the HUNTFRONTIERSTATE subroutine

2.3.1 Task 1: Stochastic Path Length Prediction

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 $\tilde{V}$ makes predictions by interpolating between values at every fourth state. This is equivalent to using a linear approximator over the four-element feature vector representation depicted in Figure 2.2.

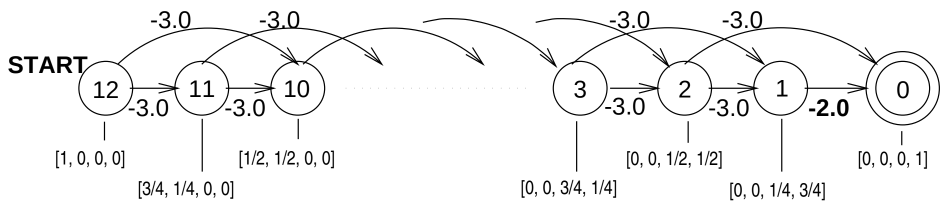


FIGURE 2.2. The Hopworld Markov chain. Each state is represented by a four-element feature vector as shown. The function approximator is linear.

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 2.4. ROUT's performance is efficient and predictable on this contrived problem. At the start, HUNTFRONTIERSTATE finds $\tilde{V}$ is inconsistent and trains $\tilde{V}(1)$ and $\tilde{V}(2)$ to be -2 and -4, respectively. Linear extrapolation then forces states 3 and 4 to be correct. On the third iteration, $\tilde{V}(5)$ 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, or better yet, by eliminating its learning rates and performing *Least-Squares TD* as described later in Section 6.1.2.

2.3.2 Task 2: A Two-Player Dice Game

"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 96] similar to Equation 2.4. 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 backpropagation would be used with ROUT.[2]

The Pig results are charted in Table 2.4 and graphed in Figure 2.3. The graph shows the learning curves for the best single trial of each of six classes of runs: TD(0), TD(0.8) and TD(1), with and without exploration. (The 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.

2.3.3 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 85]. While an optimal solution in the infinite-horizon case can be found efficiently using Gittins indices, solving the finite-horizon problem is equivalent to solving a large acyclic, stochastic MDP in belief space [Berry and Fristedt 85]. I 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

[2] Unlike TD, which works only with parametric function approximators for which $\nabla_w \tilde{V}(x)$ 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 network fits for both algorithms.

Problem	Method	training samples	total evals	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 2.4. Summary of results. For each algorithm on each problem, I list two measurements of time to quiescence followed by three measurements of the solution quality. The measurements for TD were taken at the time when, roughly, best performance was first consistently reached. (Key: M=10^6; * denotes optimal performance for each task.)

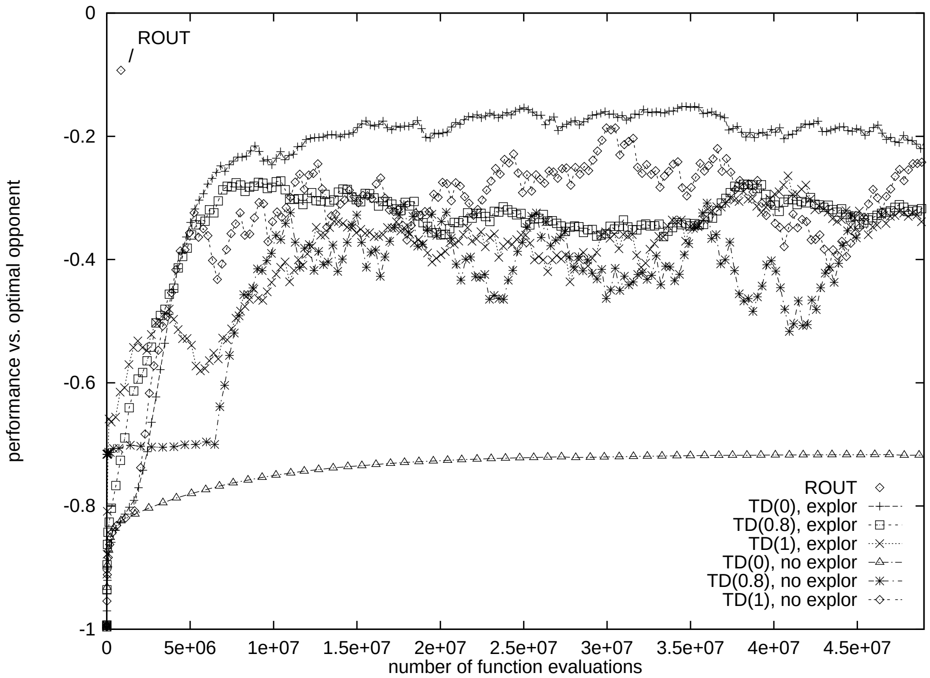


FIGURE 2.3. Performance of Pig policies learned by TD and ROUT. ROUT's performance is marked by a single diamond at the top left of the graph.

optimal exploration policy, which has an expected reward of 0.6821 per pull.

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

The results are shown in Table 2.4. All methods do spectacularly well, although the TD methods again require more trajectories and more evaluations. Careful inspection of the problem reveals that a globally linear value function, extrapolated from the states close to the end, has low Bellman residual and performs very nearly optimally. Both ROUT and TD successfully exploit this linearity.

2.3.4 Task 4: Scheduling a Factory Production Line

Production scheduling, the problem of deciding how to configure a factory sequentially to meet demands, is a critical problem throughout the manufacturing industry.[3] We assume we have a modest number of products (2–100) and must produce enough of each to keep warehouse stocks high enough to satisfy customer requests for bulk shipments. This production model is common, for example, for most goods found in a supermarket.

An instance of the production scheduling problem is composed of five parts:

Machines and products. This is a list of what machines are present in the factory, and what products can be made on the machines. There may be complex constraints such as "machine A can only make product 1 when machine B is not making product 3." A complete, legal assignment of products onto the set of machines is called a *configuration.* There is also a special "closed" configuration which represents a decision to shut the factory down.

Changeover times. It generally takes a certain amount of time to switch the factory from one configuration to another. During that time, there is no production. The problem definition includes a (possibly stochastic) estimate of how long it takes to change each configuration to each other configuration.

Production rates. Each configuration produces a set of products at a certain rate. There may be dependencies between the machines. For example, machine B may produce product 2 faster if machine A is also producing product 2. The actual production rates in the factory may be very stochastic; for example, some machines may jam frequently, causing irregular delays on the production line.

[3]The application of RL to production scheduling reported here is the result of a collaboration with Jeff Schneider and Andrew Moore [Schneider *et al.* 98].

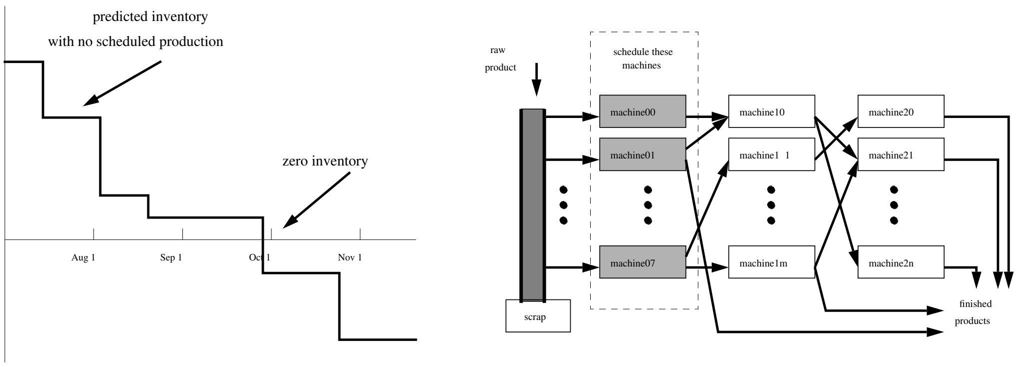


FIGURE 2.4. Demand inventory curves (left) and factory layout (right). See text for further explanation.

Inventory demand curves. At the time a schedule is created, a demand curve for each product is available from a corporate marketing and forecasting group. As shown in Fig. 2.4, each curve starts at the left with the current inventory of that product. The inventory decreases over time as future product shipments are made and eventually goes below zero if no new production occurs. To avoid penalties, the scheduler should call for more production before the demand curve falls below zero. These curves may also change over time as new information about future product demand becomes available.

Schedule costs. Running a schedule generates a dollar measure of net profit or loss. This includes the costs of running the factory, paying the workers, purchasing the raw materials, and carrying inventory at the warehouse. It also includes heuristic costs such as an estimate of the damage done by failing to fill a customer request when the warehouse inventory goes to zero. Finally, it includes the revenue generated from selling product to a customer.

Given this problem description, the task of production scheduling is to maximize expected profit by selecting factory configurations over a period of time. In cases where the production rates and demand curves are assumed deterministic, the problem reduces to finding the optimal *open-loop* schedule: that is, find a fixed sequence of configurations that maximizes profit. In the general stochastic case, the optimal choice of configuration at time t will depend on the outcomes of earlier configurations, so the optimal solution has the form of a *closed-loop* scheduling policy.

The production scheduling problem is modelled very naturally as a Markov Decision Process, as follows:

- The system state is defined by the current time $t \in 0 \ldots T$; the current inventory of each product $p_1 \dots p_N$; and, if there are configuration-dependent changeover times, the current factory configuration.

- The action set consists of all legal factory configurations. We assume a discrete-time model, so the configuration chosen at time t will run unchanged until time $t+1$.

- The stochastic transition function applies a simulation of the factory to compute the change in all inventory levels realized by running configuration c_t for 1 timestep. This model handles random variations in production rates straightforwardly; it also handles changeover times by simply decreasing production in proportion to the (possibly stochastic) downtime. The time t is incremented on each step, and the process terminates when $t = T$.

- The immediate reward function is computed from the inventory levels, based on the demand curve at time t. It incorporates the revenues from production, penalties from late production, employee costs, operating costs and changeover cost incurred during the period. On the final time period (transition from $t = T - 1$ to T), a terminal "reward" assigns additional penalties for any outstanding unsatisfied demands.

The MDP model fully represents uncertainty in production rates and changeover times. As defined here, the model also handles noise in the demands if that noise is time-independent, but it cannot account for the possibility of the demand curves being randomly updated in the middle of a schedule, since that would make the MDP transition probabilities nonstationary. Finally, since the current time t is included as part of the state, the MDP is acyclic: ROUT may be applied.

I applied ROUT to a highly simplified version of a real factory's scheduling problem. The task involves scheduling 8 weeks of production; however, configurations may be changed only at 2-week intervals, and only 17 configuration choices are available. Of these 17, nine have deterministic production rates; the other eight each have two stochastic outcomes, producing only 1/3 of their usual amount with probability 0.5. With a total of $9 \times 1 + 8 \times 2 = 25$ outcomes possible from every state, and four scheduling periods, there are $25^4 = 390,625$ possible trajectories through the space. The optimal policy can be computed by tabulating $V^*(x)$ at every possible intermediate state x of the factory, of which there are $1 + 25 + 25^2 + 25^3 = 16,276$. The optimal policy results in an expected cumulative reward of $-\$22.8$M. By contrast, a random schedule attains a reward of $-923M$. A greedy policy, which at each step selects a configuration to maximize only the next period's profit, attains $-\$97.9$M.

I applied ROUT to this instance, trying two different memory-based function approximators: 1-nearest neighbor and locally weighted linear regression [Cleveland and Devlin 88]. (The local regression used a kernel width of $\frac{1}{8}$ of the range of each input dimension in the training data; this fraction was tuned manually over powers of $\frac{1}{2}$.) Since these function approximators are nonparametric, TD(λ) cannot be used to train them, so ROUT is compared only to the optimal, greedy, and random policies. ROUT's exploration and tolerance parameters were also tuned manually.

Algorithm	Mean Profit	95% C.I.	optimal runs
Optimal	-22.8		1
Random	-923.2	±58.7	0
Greedy	-97.9	±15.1	0
ROUT + nearest neighbor	N/A		0
ROUT + locally weighted linear	-45.0	±16.9	10/16

TABLE 2.5. Results on the production scheduling task

Table 2.5 summarizes the results. When nearest-neighbor was used as the function approximator, ROUT did not obtain sufficient generalization from its training set and failed to terminate within a limit of several hours. However, with a locally weighted regression model, ROUT did run to completion and produced an approximate value function which significantly outperformed the greedy policy. Moreover, over half of these runs did indeed terminate with the *optimal* closed-loop scheduling policy. In these cases, ROUT's final self-built training set for value function approximation consisted of only about 110 training points—a substantial reduction over the 16,276 required for full tabulation of V^*. ROUT's total running time ($\approx$ 1 hour on a 200 MHz Pentium Pro) was roughly half of that required to enumerate V^* manually.

From these preliminary results, I conclude that ROUT does indeed have the potential to approximate V^* extremely well, given a suitable function approximator for the domain. However, since it runs quite slowly on even this simple problem, I believe ROUT will not scale up to practical scheduling instances without further modification.

2.4 Discussion

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

Grow-Support and ROUT represent first steps toward a new class of algorithms for solving large-scale MDPs. Their primary innovation is that, without falling prey to the curse of dimensionality, they are able to explicitly represent which states are already solved and which are not yet solved. Using this information, they "work backwards," computing accurate V^* values at targeted unsolved states using either function approximator predictions at solved states or Monte Carlo rollouts. Importantly, and unlike the Dijkstra and DAG-SP exact algorithms on which they are based, they grow the solution set for V^* back from the goal without requiring an explicit backward model for the MDP. Only forward simulations are used; this constrains the distribution of visited states to areas that are actually reachable during task execution. By treating solved and unsolved states differently, Grow-Support and ROUT eliminate the possibility of divergence caused by repeated value re-estimation.

This chapter has reviewed the state of the art in reinforcement learning, a field which is grounded solidly in the theory of dynamic programming but provides few guarantees for the practical cases where function approximators, rather than lookup tables, must be used to construct the value function. I have described two novel algorithms for approximating V^* which are guaranteed stable and which perform well in practice. In the remaining chapters of this thesis, I consider the simpler VFA problem of approximating V^π for a fixed policy π, on which TD(λ) with linear function approximators does come with strong convergence guarantees—and I demonstrate a practical way of exploiting these approximations to improve search performance on global optimization tasks.

Chapter 3

Learning Evaluation Functions for Global Optimization

3.1 Introduction

3.1.1 Global Optimization

Global optimization—the problem of finding the best possible configuration from a large space of possible configurations—is among the most fundamental of computational tasks. Its numerous applications in science, engineering, and industry include

- design and layout: optimizing VLSI circuit designs for computer architectures [Wong *et al.* 88], packing automobile components under a car hood [Szykman and Cagan 95], architectural design, magazine page layout
- resource allocation: airline scheduling [Subramanian *et al.* 94], school timetabling, factory production scheduling, military logistics planning
- parameter optimization: generating accurate models for weather prediction, ecosystem modeling [Duan *et al.* 92], economic modeling, traffic simulations, intelligent database querying [Boyan *et al.* 96]
- scientific analysis: computational biology (gene sequencing) [Karp 97], computational chemistry (protein folding) [Neumaier 97]
- engineering: medical robotics (radiotherapy for tumor treatment) [Webb 91], computer vision (line matching) [Beveridge *et al.* 96]

Formally, an instance of global optimization consists of a *state space* X and an *objective function* $\mathrm{Obj} : X \to \Re$. The goal is to find a state $x^* \in X$ which minimizes Obj, that is, $\mathrm{Obj}(x^*) \leq \mathrm{Obj}(x)\ \forall x \in X$. If the space X is so small that every state can be evaluated, then obtaining the exact solution x^* is trivial; otherwise, special knowledge of the problem structure must be exploited. For example, if X is a convex linearly constrained subset of $\Re^n$ and Obj is linear, then x^* can be found efficiently by linear programming.

But for many important optimization problems, efficient solution methods are unknown. Practical *combinatorial optimization* problems where X is finite but enormous, such as the applications listed under "design and layout" and "resource allocation" above, all too often fall into the class of *NP-hard* problems, for which efficient exact algorithms are thought not to exist [Garey and Johnson 79]. Recent progress in cutting-plane and branch-and-bound algorithms, especially as applied to mixed-integer linear programming and Travelling Salesperson Problems, has led to exact solutions for some large NP-hard problem instances [Applegate *et al.* 95, Subramanian *et al.* 94]. However, for most real-world domains, practitioners resort to heuristic *approximation methods* which seek good approximate solutions.

To illustrate the discussion that follows, I will present an example optimization instance from the domain of one-dimensional bin-packing. In bin-packing, we are given a *bin capacity* C and a list $L = (a_1, a_2, ... a_n)$ of n *items*, each having a *size* $s(a_i) > 0$. The goal is to pack the items into as few bins as possible, i.e., partition them into a minimum number m of subsets $B_1, B_2, ..., B_m$ such that for each B_j, $\sum_{a_i \in B_j} s(a_i) \leq C$. This problem has many real-world applications, including loading trucks subject to weight limitations, packing commercials into station breaks, and cutting stock materials from standard lengths of cable or lumber [Coffman *et al.* 96]. It is also a classical NP-complete optimization problem [Garey and Johnson 79]. Figure 3.1 depicts a small bin-packing instance with thirty items. Packed optimally, these items fill 9 bins exactly to capacity.

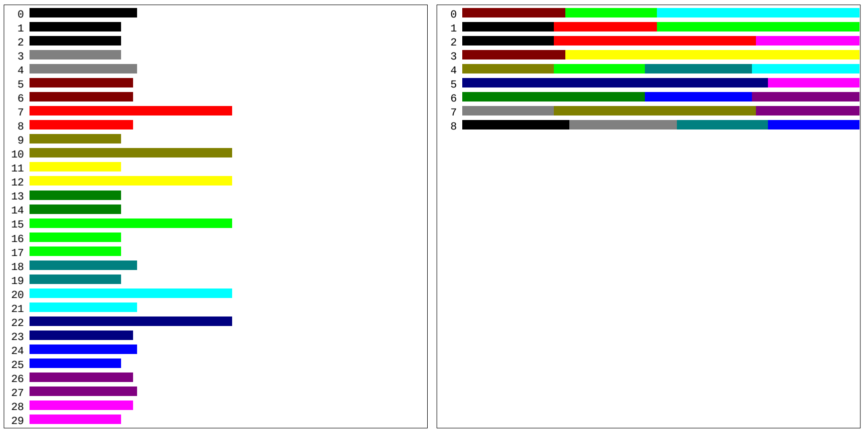


FIGURE 3.1. A small bin-packing domain. Left: initial state (30 items, each in its own bin). Right: the global optimum (nine bins packed with 0 waste).

3.1.2 Local Search

Many special-purpose approximation algorithms have been proposed and analyzed for the bin-packing problem [Coffman *et al.* 96]. However, we will focus instead on a broad class of general-purpose algorithms for the general global optimization problem: the class of *local search* algorithms. All of these work by defining a *neighborhood* $N(x_i) \subset X$ for each state $x_i \in X$. Usually, the neighborhood of x consists of simple perturbations to x—states whose new Obj value can be computed incrementally and efficiently from $\mathrm{Obj}(x)$. However, complex neighborhood structures can also be effective (e.g., the Lin-Kernighan algorithm for the Traveling Salesman Problem [Lin and Kernighan 73]). The neighborhood structure and objective function together give rise to a *cost surface* over the state space. Local search methods start at some initial state, perhaps chosen randomly, and then search for a good solution by iteratively moving over the cost surface from state to neighboring state.

The simplest local search method is known variously as *greedy descent*, *iterative improvement* or, most commonly, *hillclimbing*.[1] Hillclimbing's essential property is that it never accepts a move that worsens Obj. There are several variants. In *steepest-descent* hillclimbing, every neighbor $x' \in N(x)$ is evaluated, and the neighbor with minimal $\mathrm{Obj}(x')$ is chosen as the successor state (ties are broken randomly). Steepest-descent terminates as soon as it encounters a *local optimum*, a state x for which $\mathrm{Obj}(x) \leq \mathrm{Obj}(x') \; \forall x' \in N(x)$. In another variant, *stochastic hillclimbing* (also known as *first-improvement*), random neighbors of x are evaluated one by one, and the first one which improves on $\mathrm{Obj}(x)$ is chosen. Whether or not to accept *equi-cost moves*, to a neighboring state of equal cost, is a parameter of the algorithm. Another parameter, called *patience*, governs termination: the search halts after *patience* neighbors have been evaluated consecutively and all found unhelpful. Thus, the final state reached is only probabalistically, not with certainty, a local optimum. Despite this, stochastic hillclimbing is often preferred for its ability to find a good solution very quickly.

Hillclimbing's obvious weakness is that it gets stuck at the first local optimum it encounters. Alternative local search methods provide a variety of heuristics for accepting some moves to worse neighbors. The possibilities include

- "Force-best-move" approaches: move to the best neighbor, even if its value is worse. (This approach has been successful in the GSAT algorithm [Selman and Kautz 93].)

[1] In this dissertation, I will use the term "hillclimbing" even though we seek a minimum of the cost surface. Simply imagine that the goal of our metaphorical mountain climber is to minimize his distance from the sky.

- More generally, "biased random walk" approaches: allow moves to worse neighbors stochastically, perhaps depending on how much worse the neighbor is. (This approach has been successful in the WALKSAT algorithm [Selman *et al.* 96] and others.)

- Simulated annealing [Kirkpatrick *et al.* 83]. This popular approach is like the biased random walk, but gradually lowers the probability of accepting a move to a worse neighbor. Search terminates at a local optimum after the probability falls to zero. Simulated annealing approaches are discussed in detail in Appendix B.

- Multiple restarting. Because stochastic hillclimbing is so fast, a reasonable approach is to apply it repeatedly and return the best result. The restart can be from the start state, a randomly chosen state, or a state selected according to some "smarter" heuristic. I review the literature on smart restarting methods in Section 7.1. Multiple restarting is effective not only for hillclimbing but for any of the search procedures listed above.

In all of these procedures, since the objective function value does not improve monotonically over time, the search outcome is defined to be the best state ever evaluated over the entire trajectory.

Local search approaches are easy to apply. For the bin-packing domain described above, a solution state x simply assigns a bin number $b(a_i)$ to each item. Each item is initially placed alone in a bin: $b(a_1) = 1, b(a_2) = 2, \ldots, b(a_n) = n$. Neighboring states can be generated by moving any single item a_i into a random other bin with enough spare capacity to accommodate it. (For details, please see Section 4.2.) Figure 3.2 illustrates the solutions discovered by three independent runs of stochastic hillclimbing without equi-cost moves on the example instance of Figure 3.1. The local optima shown use 11, 14, and 12 bins, respectively. In 400 further runs, hillclimbing produced a 10-bin solution seven times but never the optimal 9-bin solution.

3.1.3 Using Additional State Features

Local search has been likened to "trying to find the top of Mount Everest in a thick fog while suffering from amnesia" [Russell and Norvig 95, p.111]. Whenever the climber considers a step, he consults his altimeter, and decides whether to take the step based on how his altitude has changed. But suppose the climber has access to not only an altimeter, but also additional senses and instruments—for example, his x and y location, the slope of the ground underfoot, whether or not he is on a trail, and

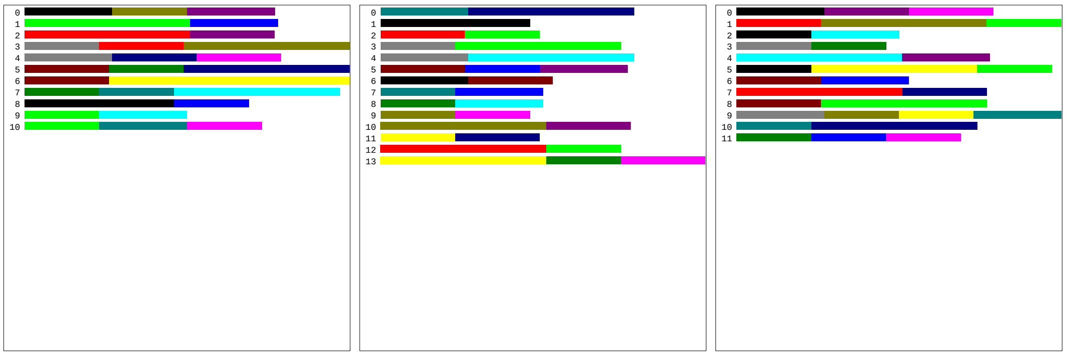


FIGURE 3.2. Three bin-packing local-optima

the height of the nearest tree. These additional "features" may enable the climber to make a more informed, more foresightful, evaluation of whether to take a step.

In real optimization domains, such additional features of a state are usually plentiful. For example, here are a few features for the bin-packing domain:

- the variance in fullness of the bins
- the variance in the number of items in each bin
- the fullness of the least-full bin
- the average ratio of largest to smallest item in each bin
- the "raw" state $b(a_1), b(a_2), \ldots$

Features like these, if combined and weighted correctly, can form a new objective function which indicates not just how good a state is itself as a final solution, but how good it is at leading search to other, better solutions.

This point is illustrated in Figure 3.3. The figure plots the three hillclimbing trajectories which produced the three bin-packing solutions shown above (Figure 3.2). Each visited state x_i is plotted as a point in a 2-D feature space where feature #1 is simply $\mathrm{Obj}(x_i)$ and feature #2 is the variance in fullness of the bins under packing x_i. In this feature space, the three trajectories all begin at the bottom left, which corresponds to the initial state shown in Figure 3.1: here, $\mathrm{Obj}= 30$ and the variance is low, since all the bins are nearly empty. The trajectories proceed rightward, as each accepted move reduces by one the number of bins used. The variance first increases, as some bins become fuller than others, and then decreases near the end of the trajectory as all the bins become rather full. The trajectories terminate at local minima of quality 11, 14, and 12, respectively.

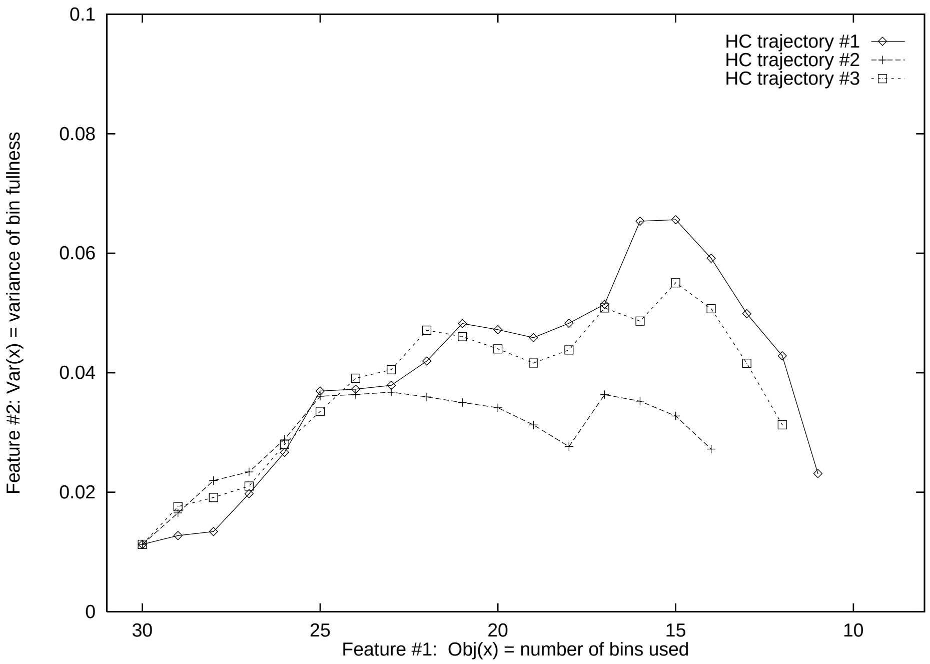


FIGURE 3.3. Three stochastic hillclimbing trajectories in bin-packing feature space

The key observation to make in Figure 3.3 is that the variance feature can help predict the future search outcome. Apparently, hillclimbing trajectories which pass through higher-variance states tend to reach better-quality solutions in the end. This makes sense: higher variance states have more nearly empty bins, which can be more easily emptied. This is the kind of knowledge that we would like to integrate into an improved evaluation function for local search.

It is well-known that extra features can be used to help improve the searchability of the cost surface. In simulated annealing practice, 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.* 88], 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, the following cost function is used:
>
> $$C = w^2 + \lambda_p \cdot p^2 + \lambda_U \cdot U \qquad (3.1)$$
>
> where ... λ_p and λ_U are constants, ...

In this application, the authors hand-tuned the coefficients and set $\lambda_p = 0.5$, $\lambda_U = 10$. (The next chapter will demonstrate an algorithm that discovered that much better performance can be achieved by assigning, counterintuitively, a *negative* value to λ_U.) Similar examples of evaluation functions being manually configured and tuned for good performance can be found in, e.g., [Cohn 92, Szykman and Cagan 95, Falkenauer and Delchambre 92].

The question I address is the following: can extra features of an optimization problem be incorporated automatically into improved evaluation functions, thereby guiding search to better solutions?

3.2 The "STAGE" Algorithm

This section introduces the algorithm which is the main contribution of this dissertation. **STAGE** applies the methods of value function approximation to automatically analyze sample trajectories, like those shown above in Figure 3.3, and to construct predictive evaluation functions. It then uses these new evaluation functions to guide further search. STAGE is general, principled, simple, and efficient. In the next chapter, I will also demonstrate empirically that it is successful at finding high-quality solutions to large-scale optimization problems.

3.2.1 Learning to Predict

STAGE aims to exploit a simple observation: the performance of a local search algorithm depends on the state from which the search starts. We can express this dependence in a mapping from starting states x to expected search result:

$$V^\pi(x) \stackrel{\text{def}}{=} \begin{array}{l}\text{expected best Obj value seen on a trajectory that starts}\\ \text{from state } x \text{ and follows local search method } \pi\end{array} \tag{3.2}$$

Here, π represents a local search method such as any of the hillclimbing variants or simulated annealing. Formal conditions under which V^π is well-defined will be given in Section 3.4.1. For now, the intuition is most important: $V^\pi(x)$ evaluates x's *promise* as a starting state for π.

For example, consider minimizing the one-dimensional function $\text{Obj}(x) = (|x| - 10)\cos(2\pi x)$ over the domain $X = [-10, 10]$, as depicted in Figure 3.4. Assuming a neighborhood 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 minimum reached does correlate strongly with the starting position x, making it possible to learn useful predictions.

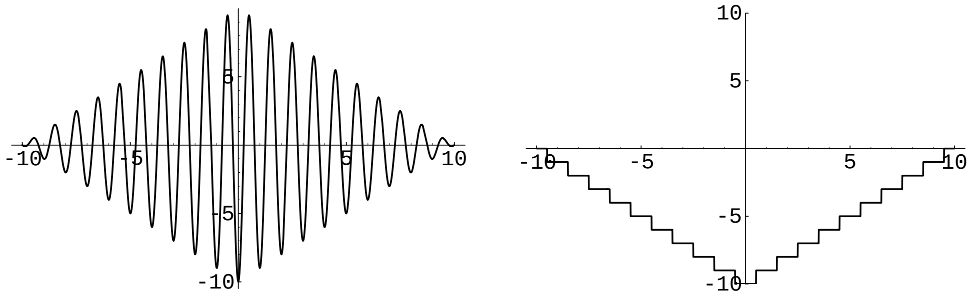


FIGURE 3.4. Left: Obj(x) for a one-dimensional function minimization domain. Right: the value function $V^\pi(x)$ which predicts hillclimbing's performance on that domain.

We seek to approximate V^π using a function approximation model such as linear regression or multi-layer perceptrons, where states x are encoded as real-valued feature vectors. As discussed above in Section 3.1.3, these input features may encode any relevant properties of the state, including the original objective function Obj(x) itself. We denote the mapping from states to features by $F : X \rightarrow \Re^D$, and our approximation of $V^\pi(x)$ by $\tilde{V}^\pi(F(x))$.

Training data for supervised learning of $\tilde{V}^\pi$ may be readily obtained by running π from different starting points. Moreover, if the algorithm π behaves as a Markov chain—i.e., the probability of moving from state x to x' is the same no matter when x is visited and what states were visited previously—then intermediate states of each simulated trajectory may also be considered alternate "starting points" for that search, and thus used as training data for $\tilde{V}^\pi$ as well. This insight enables us to get not one but perhaps hundreds of pieces of training data from each trajectory sampled. Under conditions which I detail in Section 3.4.1 below, all of the local search methods mentioned in Section 3.1.2 have the Markov property.

Under certain additional conditions, detailed in Section 3.4.1, the function V^π can be shown to be precisely the policy value function of a Markov chain. It is then possible to apply dynamic-programming-based algorithms such as TD(λ), which may learn more efficiently than supervised learning. I defer a detailed discussion of this approach until Section 6.1. For the remainder of this and the next two chapters, I will assume that V^π is approximated by supervised learning as outlined above.

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^π 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^π is fairly smooth, this hope is reasonable.

3.2.2 Using the Predictions

The learned evaluation function $\tilde{V}^\pi(F(x))$ evaluates how promising x is as a starting point for algorithm π. To find the best starting point, we must optimize $\tilde{V}^\pi$ over X. We do this by simply applying stochastic hillclimbing with $\tilde{V}^\pi$ instead of Obj as the evaluation function.[2]

The "STAGE" algorithm provides a framework for learning and exploiting $\tilde{V}^\pi$ on a single optimization instance. As illustrated in Figure 3.5, STAGE repeatedly alternates between two different stages of local search: running the original method π on Obj, and running hillclimbing on $\tilde{V}^\pi$ to find a promising new starting state for π. Thus, STAGE can be viewed as a *smart multi-restart* approach to local search.

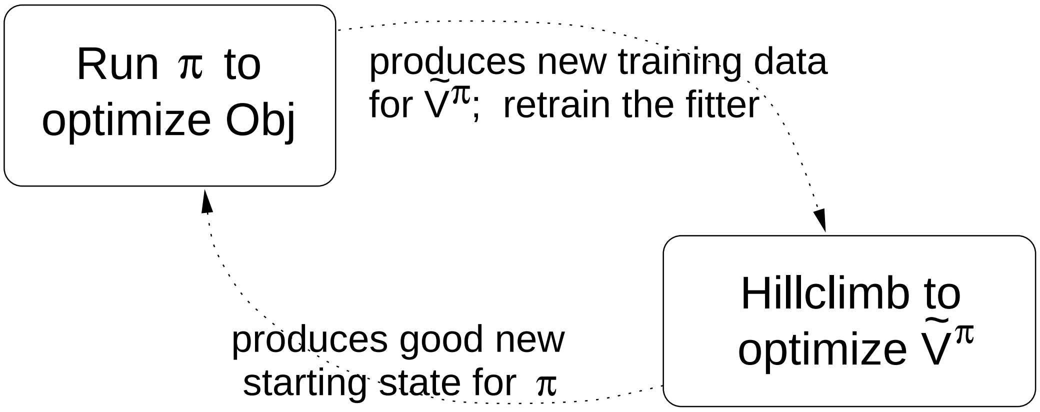


FIGURE 3.5. A diagram of the main loop of STAGE

A compact specification of the algorithm is given in Table 3.2.2 (p. 54). In the remainder of this section, I give a verbose description of the algorithm.

STAGE's inputs are X, S, π, Obj, ObjBound, F, *Fit*, N, Pat, and TotEvals:

- the state space X
- starting states $S \subset X$ (and a method for generating a random state in S)
- π, the local search method from which STAGE learns. π is assumed to be Markovian and proper, conditions which I discuss in detail in Section 3.4.1.

[2]Note that even if $\tilde{V}^\pi$ is smooth with respect to the feature space—as it surely will be if we represent $\tilde{V}^\pi$ with a simple model like linear regression—it may still give rise to a complex cost surface with respect to the neighborhood structure on X.

Intuitively, it is easiest to think of the prototypical case, hillclimbing. Note that π encapsulates the full specification of the method including all its internal parameters; for example, π = stochastic hillclimbing, rejecting equi-cost moves, with patience 200.

- the objective function, Obj : $X \to \Re$, to be minimized
- a lower bound on Obj, ObjBound $\in \Re$ (or $-\infty$ if no bound is known). Its use is described in step **2d** below.
- a featurizer F mapping states to real-valued features, $F : X \to \Re^D$
- *Fit*, a function approximator. Given a set of training pairs of the form $\{(f_{i1}, f_{i2}, \ldots, f_{iD}) \mapsto y_i\}$, *Fit* produces a real-valued function over $\Re^D$ which approximately fits the data.
- a neighborhood structure $N : X \to 2^X$ and patience parameter Pat for running stochastic hillclimbing on $\tilde{V}^\pi$
- TotEvals, the number of state evaluations allotted for this run.

STAGE's output is a single state $\tilde{x}$, which has the lowest Obj value of any state evaluated during the run. It also outputs the final learned evaluation function $\tilde{V}^\pi$, which provides interesting insights about what combination of features led to good performance. (If those insights apply generally to more than one problem instance, then the learned $\tilde{V}^\pi$ may be profitably *transferred*, as I show in Section 6.2.)

STAGE begins by initializing x_0 to a random start state. The main loop of STAGE proceeds as follows:

Step 2a: Optimize Obj using π. From x_0, run search algorithm π, producing a search trajectory $(x_0, x_1, x_2, \ldots, x_T)$.

Note that we have assumed π is a *proper* search procedure: the trajectory is guaranteed to terminate.

Step 2b: Train $\tilde{V}^\pi$. For each point x_i on the search trajectory, define $y_i := \min_{j=i\ldots T} \mathrm{Obj}(x_j)$, and add the pair $\{F(x_i) \mapsto y_i\}$ to the training set for *Fit*. Retrain *Fit*; call the resulting learned evaluation function $\tilde{V}^\pi$.

We accumulate a training set of all states ever visited by π. The target values y_i correspond to our definition of V^π in Equation 3.2 (p. 49). Note that for hillclimbing-like policies π which monotonically improve Obj, $y_i = Obj(x_T)\ \forall i$. Section 3.4.3 below shows how to make this step time- and space-efficient.

Step 2c: Optimize $\tilde{V}^\pi$ using hillclimbing. Continuing from x_T, optimize $\tilde{V}^\pi(F(x))$ by performing a stochastic hillclimbing search over the neighborhood structure N. Cut off the search when either PAT consecutive moves produce no improvement, or a candidate state z_{t+1} is predicted to be impossibly good, i.e. $\tilde{V}^\pi(F(z_{t+1})) <$ OBJBOUND. Denote this search trajectory by $(z_0, z_1, \ldots, z_t)$.

Note that this stochastic hillclimbing trajectory begins from where the previous π trajectory ended. This decision is evaluated empirically in Section 5.2.4. $\tilde{V}^\pi$ leads search to a state which is predicted to be a good new start state for π. We cut off the search if it reaches a state which promises a solution better than a known lower bound for the problem. For example, in bin-packing, we cut off if $\tilde{V}^\pi$ predicts that π will lead to a solution which uses a negative number of bins! This refinement can help prevent $\tilde{V}^\pi$ from leading search too far astray if the function approximation is very inaccurate. I show the empirical benefits of this refinement in Section 5.2.5.

Step 2d: Set smart restart state. Set $x_0 := z_t$. But in the event that the $\tilde{V}^\pi$ hillclimbing search accepted no moves (i.e., $z_t = x_T$), then reset x_0 to a new random starting state.

This reset operation is occasionally necessary to "un-stick" the search from a state which is a local optimum of both Obj and $\tilde{V}^\pi$. For example, on STAGE's first iteration, *Fit* has been trained on only one outcome of π, so $\tilde{V}^\pi$ will be constant—presenting no hill to climb. Lacking any information about which search directions lead to smart restart states, we restart randomly. This provision ensures that STAGE reverts to random multi-restart π in certain degenerate cases, e.g. if every state in X has identical features.

STAGE terminates as soon as TOTEVALS states have been evaluated. This count includes both accepted and rejected states considered during both the Step 2a search and the Step 2c search.

3.3 Illustrative Examples

We now illustrate STAGE's performance on the two sample domains described earlier in this chapter, the one-dimensional wave function and the small bin-packing problem.

3.3.1 1-D Wave Function

For the wave function example of Figure 3.4 (p. 50), the baseline search from which STAGE learns is hillclimbing with neighborhood moves of ± 0.1. We encode the

---

STAGE(X, S, π, Obj, ObjBound, F, *Fit*, N, Pat, TotEvals):

Given:

- a state space X
- starting states $S \subset X$ (and a method for generating a random state in S)
- a local search procedure π that is Markovian and proper (e.g., hillclimbing)
- an objective function, $\text{Obj} : X \to \Re$, to be minimized
- a lower bound on Obj, $\text{ObjBound} \in \Re$ (or $-\infty$ if no bound is known)
- a featurizer F mapping states to real-valued features, $F : X \to \Re^D$
- a function approximator *Fit*
- a neighborhood structure $N : X \to 2^X$ and patience parameter Pat for running stochastic hillclimbing on $\tilde{V}^\pi$
- TotEvals, the number of state evaluations allotted for this run

1. **Initialize** the function approximator; let $x_0 \in S$ be a random starting state for search.
2. **Loop** until number of states evaluated exceeds TotEvals:
   (a) **Optimize Obj using π.** From x_0, run search algorithm π, producing a search trajectory $(x_0, x_1, x_2, \ldots, x_T)$.
   (b) **Train $\tilde{V}^\pi$.** For each point x_i on the search trajectory, define $y_i := \min_{j=i\ldots T} \text{Obj}(x_j)$, and add the pair $\{F(x_i) \mapsto y_i\}$ to the training set for *Fit*. Retrain *Fit*; call the resulting learned evaluation function $\tilde{V}^\pi$.
   (c) **Optimize $\tilde{V}^\pi$ using hillclimbing.** Continuing from x_T, optimize $\tilde{V}^\pi(F(x))$ by performing a stochastic hillclimbing search over the neighborhood structure N. Cut off the search when either Pat consecutive moves produce no improvement, or a candidate state z_{t+1} is predicted to be impossibly good, i.e. $\tilde{V}^\pi(F(z_{t+1})) < \text{ObjBound}$. Denote this search trajectory by $(z_0, z_1, \ldots, z_t)$.
   (d) **Set smart restart state.** Set $x_0 := z_t$. But in the event that the $\tilde{V}^\pi$ hillclimbing search accepted no moves (i.e., $z_t = x_T$), then reset x_0 to a new random starting state.
3. **Return** the best state found.

Table 3.1. The STAGE algorithm

state using a single input feature, x itself, and we model $\tilde{V}^\pi$ by quadratic regression. Thus, STAGE will be building parabola-shaped approximations to the staircase-shaped true V^π. We assume no prior knowledge of a bound on the objective function, i.e., OBJBOUND $= -\infty$.

A sample run is depicted in Figure 3.6. The first iteration begins at a random starting state of $x_0 = 9.3$ and greedily descends to the local minimum at $(9, -1)$. In Step 2b, our function approximator trains on the trajectory's feature/outcome pairs: $\{\{9.3 \mapsto -1\}, \{9.2 \mapsto -1\}, \{9.1 \mapsto -1\}, \{9.0 \mapsto -1\}\}$ (shown in the diagram as small diamonds). The resulting least-squares quadratic approximation is, of course, the line $\tilde{V}^\pi = -1$. In Step 2c, hillclimbing on this flat function accepts no moves, so in Step 2d, we reset to a new random state—in this example run, $x_0 = 7.8$.

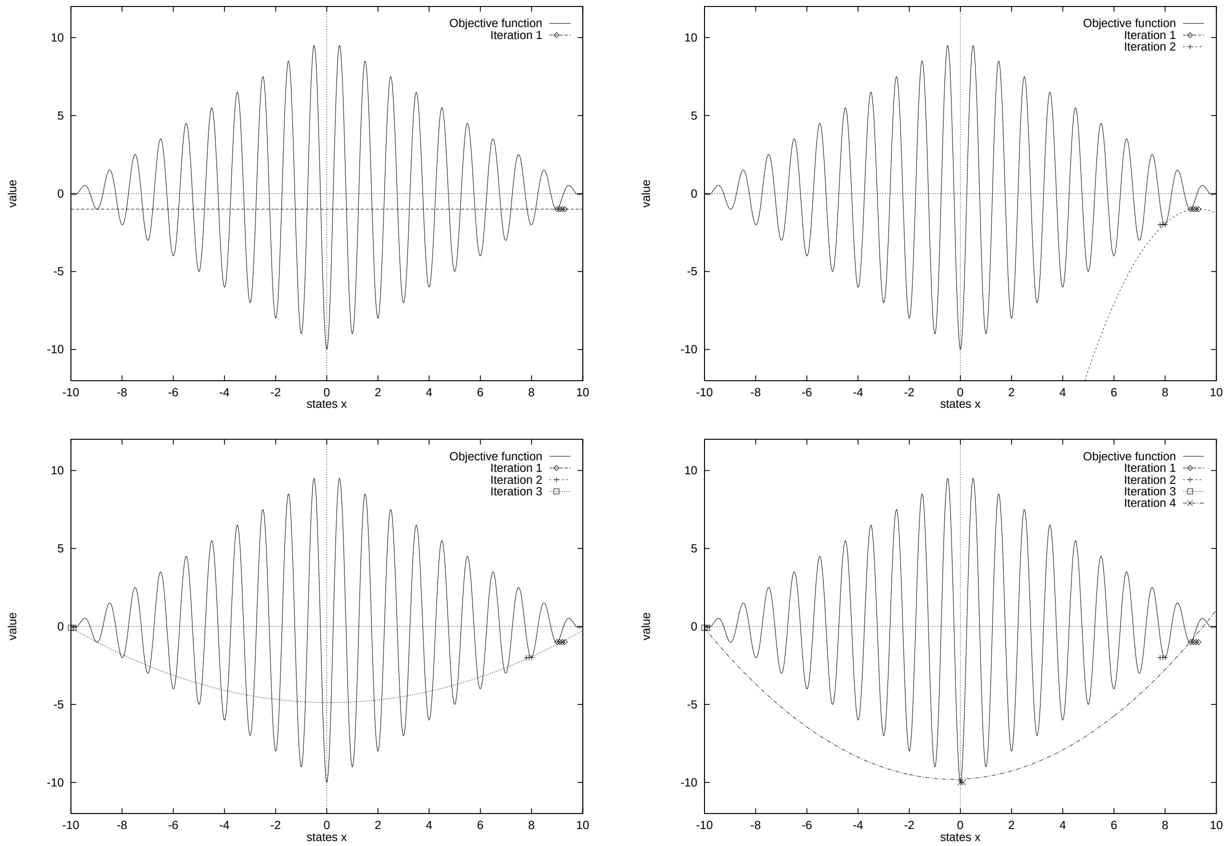


FIGURE 3.6. STAGE working on the 1-D wave example

On the second iteration (top right), greedy descent leads to the local minimum at $(8, -2)$, producing three new training points for $\tilde{V}^\pi$ (shown as small '+' symbols). The new best-fit parabola predicts that fantastically promising starting states can be found to the left. Hillclimbing on this parabola in Step 2c, we move all the way to the left edge of the domain, $x_0 = -10$, from which $\tilde{V}^\pi$ predicts that hillclimbing will produce an outcome of -212! (Note that with a known ObjBound, STAGE would have recognized $\tilde{V}^\pi$'s overoptimistic extrapolation and cut off hillclimbing before reaching the left edge. But no harm is done.)

The third iteration (bottom left) quickly punishes $\tilde{V}^\pi$'s overenthusiasm. Greedy descent takes one step, from $(-10, 0)$ to $(-9.9, -0.08)$. The two new training points, $(-10 \mapsto -0.08)$ and $(-9.9 \mapsto -0.08)$ (shown as squares), give rise to a nice concave parabola which correctly predicts that the best starting points for π are near the center of the domain. Hillclimbing on $\tilde{V}^\pi$ produces a smart restart point of $x_0 = 0.1$. From there, on iteration 4, π easily reaches the global optimum at $(0, -10)$. After ten iterations (bottom right), much more training data has been gathered near the center of the domain, and $\tilde{V}^\pi$ leads π to the global optimum every time.

This problem is contrived, but its essential property—that features of the state help to predict the performance of an optimizer—does indeed hold in many practical domains. Simulated annealing does not take advantage of this property, and indeed performs poorly on this problem. This problem also illustrates that STAGE does more than simply smoothing out the wiggles in Obj(x)—doing so here would produce an unhelpful flat function. STAGE does smooth out the wiggles, but in a way that incorporates predictive knowledge about local search.

3.3.2 Bin-packing

Our second illustrative domain, bin-packing, is more typical of the kind of practical combinatorial optimization problem we want STAGE to attack. We return to the example instance of Figure 3.1 (p. 44). The baseline search from which STAGE will learn is stochastic hillclimbing over the search neighborhood described on page 45. The starting state is the packing which places each item in its own separate bin.

Recall that in Figure 3.3 (p. 48), we observed that a simple state feature, variance in bin fullness levels, correlated with the quality of solution hillclimbing would eventually reach. To apply STAGE, we will use this feature and the true objective function to encode each state. We again model $\tilde{V}^\pi$ by quadratic regression over these two features. We assume no prior knowledge of bounds on the objective function, i.e., ObjBound $= -\infty$.

Snapshots from iterations 1, 2, 3 and 7 of a STAGE run are depicted in Figure 3.7. On the first iteration (top left plot), STAGE hillclimbs from the initial state $(\mathrm{Obj}(x) = 30, \mathrm{Var}(x) = 0.011)$ to a local optimum $(\mathrm{Obj}(x) = 13, \mathrm{Var}(x) = 0.019)$. Training each state of that trajectory to predict the outcome 13 results in a flat $\tilde{V}^\pi$ function (top right). Hillclimbing on this flat $\tilde{V}^\pi$ accepts no moves, so in Step 2d STAGE resets to the initial state.

On the second iteration of STAGE (second row of Figure 3.7), the new stochastic hillclimbing trajectory happens to do better than the first, finishing at a local optimum $(\mathrm{Obj}(x) = 11, \mathrm{Var}(x) = 0.022)$. Our training set is augmented with target values of 11 for all states on the new trajectory. The resulting quadratic $\tilde{V}^\pi$ already has significant structure. Note how the contour lines of $\tilde{V}^\pi$, shown on the base of the surface plot, correspond to smoothed versions of the trajectories in our training set. Extrapolating, $\tilde{V}^\pi$ predicts that the the best starting points for π are on arcs with higher Var(x).

STAGE hillclimbs on the learned $\tilde{V}^\pi$ to try to find a good starting point. The trajectory, shown as a dashed line in the third plot, goes from $(\mathrm{Obj}(x) = 11, \mathrm{Var}(x) = 0.022)$ up to $(\mathrm{Obj}(x) = 12, \mathrm{Var}(x) = 0.105)$. Note that the search was willing to accept some harm to the true objective function during this stage. From the new starting state, hillclimbing on Obj does indeed lead to a yet better local optimum at $(\mathrm{Obj}(x) = 10, \mathrm{Var}(x) = 0.053)$.

During further iterations, the approximation of $\tilde{V}^\pi$ is further refined. Continuing to alternate between standard hillclimbing on Obj (solid trajectories) and hillclimbing on $\tilde{V}^\pi$, STAGE manages to discover the global optimum at $(\mathrm{Obj}(x) = 9, \mathrm{Var}(x) = 0)$ on iteration seven. STAGE's complete trajectory is plotted at the bottom left of Figure 3.7. Contrast this STAGE trajectory with the multi-restart stochastic hillclimbing trajectory shown in Figure 3.8, which never reached any solution better than $\mathrm{Obj}(x) = 11$.

This example illustrates STAGE's potential to exploit high-level state features to improve performance on combinatorial optimization problems. It also illustrates the benefit of training $\tilde{V}^\pi$ on entire trajectories, not just starting states: in this run a useful quadratic approximation was learned after only two iterations. Extensive results on larger bin-packing instances, and on many other large-scale domains, are presented in Chapter 4.

3.4 Theoretical and Computational Issues

STAGE is a general algorithm: it learns to predict the outcome of a local search method π with a function approximator *Fit* over a feature space F. There are many choices for π, *Fit*, and F. This section describes how to make those choices so that STAGE is well-defined and efficient.

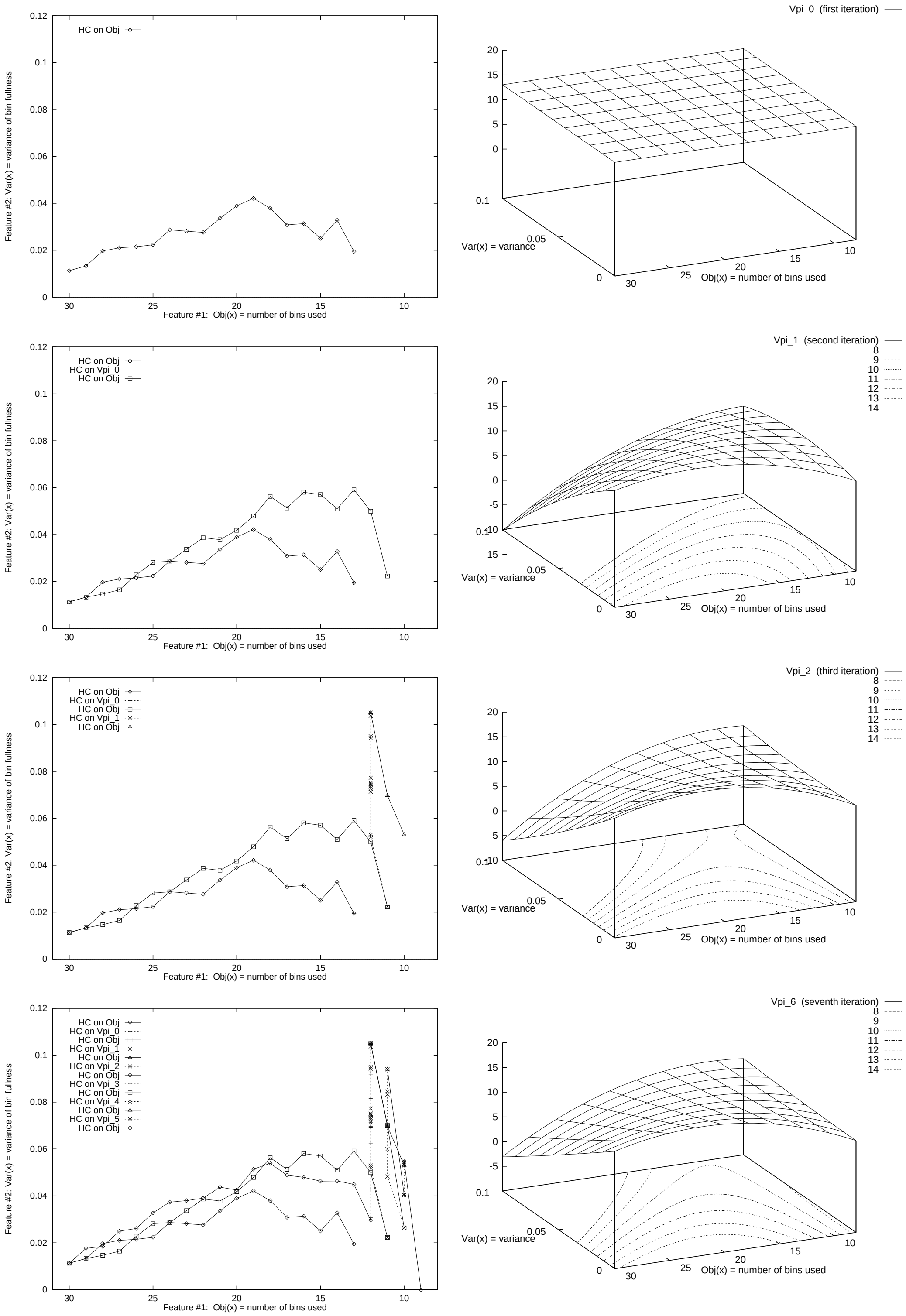


FIGURE 3.7. STAGE working on the bin-packing example

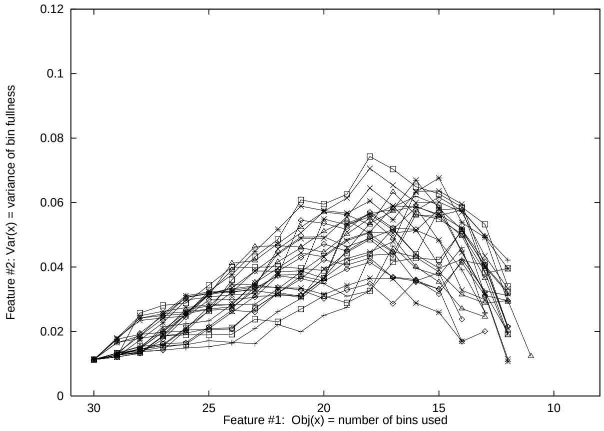


FIGURE 3.8. Trajectory of multi-restart hillclimbing in bin-packing feature space

3.4.1 Choosing π

STAGE learns from a baseline local search procedure π. Here, I identify precise conditions on π that make it suitable for STAGE. First, some definitions.

A run of a local search procedure π, starting from state $x_0 \in X$, stochastically produces a sequence of states called a *trajectory*, denoted $\tau \equiv (x_0^\tau, x_1^\tau, x_2^\tau, \dots)$. In general, the trajectory may be infinite, or it may terminate after a finite number of steps $|\tau|$. For any terminating trajectory, we define x_i^τ to equal a special state, denoted END, for all $i > |\tau|$.

Let $\mathcal{T}$ denote the set of all possible trajectories over X. Formally, π is characterized by the probability with which it generates each trajectory:

Definition 1. A *local search procedure* is a stochastic function $\pi : X \to \mathcal{T}$. Given a starting state $x \in X$, let $\mathrm{P}(\tau | x_0^\tau = x)$ denote the probability distribution with which π produces $\tau \in \mathcal{T}$.

I now define three key properties that π may have:

Definition 2. A local search procedure π is said to be *proper* if, from any starting state $x \in X$, the END state is eventually reached with probability 1.

Definition 3. A local search procedure π is said to be *Markovian* if, no matter when a state x_i is visited, a step to another state x_{i+1} occurs with the same fixed probability, denoted $p(x_{i+1}|x_i)$. In symbols: for all $i \in \mathbb{N}$ and $x_0, x_1, \dots, x_i, x_{i+1} \in X \cup \{\text{END}\}$,

$$\mathrm{P}(x_{i+1}^\tau = x_{i+1} \mid x_0^\tau = x_0, x_1^\tau = x_1, \dots, x_i^\tau = x_i) = p(x_{i+1}|x_i)$$

Definition 4. A local search procedure π is said to be *monotonic* with respect to Obj if, for every trajectory $\tau \in \mathcal{T}$ that π can generate,

$$\mathrm{Obj}(x_0^\tau) \geq \mathrm{Obj}(x_1^\tau) \geq \mathrm{Obj}(x_2^\tau) \geq \cdots$$

If all the inequalities are strict (until perhaps reaching an END state) for every trajectory, then π is called *strictly monotonic.*

Note that the conditions of being proper, Markovian, or monotonic are independent: a local search procedure may satisfy none, any one, any two, or all three definitions.

What conditions on π make it suitable for learning by STAGE? The essence of STAGE is to approximate the function $V^\pi(x)$, which predicts the expected best Obj value on a trajectory that starts from state x and follows procedure π. Formally, we define V^π as follows.

Definition 5. For any local search procedure π and objective function $\mathrm{Obj} : X \to \Re$, the function $V^\pi : X \to \Re$ is defined by

$$V^\pi(x) \stackrel{\text{def}}{=} \mathrm{E}\big[\inf\{\mathrm{Obj}(x_k^\tau) \mid k = 0, 1, 2, \dots\}\big],$$

where the expectation is taken over the trajectory distribution $\mathrm{P}(\tau | x_0^\tau = x)$.

Under one very reasonable assumption, we can show that V^π is well-defined for any policy π, i.e., that the expectation of Definition 5 exists:

Proposition 1. *If Obj is bounded below, then $V^\pi(x)$ is well-defined at every state $x \in X$ and for all policies π.*

Proof. Writing out the expectation of Definition 5, we have

$$V^\pi(x) = \sum_{\tau \in \mathcal{T}} P(\tau | x_0^\tau = x) \inf\{\mathrm{Obj}(x_k^\tau) \mid k = 0, 1, 2, \dots\}.$$

Each trajectory's infimum is bounded below by the assumed global bound on Obj, and bounded above by the value of the starting state, $\mathrm{Obj}(x_0^\tau) = \mathrm{Obj}(x)$. Thus, $V^\pi(x)$ is a convex sum of bounded quantities, which is a well-defined quantity. □

V^π is well-defined for any policy π, even improper ones. However, STAGE learns by collecting multiple sample trajectories of π; and in order for that to make sense, the trajectories must terminate. Hence, STAGE requires that π be proper (Definition 2). Later in this section, I will discuss several ways of turning improper policies into proper ones for use with STAGE.

STAGE also requires π to be Markovian. The reason for this condition is that when π is Markovian, STAGE can use the sample trajectory data it collects much more efficiently. The key insight is that in any trajectory $\tau = (x_0, x_1, \ldots, x_i, x_{i+1}, \ldots)$, each tail of the trajectory $(x_i, x_{i+1}, \ldots)$ is generated with exactly the same probability as if π had started a new trajectory from x_i. In other words, when π is Markovian, every state visited is effectively a new starting state. This means that in Step 2b of the STAGE algorithm (refer to page 54), STAGE can use every state of every sample trajectory as training data for approximating V^π.

Exploiting the Markov property can increase the amount of available training data by several orders of magnitude. In practice, though, the extra training points collected this way may be highly correlated, so it is unclear how much they will improve optimization performance. Section 5.2.3 shows empirically that the improvement can be substantial.

So far, we have required that π be proper for reasons of algorithmic validity and that π be Markovian for reasons of data efficiency. These are the only conditions imposed by the basic STAGE algorithm of page 54. However, we can impose the additional condition that π be monotonic, for reasons of memory efficiency. When π is monotonic, Markovian and proper, the infimum in the definition of V^π can be rewritten as an infinite sum:

$$\begin{aligned} V^\pi(x) &= \mathrm{E}\big[\inf\{\mathrm{Obj}(x^\tau_k) \mid k = 0,1,2,\ldots\}\big] \\ &= \mathrm{E}\big[\lim_{k\to\infty} \mathrm{Obj}(x^\tau_k)\big] \qquad \textit{(since } \pi \textit{ is monotonic)} \\ &= \mathrm{E}\left[\sum_{k=0}^{\infty} R(x^\tau_k, x^\tau_{k+1})\right] \qquad \textit{(since } \pi \textit{ is proper)} \end{aligned} \tag{3.3}$$

where all expectations are taken over the trajectory distribution $\mathrm{P}(\tau | x^\tau_0 = x)$, and the additive cost function R is defined by

$$R(x, x') \stackrel{\text{def}}{=} \begin{cases} \mathrm{Obj}(x) & \text{if } x \neq \text{END and } x' = \text{END}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

Writing V^π in this form reveals that it is precisely the policy value function of the Markov chain (X, π, R), as defined earlier in Section 2.1.1. This means that V^π satisfies the Bellman equations for prediction (Eq. 2.3), and all the algorithms of reinforcement learning apply. In particular, using the method of Least-Squares TD(λ), STAGE can learn a linear approximation to V^π without ever storing a trajectory in memory, thereby reducing memory usage significantly, and with no additional computational expense over supervised linear regression. The details of this technique are given in Section 6.1.

We have shown that in order for STAGE to learn from a local search procedure π, it is desirable for π to be proper, Markovian and monotonic. The remainder of this section investigates to what extent these conditions hold, or can be made to hold, for commonly used local search algorithms.

Steepest-descent hillclimbing. Steepest-descent takes a step from x to a neighboring state $x' \in N(x)$ which maximally improves over Obj(x). If no neighbors improve Obj, the trajectory terminates.

Steepest-descent is strictly monotonic. A strictly monotonic policy never visits the same state twice on any trajectory, so if X is finite (as is the case with combinatorial optimization problems), steepest-descent is guaranteed to terminate. Steepest-descent is also clearly Markovian: at a local optimum it terminates deterministically, and from any other state it steps with equal probability to any $x' \in N(x)$ for which $\mathrm{Obj}(x') = \min_{z \in N(x)} \mathrm{Obj}(z)$.

Thus, steepest-descent procedures are strictly monotonic, Markovian, and (if X is finite) proper.

Stochastic hillclimbing. For search problems where the neighborhoods $N(x)$ are large, stochastic hillclimbing is cheaper to run than steepest-descent. We consider first the case where equi-cost moves are rejected, that is, a move from x to x' is accepted only if x' belongs to the set $G(x) \stackrel{\text{def}}{=} \{g \in N(x) : \mathrm{Obj}(g) \lneqq \mathrm{Obj}(x)\}$. Let HC represent this procedure for a given state space X, neighborhood function N, objective function Obj, and patience value PAT.

HC is strictly monotonic. As above, assuming X is finite, HC is guaranteed to terminate. HC is also Markovian, with the following transition probabilities:

$$p(\textsc{end}|x) = \left(\frac{|N(x)|-|G(x)|}{|N(x)|}\right)^{\textsc{Pat}}$$

$$p(x'|x) = \begin{cases} \frac{1}{|G(x)|}\big(1 - p(\textsc{end}|x)\big) & \text{if } x' \in G(x) \\ 0 & \text{if } x' \notin G(x) \cup \{\textsc{end}\}. \end{cases}$$

These transition probabilities assume that all neighbors are equally likely to be sampled; it is straightforward to reweight them in the case of non-uniform sampling distributions.

However, if we drop the assumption of rejecting equi-cost moves, stochastic hillclimbing with patience-based termination is no longer Markovian. The reason is that after an equi-cost move, the patience counter is not reset to zero, so $p(\textsc{end}|x)$ is not fixed but rather depends on the previous states visited. Possibility 3 listed in the next paragraph describes a remedy.

Biased random walks. This general category includes any local search procedure where state transitions are memoryless: stochastic hillclimbing with or without equi-cost moves; force-best-move; GSAT and WALKSAT; and random walks in the state space X. These procedures are Markovian over X but, in general, not proper. To make them proper, a termination condition must be specified. I consider three possibilities:

Possibility 1: Run the procedure for a fixed number of steps.

This destroys the Markov condition, since the termination probability $p(\text{END}|x)$ depends not just on x but on the global step counter. It is possible, however, to include this counter as part of the state, as I will discuss under the heading of Simulated Annealing below.

Possibility 2: Introduce a termination probability $\epsilon > 0$.

If $p(\text{END}|x) \geq \epsilon$ for every state x, then the procedure remains Markovian and becomes proper, thus making it suitable for STAGE. However, this approach may randomly cause termination to occur during a fruitful part of the search trajectory.

Possibility 3: Use patience-based termination and the best-so-far abstraction.

This approach means cutting off search after PAT consecutive steps have failed to improve on the best state found so far on the trajectory. This makes the search procedure proper if $|X|$ is finite, but breaks the Markov property, since $p(\text{END}|x)$ depends on not just x but also the current patience counter and the best Obj value seen previously. However, we can use a simple trick to reclaim the Markov property.

Definition 6. Given a local search procedure π, the *best-so-far abstraction* of this policy is a new policy $\text{BSF}(\pi)$ which filters out all but the best-so-far states on each trajectory produced by π. That is, if π produces $\tau = (x_0^\tau, x_1^\tau, x_2^\tau, \dots)$ with probability $P(\tau|x_0^\tau = x_0)$, then with that same probability, $\text{BSF}(\pi)$ produces

$$\tau' = (x_{i_0}^\tau, x_{i_1}^\tau, x_{i_2}^\tau, \dots)$$

where $(i_0, i_1, i_2, \dots)$ is the subsequence consisting of all indices that satisfy

$$\text{Obj}(x_{i_k}) < \min_{\{j: 0 \leq j < i_k\}} \text{Obj}(x_j)$$

For example, given a trajectory of π with associated Obj values:

$$\begin{array}{cccccccccc} x_0, & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7, & x_8, & \text{END} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 42, & 44, & 39, & 31, & 35, & 31, & 40, & 28, & 30 & \end{array}$$

the procedure $\text{BSF}(\pi)$ would produce the monotonic trajectory

$$\begin{array}{cccccccccc} x_0, & & x_2, & x_3, & & & & x_7, & & \text{END} \\ \downarrow & & \downarrow & \downarrow & & & & \downarrow & & \\ 42, & & 39, & 31, & & & & 28 & & \end{array}$$

Given this definition, we can prove the following:

Proposition 2. *If local search procedure π is Markovian over a finite state space X, and π' is the procedure that results by adding patience-based termination to π, then procedure $\text{BSF}(\pi')$ is proper, Markovian, and strictly monotonic.*

The proof is given in Appendix A.1. In practical terms, this means we can run STAGE just as described on page 54, except that in Step 2b, we train the fitter on only the best-so-far states of each sample trajectory. Compared with random termination (Possibility 2), patience-based termination vastly reduces the number of training samples we collect for fitting V^π, but gives us both a more natural cutoff criterion and the monotonicity needed to apply reinforcement-learning methods. In Section 4.7, I compare these two possibilities empirically on the domain of Boolean satisfiability.

Simulated annealing. The steps taken during simulated annealing search depend on not only the current state but also a time-varying *temperature* parameter $t_i > 0$. In particular, from state x_i at time i, simulated annealing evaluates a random neighbor $x' \in N(x_i)$ and sets

$$x_{i+1} := \begin{cases} \text{END} & \text{if } i > \text{TotEvals} \\ x' & \text{if RAND} < e^{[\text{Obj}(x_i) - \text{Obj}(x')]/t_i} \\ x_i & \text{otherwise} \end{cases} \tag{3.5}$$

where RAND is a random variable uniformly chosen from $[0, 1)$. The temperature t_i decreases over time. At high temperatures, moves that worsen the objective function even by quite a lot are often accepted, whereas at low temperatures, worsening moves are usually rejected. Improving moves and equi-cost moves are always accepted no matter what the temperature.

Because of the dependence on temperature, simulated annealing is not Markovian. However, any local search procedure can be made Markovian by augmenting the state space with whatever extra variables are relevant to future transition probabilities. In the case of simulated annealing, if the temperature schedule t_i is fixed in advance, it suffices to augment X with a single variable $i \in \mathbb{N}$, the move counter. For example, if π uses the schedule

$$t_i := \begin{cases} 2.0 - i/1000 & \text{if } i < 1000 \\ 0.99^{(i-1000)} & \text{if } i \geq 1000 \end{cases},$$

which decays the temperature linearly and then exponentially, then π is Markovian in $X \times \mathbb{N}$. In the expanded space, $V^\pi(x, i)$ predicts the outcome of simulated annealing when search starts from state x at time i.

However, this formulation is of limited usefulness to STAGE. For any fixed x, we expect the best value of V^π to occur at $i = 0$, since search should always benefit from having more time remaining. Thus, in Step 2c, STAGE can fix $i = 0$ while searching for a good starting point; but then there is little benefit in having trained on all the simulated annealing trajectory states with $i > 0$. In other words, it would seem that to apply the basic STAGE algorithm to simulated annealing, one may as well train on only the actual starting state x_0 of each trajectory, and forego the improved data efficiency that the Markov assumption usually brings. I test this empirically in Section 5.2.3. Later, in Section 8.2.2, I also discuss a modified version of STAGE which allows simulated annealing to exploit V^π more fully.

This section has analyzed a variety of hillclimbing and random-walk local search procedures from which STAGE can learn. From a theoretical point of view, the ideal procedure should be proper, Markovian, and monotonic. From a practical point of view, the procedure should also be (1) effective at finding good solutions on its own, so STAGE begins from a high performance baseline; and (2) predictable, so that V^π has learnable structure. In practice, stochastic hillclimbing rejecting equi-cost moves seems to be a good choice; it is used for the bulk of the results in Chapter 4. Alternative choices for π are explored in Section 4.7 (π = WALKSAT) and Section 5.2.3 (π = simulated annealing).

3.4.2 Choosing the Features

STAGE approximates V^π with statistical regression over a real-valued feature representation of the state space X, $\tilde{V}^\pi(F(x)) \approx V^\pi(x)$. Clearly, the quality of the approximation will depend on the feature representation we choose. As with any function approximation task, the features are "usually handcrafted, based on whatever human intelligence, insight, or experience is available, and are meant to capture the most important aspects of the current state" [Bertsekas and Tsitsiklis 96].

As discussed earlier in Section 3.1.3, most practical problems are awash in features that could help predict the outcome of local search. In the bin-packing example of Section 3.1.3, we listed half a dozen plausible features, such as variance in bin fullness. For a travelling salesperson problem, some reasonable features of a tour x might include

- $\mathrm{Obj}(x)$ = the sum of the intercity distances in x
- the variance of the distances in x. (This could identify whether tours with some short and some long hops are more or less promising than tours with mostly medium-length hops.)
- the number of improving steps in the search neighborhood of x. (The general usefulness of this feature as a local search heuristic has been investigated by [Moll *et al.* 97].)
- for each city c, the distance to the next city c' assigned by x. (These fine-grained features nearly specify x completely, but may be too numerous for efficient learning.)
- geometric features of the tour (if applicable), such as the average bend angle at each city or average Δx and Δy of the distances in x.

Empirically, STAGE seems to do at least as well as random multi-restart π no matter what features are chosen. Note that if *no* features are used, $\tilde{V}^{\pi}$ is always constant, and STAGE reduces to random multi-restart π. I generally choose just a few coarse, simple-to-compute features of a problem space, such as the variance features mentioned above or subcomponents of the objective function. Using only a few features minimizes the computational overhead of training $\tilde{V}^{\pi}$, and works well in practice. Chapter 4 gives many more examples of effective feature sets for large-scale domains, and Section 5.2.1 investigates the empirical effect of using different feature sets.

3.4.3 Choosing the Fitter

STAGE relies on a function approximator *Fit* to produce $\tilde{V}^{\pi}$ from sample training data. Examples of function approximators include polynomial regression; memory-based methods such as k-nearest-neighbor and locally weighted regression [Cleveland

and Devlin 88]; neural networks such as multi-layer perceptrons [Rumelhart *et al.* 86], radial basis function networks [Moody and Darken 89], and cascaded architectures [Fahlman and Lebiere 90]; CMACs [Albus 81]; multi-dimensional splines [Friedman 91]; decision trees such as CART [Breiman *et al.* 84]; and many others.

What qualities of *Fit* make it most suitable for STAGE? The most important requirements are the following:

Incremental STAGE trains on many states—perhaps on the order of millions—during the course of an optimization run, so the fitter must be able to handle large quantities of data without an undue memory or computational burden. Training occurs once per STAGE iteration and must be efficient. Evaluating the learned function occurs on every step of hillclimbing on $\tilde{V}^{\pi}$ and must be very efficient. In the terminology of [Sutton and Whitehead 93], the fitter must be *strictly incremental.*

Noise-tolerant The training values STAGE collects are the outcomes of long stochastic search trajectories. Thus, the fitter must be able to tolerate substantial noise in the training set.

Extrapolating STAGE hillclimbs on the learned function in search of promising, previously unvisited states. Thus, STAGE can benefit from a fitter that extrapolates trends from the training samples. Figure 3.9 contrasts the fits learned by quadratic regression and 1-nearest-neighbor on a small one-dimensional training set. Even though the quadratic approximation has worse residual error on the training samples, it is more useful for STAGE's hillclimbing. Note that STAGE's ObjBound cutoff helps compensate in cases where the fitter over-extrapolates.

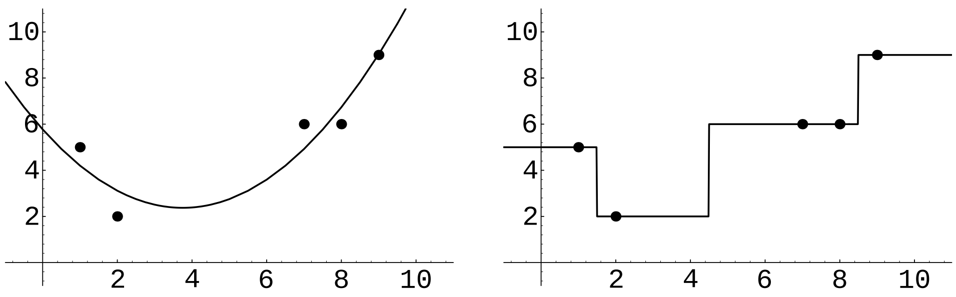


Figure 3.9. Quadratic regression and 1-nearest-neighbor

Given these requirements, the ideal function approximators for STAGE are those in the class of *linear architectures*. Following the development of [Bertsekas and Tsitsiklis 96], a general linear architecture has the form

$$\tilde{V}(x, \boldsymbol{\beta}) = \sum_{k=1}^{K} \beta[k]\phi_k(x) \tag{3.6}$$

where $\beta[1], \beta[2], \ldots, \beta[K]$ are the components of the coefficient vector $\boldsymbol{\beta}$, and the ϕ_k are fixed, easily computable real-valued *basis functions*. For example, if the state x ranges over $\Re$ and $\phi_k(x) = x^{k-1}$ for each $k = 1 \ldots K$, then $\tilde{V}$ represents a polynomial in x of degree $K-1$. Other examples of linear architectures include CMACs, random representation networks [Sutton and Whitehead 93], radial basis function networks with fixed basis centers, and multi-dimensional polynomial regression. Figure 3.10 illustrates the chain of mappings by which STAGE produces a prediction $\tilde{V}^\pi$ from an optimization state x.

The particular linear architecture I prefer for STAGE is quadratic regression. Quadratic regression produces $K = (D+1)(D+2)/2$ basis functions from the features $F(x) = f_1, f_2, \ldots, f_D$:

$$\Phi(F(x)) = (1, f_1, f_2, \ldots, f_D,\ f_1^2, f_1 f_2, \ldots, f_1 f_D,\ f_2^2 \ldots, f_2 f_D,\ \ldots,\ f_D^2) \tag{3.7}$$

Quadratic regression is flexible enough to capture global first-order and second-order trends in the feature space and to represent a global optimum at any point in feature space, but also biased enough to smooth out significant training set noise and to extrapolate aggressively.

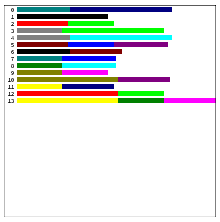
state	$\mapsto$	features	$\mapsto$	basis vector	$\mapsto$	prediction
$x \in X$	$\mapsto$	$F(x) \in \Re^D$	$\mapsto$	$\Phi(F(x)) \in \Re^K$	$\mapsto$	$\boldsymbol{\beta} \cdot \Phi(F(x)) \in \Re$
	$\mapsto$	$\{Obj = 14,$ $Var = 0.027\}$	$\mapsto$	$\{1, 14, 0.027,$ $196, 0.381,$ $0.000741\}$	$\mapsto$	12.4

FIGURE 3.10. Approximation of $\tilde{V}^\pi$ by a linear architecture (quadratic regression)

Linear architectures meet the "strictly incremental" requirement: compared with any of the other function approximators listed at the beginning of this section, training them is very efficient in time and memory. Let our training set be denoted by

$$\{\boldsymbol{\phi}_{(1)} \mapsto y_1, \boldsymbol{\phi}_{(2)} \mapsto y_2, \ldots, \boldsymbol{\phi}_{(N)} \mapsto y_N\}$$

where $\boldsymbol{\phi}_{(i)} = (\phi_1(F(x_i)), \ldots, \phi_K(F(x_i)))$ is the i^{th} training basis vector, and y_i is the i^{th} target output. The goal of training is to find coefficients that minimize the squared residuals between predictions and target values; that is,

$$\boldsymbol{\beta}^* = \underset{\beta \in \Re^K}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y_i - \beta \cdot \boldsymbol{\phi}_{(i)}\right)^2$$

Finding the optimal coefficients $\boldsymbol{\beta}^*$ is a *linear least squares* problem that can be solved by efficient linear algebra techniques. The sufficient statistics for $\boldsymbol{\beta}^*$ are the matrix $\mathbf{A}$ (of dimension $K \times K$) and vector $\mathbf{b}$ (of length K), computed as follows:

$$\mathbf{A} = \sum_{i=1}^{N} \phi_{(i)} \phi_{(i)}^{\mathsf{T}} \qquad\qquad \mathbf{b} = \sum_{i=1}^{N} \phi_{(i)} y_i \tag{3.8}$$

Given this compact representation of the training set, the coefficients of the linear fit can be calculated as

$$\boldsymbol{\beta}^* = \mathbf{A}^{-1}\mathbf{b} \tag{3.9}$$

Singular Value Decomposition is the method of choice for inverting $\mathbf{A}$, since it is robust when $\mathbf{A}$ is singular [Press *et al.* 92].

On each iteration of STAGE, many new training samples are added to the training set, and then the function approximator is re-trained once. With a linear architecture, adding new training samples is simply a matter of incrementing the $\mathbf{A}$ matrix and $\mathbf{b}$ vector of Equation 3.8. For each sample, this incurs a cost of $O(K^2)$ to compute the outer product $\boldsymbol{\phi}_{(i)}\boldsymbol{\phi}_{(i)}^{\mathsf{T}}$ and $O(K)$ to compute $\boldsymbol{\phi}_{(i)} y_i$. The training samples can then be discarded. Updated values of $\boldsymbol{\beta}^*$ are computed by Equation 3.9 in $O(K^3)$ time, independent of the number of samples in the training set.

Linear architectures also have the advantage of low memory use. Between iterations of STAGE, we need store only $\mathbf{A}$ and $\mathbf{b}$, not the whole training set. Therefore, the bottleneck on memory usage is the space required to store the state features along any single trajectory during Step 2a (refer to page 54). This is usually quite modest, but can become significant for local search procedures which visit tens of thousands of states on a single trajectory. In such cases, the Least-Squares TD(1) algorithm can be used. Least-Squares TD(1) produces the same coefficients $\boldsymbol{\beta}^*$ as Equation 3.9 above with the same amount of computation, but requires *no* memory for saving trajectories: it performs its computations fully incrementally as the trajectory is generated. Least-Squares TD(1) may be applied in STAGE when the baseline local search procedure π is monotonic; the algorithm is described in its general TD(λ) form in Section 6.1.

3.4.4 Discussion

This section has considered the theoretical and computational issues that arise in choosing a local search procedure π, feature mapping F, and function approximator *Fit* for use by STAGE. To summarize the practical conclusions of this section:

1. A good choice for π is stochastic hillclimbing, rejecting equi-cost moves, with patience-based termination. This procedure is proper, Markovian, monotonic, and easy to apply in almost any domain.

2. Features $F(x)$ of a state x must be hand chosen, but are generally abundant. A good choice is to use a few simple, coarse features such as subcomponents of Obj(x).

3. A good choice for *Fit* is quadratic regression. Its training time and memory requirements are small and independent of the number of training samples.

STAGE has two further inputs that have not yet been discussed: N and PAT, the neighborhood structure and patience parameter used for stochastic hillclimbing on the learned evaluation function $\tilde{V}^{\pi}$. In general, I simply set these to the same neighborhood structure and patience parameter which were used to define π.

Chapter 4

STAGE: Empirical Results

The last chapter introduced STAGE, an optimization algorithm that learns to incorporate extra features of a problem into an evaluation function and thereby improve overall performance. In this chapter, I first define my methodology for measuring STAGE's performance and comparing it to other algorithms empirically. I then describe my implementations of seven large-scale optimization domains with widely varying characteristics:

- **Bin-packing (§4.2)**: pack a collection of items into as few bins as possible—the classic NP-complete problem, as discussed in the examples of Chapter 3;

- **Channel routing (§4.3)**: minimize the area needed to produce a specified circuit in VLSI;

- **Bayes net structure-finding (§4.4)**: determine the optimal graph of data dependencies between variables in a dataset;

- **Radiotherapy treatment planning (§4.5)**: given an anatomical map of a patient's brain tumor and nearby sensitive structures, plan a minimally harmful radiation treatment;

- **Cartogram design (§4.6)**: for geographic visualization purposes, redraw a map of the United States so that each state's area is proportional to its population, minimizing deformations;

- **Satisfiability (§4.7)**: given a Boolean formula, find a variable assignment that makes the formula true; and

- **Boggle board setup (§4.8)**: find a 5×5 grid of letters containing as many English words in connected paths as possible.

For each of these domains, I apply STAGE as described in Section 3.2, unmodified, and report statistically significant results.

4.1 Experimental Methodology

The effectiveness of a heuristic optimization algorithm should be measured along three main dimensions [Barr *et al.* 95, Johnson 96]:

- **Solution quality.** How close are the solutions found by the method to optimality?
- **Computational effort.** How long did the method take to find its best solution? How quickly does it find good solutions?
- **Robustness.** Does the method work across multiple problem instances and varying domains? In the case of a randomized method, are its results consistent when applied repeatedly to the same problem?

This chapter evaluates the performance of STAGE in each of these dimensions through a statistical analysis of the results of thousands of experimental runs. STAGE's results are contrasted to the optimal solution (if available), special-purpose algorithms for each domain (if available), and two general-purpose reference algorithms: multiple-restart stochastic hillclimbing and simulated annealing.

4.1.1 Reference Algorithms

I compare STAGE's performance to that of multi-restart stochastic hillclimbing and simulated annealing on every problem instance. Multi-restart stochastic hillclimbing is not only straightforward to implement, but also a surprisingly effective heuristic on some domains. For example, hillclimbing with 100 random restarts is generally adequate for finding high-quality solutions to a geometric line-matching problem [Beveridge *et al.* 96]. For the comparative experiments of this chapter, I run hillclimbing with the following parameters:

- At each step, moves are randomly chosen in a problem-dependent way. The first such move that either improves Obj or keeps it the same is accepted.
- The *patience* parameter is set individually for each problem; generally the best setting is on the same order of magnitude as the number of available actions. Search restarts whenever *patience* consecutive moves have been evaluated since finding the state whose objective function value is best on the current trajectory.
- Search restarts at either a special start state or a random state; this distribution is also set individually for each domain.

- The total number of moves evaluated is limited to TotEvals, the same parameter that governs STAGE's termination.

Conceptually, simulated annealing is only slightly more complicated: it accepts all improving and equi-cost moves, but also probabilistically accepts some worsening moves. In practice, though, implementing an effective *temperature annealing schedule* which regulates the evolution of the acceptance probabilities can be difficult. Based on a review of the literature and a good deal of experimentation, I implemented Swartz's modification of the Lam temperature schedule [Swartz and Sechen 90, Lam and Delosme 88]. This adaptive schedule produces excellent results across a variety of domains. Appendix B provides detailed justification for and implementation details of this "modified Lam" schedule.

4.1.2 How the Results are Tabulated

For the purpose of summarizing the solution quality, computational effort, and robustness of STAGE and competing heuristics, it would be ideal to plot each algorithm's average performance versus elapsed time. However, my experiments were run on a pool of over 100 workstations having widely varying job loads, processor speeds, and system architectures; collecting meaningful average timing measurements was therefore impossible. Instead, for each algorithm I plot average performance versus *number of moves considered.* In practical applications where evaluating Obj is relatively costly, this quantity correlates strongly with total running time, yet is independent of machine speed and load. I also do give sample timing comparisons on each domain, measured on an SGI Indigo2 R10000 workstation, but since this workstation is multi-user, these figures should be considered very rough. Each time reported is the median of three independent runs.

Note that for STAGE, the number of moves considered includes moves made during both stages of the algorithm, i.e., both running π and optimizing $\tilde{V}^{\pi}$. However, this number does not capture STAGE's additional overhead for feature construction and function approximator training. With linear approximation architectures (see Section 3.4.3) and simple features, this overhead is minimal—typically, less than 10% of the execution time.

On any single run of a search algorithm, after n moves have been considered, the performance is defined as the best Obj value found up to that point. The overall performance of the algorithm at time n, $Q(n)$, is defined as the expected single-run performance:

$$Q(n) \stackrel{\text{def}}{=} \mathrm{E}\{\min_{i=0\ldots n} \mathrm{Obj}(x_i)\}$$

where the expectation is taken over all trajectories $(x_0, x_1, \dots)$ that the algorithm may generate on the problem. For experimental evaluation purposes, I sample $Q(n)$ by taking the mean best value seen over multiple independent runs. The variable n ranges from 0 to TotEvals. Note that $Q(n)$ is non-increasing regardless of whether the search algorithm monotonically improves Obj.

Figure 4.1 illustrates how a performance curve is produced for an algorithm and a problem. In this example, a search algorithm is run five times for TotEvals = 500 steps each run. The upper-left graph plots the objective function value of the states visited during the course of these five runs. The upper-right graph plots the performance of each run, that is, the best Obj value seen so far at each step. The third graph, at bottom left, plots the mean performance of the five runs over time. Finally, the fourth graph summarizes the algorithm's overall performance in a boxplot. The boxplot is calculated at the endpoint of the runs, where $n =$ TotEvals. It gives the 95% confidence interval of the mean performance (shown as a box around the mean)[1] and the end result of the best and worst of the runs (shown as "whiskers").

For each comparative experiment in this chapter, the results are presented in two figures and a table:

- a mean performance curve, like the lower left graph of Figure 4.1, indicates how quickly each algorithm reached its best performance level;

- a boxplot, like the lower right graph of Figure 4.1, provides a useful visual means for comparing the algorithms against one another [Barr *et al.* 95]; and

- a results table numerically displays the same data as the boxplot. In each table, the best minimum, best maximum, and statistically best mean performances are boldfaced. Each table also reports the running time and overall percentage of $\frac{\text{moves accepted}}{\text{TotEvals}}$ for each algorithm, but these figures are medians of only three runs and should thus be considered rough estimates.

I now proceed to describe the various optimization domains, experiments, and results by which I have evaluated STAGE.

4.2 Bin-packing

Bin-packing, a classical NP-hard optimization problem [Garey and Johnson 79], has often been used as a testbed for combinatorial optimization algorithms (e.g.,

[1]The 95% confidence interval of the mean is simply 2 standard errors on either side: $\mu \pm \frac{2\sigma}{\sqrt{N}}$.

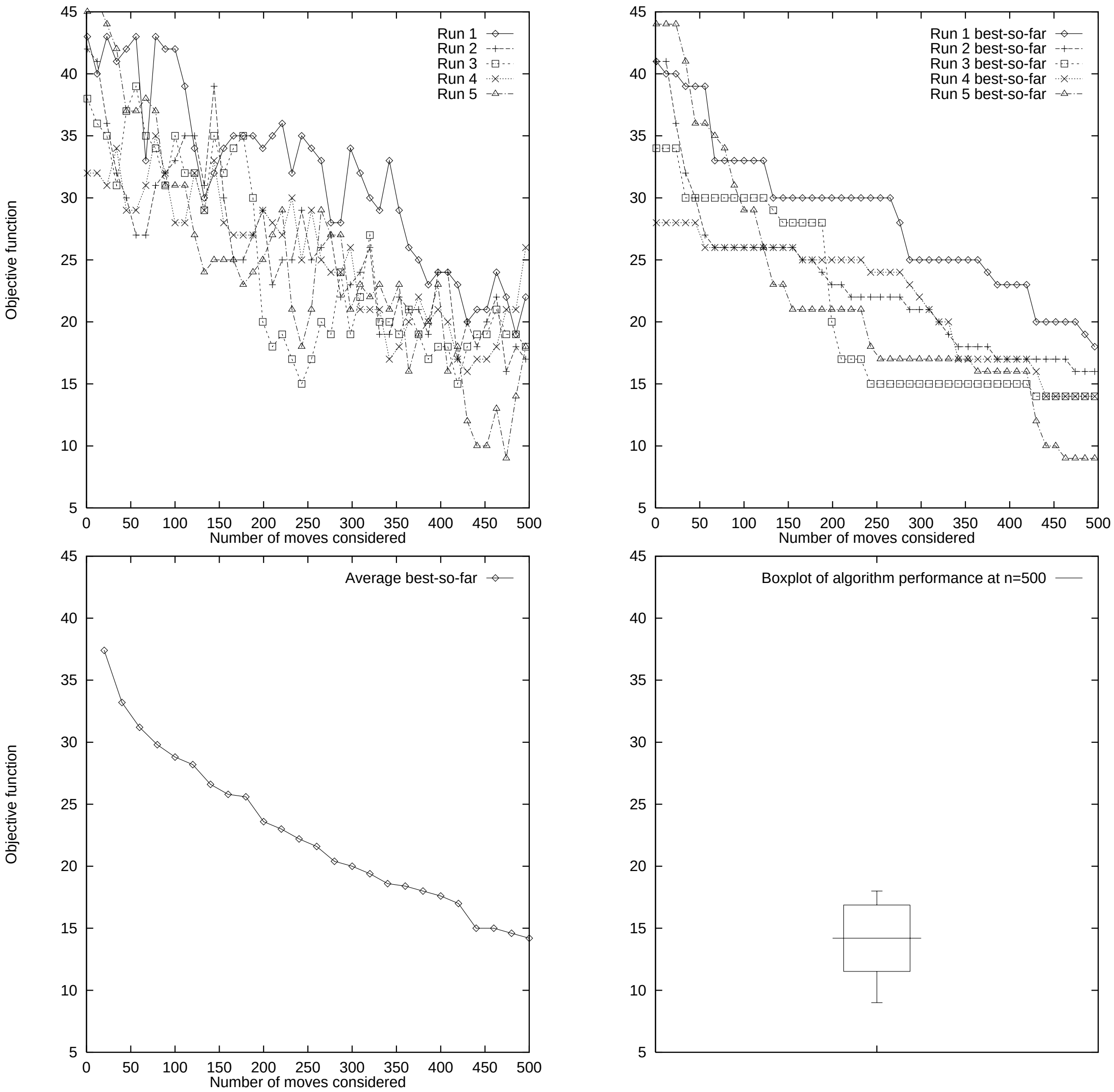


FIGURE 4.1. From a number of independent runs of an algorithm (upper left) and their best-so-far curves (upper right), a single performance curve (lower left) and box plot (lower right) are generated to summarize the algorithm's performance. Please refer to the text for a detailed explanation.

[Falkenauer 96, Baluja and Davies 97]). The problem was introduced earlier in Chapter 3. Recall that we are given a *bin capacity* C and a list $L = (a_1, a_2, ...a_n)$ of *items*, each having a *size* $s(a_i) > 0$. The goal is to pack the items into as few bins as possible, i.e., partition them into a minimum number m of subsets $B_1, B_2, ..., B_m$ such that for each B_j, $\sum_{a_i \in B_j} s(a_i) \leq C$.

In this section, I present results on benchmark problem instances from the Operations Research Library (see Appendix C.1 for details). The first instance considered, u250_13 [Falkenauer 96], has 250 items of sizes uniformly distributed in $(20, 100)$ to be packed into bins of capacity 150. The item sizes sum to 15294, so a lower bound on the number of bins required is $\lceil \frac{15294}{150} \rceil = 102$.

Falkenauer reported excellent results—on this problem, a solution with only 103 bins—using a specially modified search procedure termed the "Grouping Genetic Algorithm" and a hand-tuned objective function:

> We thus settled for the following cost function for the BPP [binpacking problem]: maximize $f_{\text{BPP}} = \frac{\sum_{i=1}^{m}(\text{fill}_i/C)^k}{m}$ with m being the number of bins used, fill_i the sum of sizes of the objects in the bin i, C the bin capacity and k a constant, $k > 1$.... The constant k expresses our concentration on the well-filled "elite" bins in comparison to the less filled ones. Should $k = 1$, only the total number of bins used would matter, contrary to the remark above. The larger k is, the more we prefer the "extremists" as opposed to a collection of equally filled bins. We have experimented with several values of k and found out that $k = 2$ gives good results. Larger values of k seem to lead to premature convergence of the algorithm, as the local optima, due to a few well-filled bins, are too hard to escape. [Falkenauer and Delchambre 92, notation edited for consistency]

STAGE requires neither the complex "group-oriented" genetic operators of Falkenauer's encoding, nor any hand-tuning of the cost function. Rather, it uses natural local-search operators on the space of legal solutions. A solution state x simply assigns a bin number $b(a_i)$ to each item. Each item is initially placed alone in a bin: $b(a_1) = 1, b(a_2) = 2, \ldots, b(a_n) = n$. Neighboring states are generated by moving a single item a_i, as follows:

1. Let B be the set of bins other than $b(a_i)$ that are non-empty but still have enough spare capacity to accommodate a_i;
2. If $B = \emptyset$, then move a_i to an empty bin;
3. Otherwise, move a_i to a bin selected randomly from B.

Note that hillclimbing methods always reject moves of type (2), which add a new bin; and that if equi-cost moves are also rejected, then the only accepted moves will be those that empty a bin by placing a singleton item in an occupied bin.

The objective function STAGE is given to minimize is simply $\text{Obj}(x) = m$, the number of bins used. There is no need to tune the evaluation function manually. For automatic learning of its own secondary evaluation function, STAGE is provided with two state features, just as in the bin-packing example of section 3.3.2 (p. 56):

- Feature 1: $\text{Obj}(x)$, the number of bins used by solution x
- Feature 2: $\text{Var}(x)$, the variance in bin fullness levels

This second feature provides STAGE with information about the proportion of "extremist" bins, similar to that provided by Falkenauer's cost function. STAGE then learns its evaluation function by quadratic regression over these two features.

The remaining parameters to STAGE are set as follows: the patience parameters are set to 250 and the ObjBound cutoff is disabled (set to $-\infty$). In a few informal experiments, varying these parameters had a negligible effect on the results. Table 4.1 lists all of STAGE's parameter settings.

Parameter	**Setting**
π	stochastic hillclimbing, rejecting equi-cost moves, patience=250
ObjBound	$-\infty$
features	2 (number of bins used, variance of bin fullness levels)
fitter	quadratic regression
Pat	250
TotEvals	100,000

Table 4.1. Summary of STAGE parameters for bin-packing results. (For descriptions of the parameters, see Section 3.2.2.)

STAGE's performance is contrasted with that of four other algorithms:

- HC0: multi-restart stochastic hillclimbing with equi-cost moves rejected, patience=1000. On each restart, search begins at the initial state which has each item in its own bin.
- HC1: the same, but with equi-cost moves accepted.
- SA: simulated annealing, as described in Appendix B.

- BFR: multi-restart "best-fit-randomized," a simple bin-packing algorithm with good worst-case performance bounds [Kenyon 96, Coffman *et al.* 96]. BFR begins with all bins empty and a random permutation of the list of items. It then successively places each item into the fullest bin that can accommodate it, or a new empty bin if no non-empty bin has room. When all items have been placed, BFR outputs the number of bins used. The process then repeats with a new random permutation of the items.

All algorithms are limited to 100,000 total moves. (For BFR, each random permutation tried counts as a single move.)

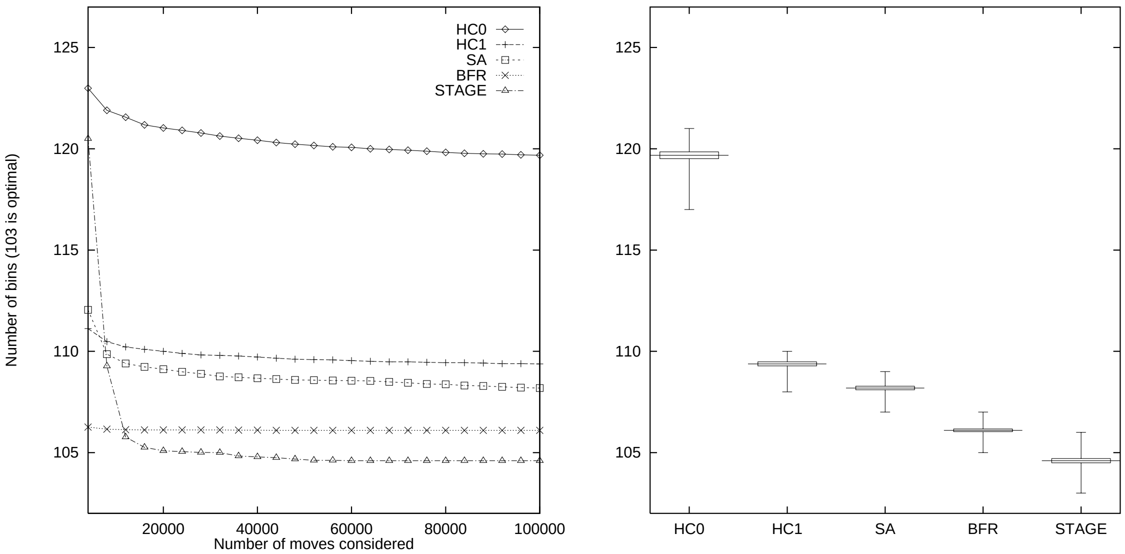


FIGURE 4.2. Bin-packing performance

The results of 100 runs of each algorithm are summarized in Table 4.2 and displayed in Figure 4.2. Stochastic hillclimbing rejecting equi-cost moves (HC0) is clearly the weakest competitor on this problem; as pointed out earlier, it gets stuck at the first solution in which each bin holds at least two items. With equi-cost moves accepted (HC1), hillclimbing explores much more effectively and performs almost as well as simulated annealing. Best-fit-randomized performs even better. However, STAGE—building itself a new evaluation function by learning to predict the behavior of HC0, the weakest algorithm—significantly outperforms all the others. Its mean solution quality is under 105 bins, and on one of the 100 runs, it equalled the best solution (103 bins) found by Falkenauer's specialized bin-packing algorithm [Falkenauer 96].

The best-so-far curves show that STAGE learns quickly, achieving good performance after only about 10000 moves, or about 4 iterations on average.

STAGE's timing overhead for learning was on the order of only 7% over HC0. In fact, the timing differences between SA, HC1, HC0, and STAGE are attributable mainly to their different ratios of accepted to rejected moves: rejected moves are slower in my bin-packing implementation since they must be undone. Since SA accepts most moves it considers early in search, it finishes slightly more quickly. The BFR runs took much longer, but the performance curve makes it clear that its best solutions were reached quickly.[2]

Instance	Algorithm	Performance (100 runs each)			≈	moves
		mean	best	worst	time	accepted
u250_13	HC0	119.68±0.17	117	121	12.4s	8%
	HC1	109.38±0.10	108	110	11.4s	71%
	SA	108.19±0.09	107	109	11.1s	44%
	BFR	106.10±0.07	105	107	95.3s	—
	STAGE	**104.60**±0.11	**103**	**106**	13.3s	6%

TABLE 4.2. Bin-packing results

As a follow-up experiment, I ran HC1, SA, BFR, and STAGE on all 20 bin-packing problem instances in the u250 class of the OR-Library (see Appendix C.1). All runs used the same settings shown in Table 4.1. The results, given in Table 4.3, show that STAGE consistently found the best packings in each case. STAGE's average improvements over HC1, SA, and BFR were 5.0±0.3 bins, 3.8±0.3 bins, and 1.6±0.3 bins, respectively.

How did STAGE succeed? The STAGE runs followed the same pattern as the runs on the small example bin-packing instance of last chapter (§3.3.2). STAGE learned a secondary evaluation function, $\tilde{V}^\pi$, that successfully traded off between the original objective and the additional bin-variance feature to identify promising start states. A typical evaluation function learned by STAGE is plotted in Figure 4.3.[3] As in

[2]Falkenauer reported a running time of 6346 seconds for his genetic algorithm to find the global optimum on this instance [Falkenauer 96], though this was measured on an SGI R4000 and our times were measured on an SGI R10000.

[3]The particular $\tilde{V}^\pi$ plotted is a snapshot from iteration #15 of a STAGE run, immediately after the solution $\mathrm{Obj}(x) = 103$ was found. The learned coefficients are

$$\tilde{V}^\pi(\mathrm{Obj}, \mathrm{Var}) = -99.1 + 636\ \mathrm{Var} + 3462\ \mathrm{Var}^2 + 2.64\ \mathrm{Obj} - 9.03\ \mathrm{Obj}\cdot\mathrm{Var} - 0.00642\ \mathrm{Obj}^2.$$

Inst.	Alg.	Performance (25 runs)			Inst.	Alg.	Performance (25 runs)		
		mean	best	worst			mean	best	worst
u250_00	HC1	105.9±0.2	105	107	u250_10	HC1	111.8±0.2	111	112
	SA	104.7±0.2	104	105		SA	110.4±0.2	110	111
	BFR	102.1±0.1	102	103		BFR	108.2±0.1	108	109
	STAGE	**100.8**±0.2	**100**	**102**		STAGE	**106.8**±0.2	**106**	**108**
u250_01	HC1	106.4±0.2	105	107	u250_11	HC1	108.2±0.2	107	109
	SA	105.0±0.2	104	106		SA	107.0±0.1	106	108
	BFR	103.0±0.1	102	**103**		BFR	104.9±0.1	104	**105**
	STAGE	**101.2**±0.2	**100**	**103**		STAGE	**103.0**±0.2	**102**	**105**
u250_02	HC1	108.8±0.2	108	109	u250_12	HC1	112.4±0.2	111	113
	SA	107.6±0.3	106	108		SA	111.2±0.2	110	112
	BFR	105.1±0.1	105	**106**		BFR	109.2±0.2	109	110
	STAGE	**103.9**±0.5	**103**	109		STAGE	**107.3**±0.2	**106**	**108**
u250_03	HC1	106.2±0.2	105	107	u250_13	HC1	109.3±0.2	109	110
	SA	105.3±0.2	105	106		SA	108.2±0.2	108	109
	BFR	103.1±0.1	103	104		BFR	106.2±0.1	106	107
	STAGE	**101.6**±0.2	**101**	**102**		STAGE	**104.5**±0.2	**104**	**105**
u250_04	HC1	107.6±0.2	106	108	u250_14	HC1	106.4±0.2	106	107
	SA	106.8±0.2	106	107		SA	105.2±0.2	105	106
	BFR	104.0±0.1	104	105		BFR	103.1±0.1	103	104
	STAGE	**102.7**±0.2	**102**	**103**		STAGE	**101.3**±0.2	**100**	**102**
u250_05	HC1	108.0± 0	108	108	u250_15	HC1	112.0±0.2	111	113
	SA	106.8±0.1	106	107		SA	111.0±0.1	110	112
	BFR	105.0±0.1	105	106		BFR	109.0±0.1	108	110
	STAGE	**103.1**±0.2	**102**	**104**		STAGE	**107.0**±0.1	**107**	**108**
u250_06	HC1	108.1±0.2	107	109	u250_16	HC1	103.8±0.2	103	104
	SA	106.8±0.2	106	107		SA	102.4±0.2	102	103
	BFR	105.0±0.1	104	106		BFR	100.0±0.1	100	101
	STAGE	**102.8**±0.2	**102**	**104**		STAGE	**98.7**±0.2	**98**	**99**
u250_07	HC1	110.4±0.2	110	111	u250_17	HC1	106.1±0.1	106	107
	SA	109.0±0.1	109	110		SA	104.9±0.2	104	106
	BFR	107.0±0.1	107	108		BFR	103.0±0.1	103	104
	STAGE	**105.1**±0.1	**105**	**106**		STAGE	**101.1**±0.2	**100**	**102**
u250_08	HC1	111.9±0.1	111	112	u250_18	HC1	107.0±0.2	106	108
	SA	111.1±0.1	111	112		SA	105.8±0.1	105	106
	BFR	109.1±0.1	109	110		BFR	103.0± 0	103	**103**
	STAGE	**107.4**±0.2	**106**	**108**		STAGE	**101.9**±0.3	**101**	104
u250_09	HC1	107.7±0.2	107	108	u250_19	HC1	108.4±0.2	108	109
	SA	106.1±0.1	106	107		SA	107.5±0.2	107	108
	BFR	104.1±0.1	104	105		BFR	105.1±0.1	105	106
	STAGE	**102.5**±0.3	**101**	**104**		STAGE	**103.8**±0.2	**103**	**105**

TABLE 4.3. Bin-packing results on 20 problem instances from the OR-Library

the example instance of last chapter (Figure 3.7), STAGE learns to direct the search toward the high-variance states from which hillclimbing is predicted to excel.

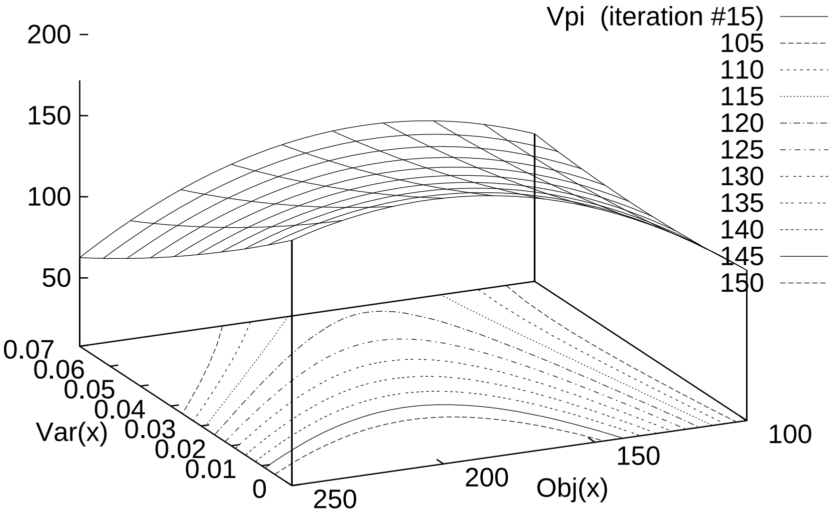


FIGURE 4.3. An evaluation function learned by STAGE on bin-packing instance u250_13. STAGE learns that the states with higher variance (wider arcs of the contour plot) are promising start states for hillclimbing.

4.3 VLSI Channel Routing

The problem of "Manhattan channel routing" is an important subtask of VLSI circuit design [Deutsch 76, Yoshimura and Kuh 82, Wong *et al.* 88, Chao and Harper 96, Wilk 96]. Given two rows of labelled terminals across a gridded rectangular channel, we must connect like-labelled pins to one another by placing wire segments into vertical and horizontal tracks (see Figure 4.4). 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-complete [Szymanski 85]. Specialized algorithms based on branch-and-bound or A* search techniques have made exact solutions attainable for some benchmarks [Lin 91]. However, larger problems still can be

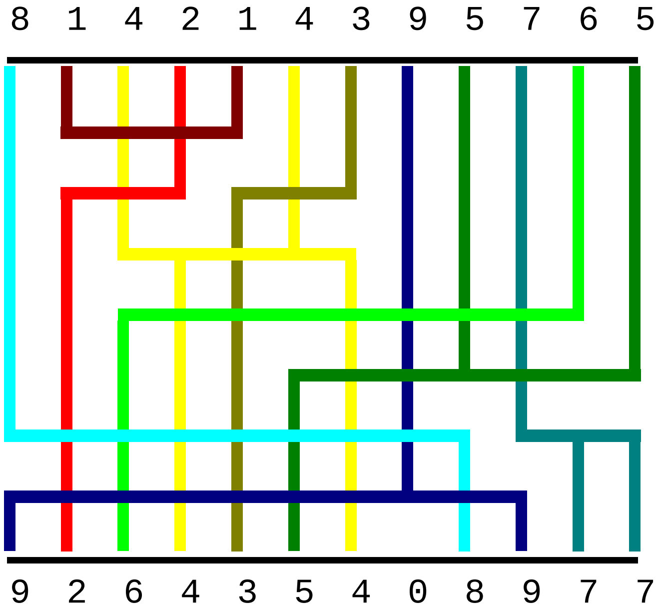


FIGURE 4.4. A small channel routing instance, shown with a solution occupying 7 horizontal tracks.

solved only approximately by heuristic techniques, e.g. [Wilk 96]. My implementation is based on the SACR system [Wong *et al.* 88, Chapter 4], a simulated annealing approach. SACR's 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.

Like Falkenauer's bin-packing implementation described above, Wong's channel routing implementation required manual objective function tuning:

> Clearly, the objective function to be minimized is the channel width $w(x)$. However, $w(x)$ is too crude a measure of the quality of intermediate solutions. Instead, for any valid partition x, the following cost function is used:
>
> $$C(x) = w(x)^2 + \lambda_p \cdot p(x)^2 + \lambda_U \cdot U(x) \tag{4.1}$$
>
> where $p(x)$ is the longest path length of G_x [a graph induced by the partitioning], both λ_p and λ_U are constants, and ... $U(x) = \sum_{i=1}^{w(x)} u_i(x)^2$, where $u_i(x)$ is the fraction of track i that is unoccupied. [Wong *et al.* 88, notation edited for consistency]

They hand-tuned the coefficients and set $\lambda_p = 0.5, \quad \lambda_U = 10$. To apply STAGE to this problem, I began with not the contrived function $C(x)$ but the natural objective function $\mathrm{Obj}(x) = w(x)$. The additional objective function terms used in Equation 4.1, $p(x)$ and $U(x)$, along with $w(x)$ itself, were given as the three input features to STAGE's function approximator. Thus, the features of a solution x are

- Feature 1: $w(x) =$ channel width, i.e. the number of horizontal tracks used by the solution.

- Feature 2: $p(x) =$ the length of the longest path in a "merged constraint graph" G_x representing the solution. This feature is a lower bound on the channel width of all solutions derived from x by merging subnets [Wong *et al.* 88]. In other words, this feature bounds the quality of solution that can be reached from x by repeated application of a restricted class of operators, namely, merging the contents of two tracks into one. The inherently predictive nature of this feature suits STAGE well.

- Feature 3: $U(x) =$ the sparseness of the horizontal tracks, measured by $\sum_{i=1}^{w(x)} u_i(x)^2$, where $u_i(x)$ is the fraction of track i that is unoccupied. Note that this feature is real-valued, whereas the other two are discrete; and that $0 \leq U(x) < w(x)$.

Table 4.4 summarizes the remaining STAGE parameter settings.

Parameter	**Setting**
π	stochastic hillclimbing, rejecting equi-cost moves, patience=250
ObjBound	$-\infty$
features	3 $(w(x), p(x), U(x))$
fitter	linear regression
Pat	250
TotEvals	500,000

Table 4.4. Summary of STAGE parameters for channel routing results

STAGE's performance is contrasted with that of four other algorithms:

- HC0: multi-restart stochastic hillclimbing with equi-cost moves rejected, patience=400. On each restart, search begins at the initial state which has each subnet on its own track.

- HC1: the same, but with equi-cost moves accepted.

- SAW: simulated annealing, using the hand-tuned objective function of Equation 4.1 [Wong *et al.* 88].
- SA: simulated annealing, using the true objective function $\mathrm{Obj}(x) = w(x)$.
- HCI: stochastic hillclimbing with equi-cost moves accepted, patience=∞ (i.e., no restarting).

All algorithms were limited to 500,000 total moves. Results on YK4, an instance with 140 vertical tracks, are given in Figure 4.5 and Table 4.5. By construction (see Appendix C.2 for details), the optimal routing x^* for this instance occupies only 10 horizontal tracks, i.e. $\mathrm{Obj}(x^*) = 10$. A 12-track solution is depicted in Figure 4.6.

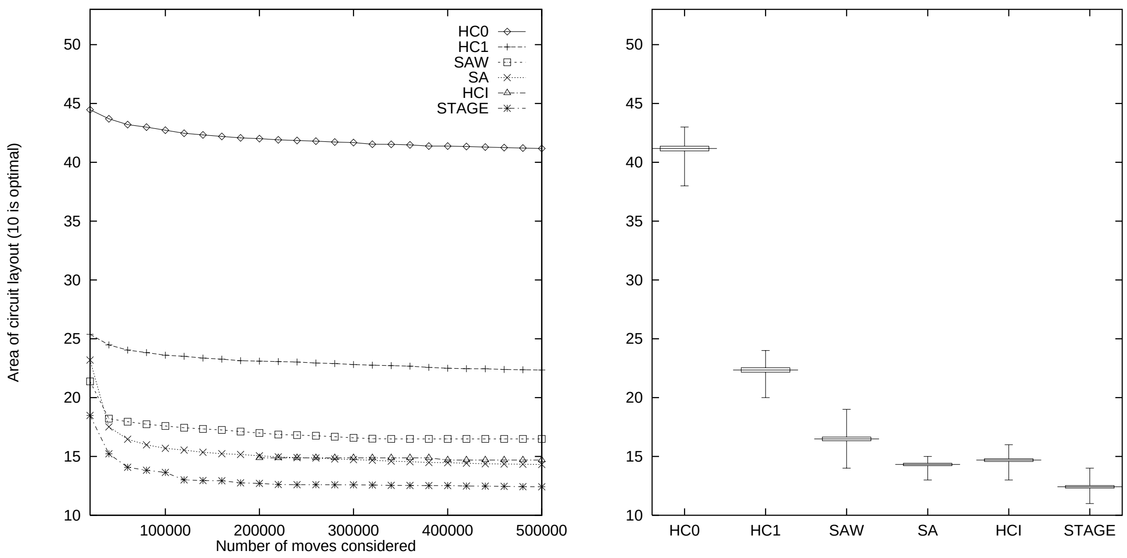


FIGURE 4.5. Channel routing performance on instance YK4

None of the local search algorithms successfully finds an optimal 10-track solution. Experiments HC0 and HC1 show that multi-restart hillclimbing performs terribly when equi-cost moves are rejected, but significantly better when equi-cost moves are accepted. Experiment SAW shows that simulated annealing, as used with the objective function of [Wong *et al.* 88], does considerably better. Surprisingly, the annealer of Experiment SA does better still. It seems that the "crude" evaluation function $\mathrm{Obj}(x) = w(x)$ allows a long simulated annealing run to effectively random-walk along the ridge of all solutions of equal cost, and given enough time it will fortuitously find a hole in the ridge. In fact, increasing hillclimbing's patience to ∞ (disabling restarts) worked nearly as well.

Instance	Algorithm	Performance (100 runs each)			≈	moves
		mean	best	worst	time	accepted
YK4	HC0	41.17±0.20	38	43	212s	8%
	HC1	22.35±0.19	20	24	200s	80%
	SAW	16.49±0.16	14	19	245s	32%
	SA	14.32±0.10	13	15	292s	57%
	HCI	14.69±0.12	13	16	350s	58%
	STAGE	12.42±0.11	11	14	405s	5%

TABLE 4.5. Channel routing results

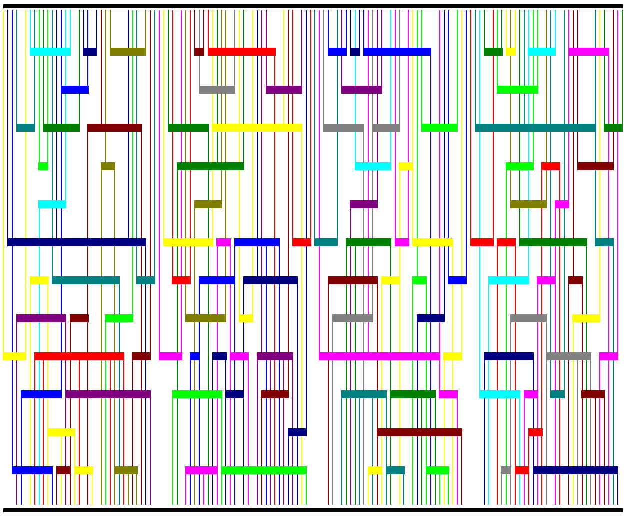


FIGURE 4.6. A 12-track solution found by STAGE on instance YK4

STAGE performs significantly better than all of these. How does STAGE learn to combine the features $w(x)$, $p(x)$, and $U(x)$ into a new evaluation function that outperforms simulated annealing? I have investigated this question extensively; the analysis is reported in Chapter 5. Later, in Section 6.2, I also report the results of transferring STAGE's learned evaluation function between different channel routing instances.

The disparity in the running times of the algorithm deserves explanation. The STAGE runs took about twice as long to complete as the hillclimbing runs, but this is *not* due to the overhead for STAGE's learning: linear regression over three simple features is extremely cheap. Rather, STAGE is slower because when the search reaches good solutions, the process of generating a legal move candidate becomes more expensive; STAGE is victimized by its own success. STAGE and HCI are also slowed relative to SA because they reject many more moves, forcing extra "undo" operations. In any event, the performance curve of Figure 4.5 indicates that halving any algorithm's running time would not affect its relative performance ranking.

4.4 Bayes Network Learning

Given a dataset, an important data mining task is to identify the Bayesian network structure that best models the probability distribution of the data [Mitchell 97, Heckerman *et al.* 94, Friedman and Yakhini 96]. The problem amounts to finding the best-scoring acyclic graph structure on A nodes, where A is the number of attributes in each data record.

Several scoring metrics are common in the literature, including metrics based on Bayesian analysis [Chickering *et al.* 94] and metrics based on Minimum Description Length (MDL) [Lam and Bacchus 94, Friedman 97]. I use the MDL metric, which trades off between maximizing fit accuracy and minimizing model complexity. The objective function decomposes into a sum over the nodes of the network x:

$$\mathrm{Obj}(x) = \sum_{j=1}^{A} \big(-\mathrm{Fitness}(x_j) + K \cdot \mathrm{Complexity}(x_j) \big) \tag{4.2}$$

Following Friedman [96], the Fitness term computes a mutual information score at each node x_j by summing over all possible joint assignments to variable j and its parents:

$$\mathrm{Fitness}(x_j) = \sum_{v_j} \sum_{V_{\mathrm{Par}_j}} N(v_j \wedge V_{\mathrm{Par}_j}) \log \frac{N(v_j \wedge V_{\mathrm{Par}_j})}{N(V_{\mathrm{Par}_j})}$$

Here, $N(\cdot)$ refers to the number of records in the database that match the specified variable assignment. I use the *AD*tree data structure to make calculating $N(\cdot)$ efficient [Moore and Lee 98].

The Complexity term simply counts the number of parameters required to store the conditional probability table at node j:

$$\text{Complexity}(x_j) = \big(\text{Arity}(j) - 1\big) \prod_{i \in \text{Par}_j} \text{Arity}(i)$$

The constant K in Equation 4.2 is set to $\log(R)/2$, where R is the number of records in the database [Friedman 97].

No efficient methods are known for finding the acyclic graph structure x which minimizes $\text{Obj}(x)$; indeed, for Bayesian scoring metrics, the problem has been shown to be NP-hard [Chickering *et al.* 94], and a similar reduction probably applies for the MDL metric as well. Thus, multi-restart hillclimbing and simulated annealing are commonly applied [Heckerman *et al.* 94, Friedman 97]. My search implementation works as follows. To ensure that the graph is acyclic, a permutation $x_{i_1}, x_{i_2}, \ldots, x_{i_A}$ on the A nodes is maintained, and all links in the graph are directed from nodes of lower index to nodes of higher index. Local search begins from a linkless graph on the identity permutation. The following move operators then apply:

- With probability 0.7, choose two random nodes of the network and add a link between them (if that link isn't already there) or delete the link between them (otherwise).

- With probability 0.3, swap the permutation ordering of two random nodes of the network. Note that this may cause multiple graph edges to be reversed.

Obj can be updated incrementally after a move by recomputing Fitness and Complexity at only the affected nodes.

For learning, STAGE was given the following seven extra features:

- Features 1–2: mean and standard deviation of Fitness over all the nodes

- Features 3–4: mean and standard deviation of Complexity over all the nodes

- Features 5–6: mean and standard deviation of the number of parents of each node

- Feature 7: the number of "orphan" nodes

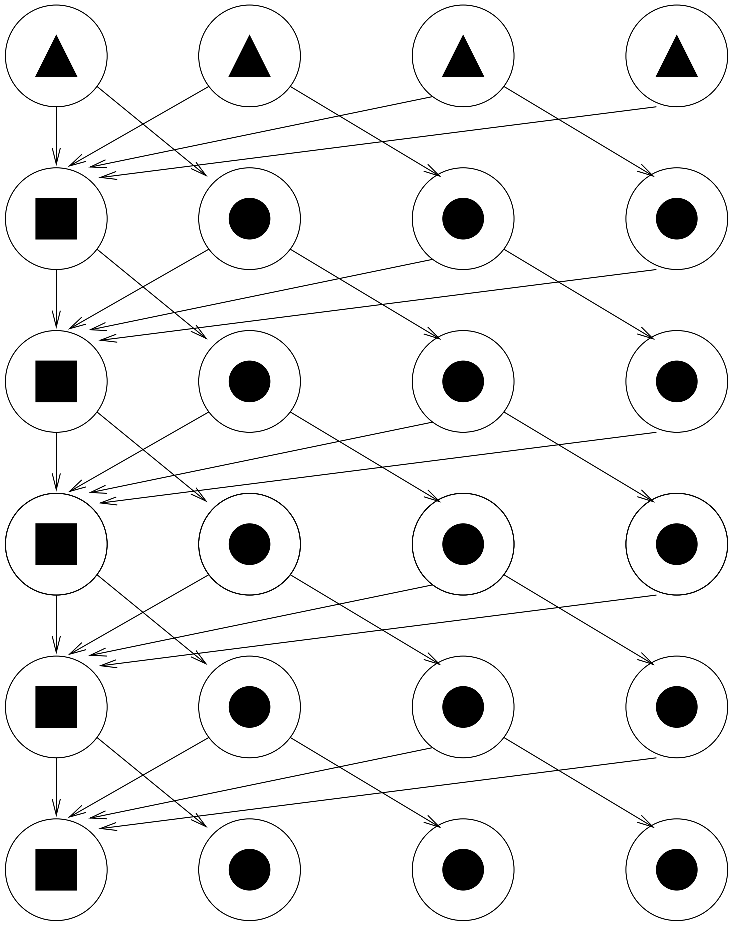


FIGURE 4.7. The SYNTH125K dataset was generated by this Bayes net (from [Moore and Lee 98]). All 24 attributes are binary. There are three kinds of nodes. The nodes marked with triangles are generated with $P(a_i = 0) = 0.8$, $P(a_i = 1) = 0.2$. The square nodes are deterministic. A square node takes value 1 if the sum of its four parents is even, else it takes value 0. The circle nodes are probabilistic functions of their single parent, defined by $P(a_i = 1 \mid Parent = 0) = 0$ and $P(a_i = 1 \mid Parent = 1) = 0.4$. This provides a dataset with fairly sparse values and with many interdependencies.

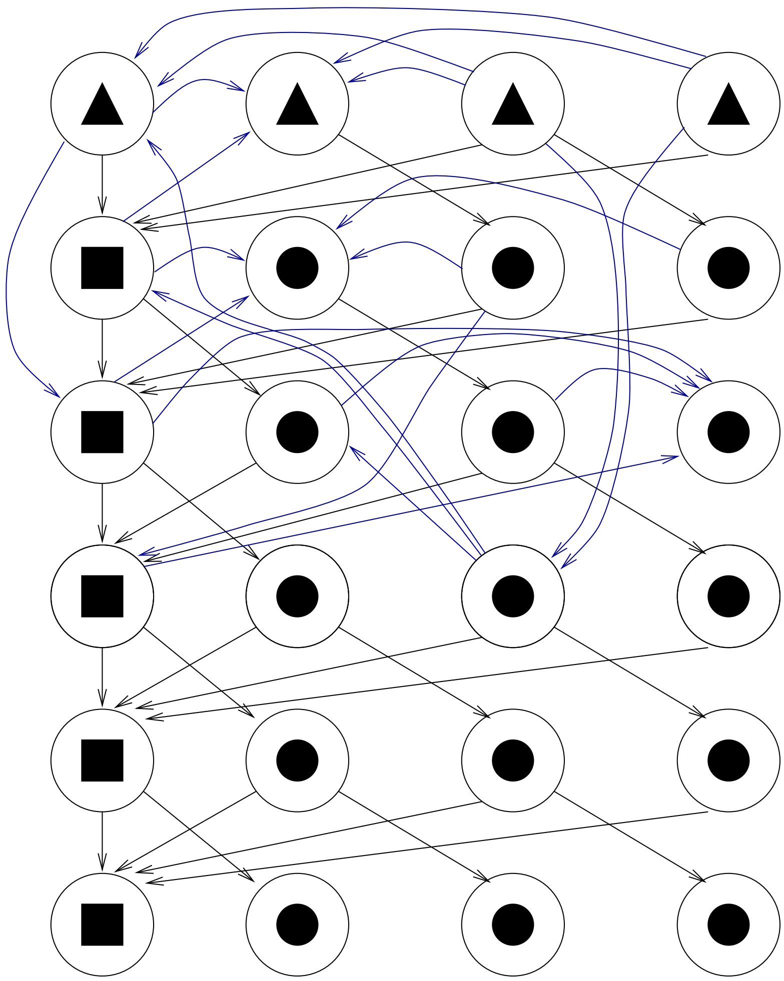


FIGURE 4.8. A network structure learned by a sample run of STAGE from the SYNTH125K dataset. Its Obj score is 719074. By comparison, the actual network that was used to generate the data (shown earlier in Figure 4.7) scores 718641. Only two edges from the generator net are missing from the learned net. The learned net includes 17 edges not in the generator net (shown as curved arcs).

I applied STAGE to three datasets: MPG, a small dataset consisting of 392 records of 10 attributes each; ADULT2, a large real-world dataset consisting of 30,162 records of 15 attributes each; and SYNTH125K, a synthetic dataset consisting of 125,000 records of 24 attributes each. The synthetic dataset was generated by sampling from the Bayes net depicted in Figure 4.7. A perfect reconstruction of that net would receive a score of $\mathrm{Obj}(x) = 718641$. For further details of the other datasets, please see Appendix C.3.

The STAGE parameters shown in Table 4.6 were used in all domains. Figures 4.9–4.11 and Table 4.7 contrast the performance of hillclimbing (HC), simulated annealing (SA) and STAGE. For reference, the table also gives the score of the "linkless" Bayes net—corresponding to the simplest model of the data, that all attributes are generated independently.

Parameter	**Setting**
π	stochastic hillclimbing, patience=200
ObjBound	0
features	7 (Fitness μ, σ; Complexity μ, σ; #Parents μ, σ; #Orphans)
fitter	quadratic regression
Pat	200
TotEvals	100,000

Table 4.6. Summary of STAGE parameters for Bayes net results

Instance	Algorithm	Performance (100 runs each) mean	best	worst	≈ time	moves accepted
MPG	HC	**3563.4**± 0.3	**3561.3**	**3567.4**	35s	5%
(linkless score	SA	3568.2± 0.9	**3561.3**	3595.5	47s	30%
= 5339.4)	STAGE	**3564.1**± 0.4	**3561.3**	3569.5	48s	2%
ADULT2	HC	440567± 52	439912	441171	239s	6%
(linkless score	SA	440924± 134	**439551**	444094	446s	28%
= 554090)	STAGE	**440432**± 57	439773	**441052**	351s	6%
SYNTH125K	HC	748201±1714	725364	766325	151s	10%
(linkless score	SA	**726882**±1405	718904	**754002**	142s	29%
= 1,594,498)	STAGE	730399±1852	**718804**	782531	156s	4%

Table 4.7. Bayes net structure-finding results

On SYNTH125K, the largest dataset, simulated annealing and STAGE both improve significantly over multi-restart hillclimbing, usually attaining a score within 2%

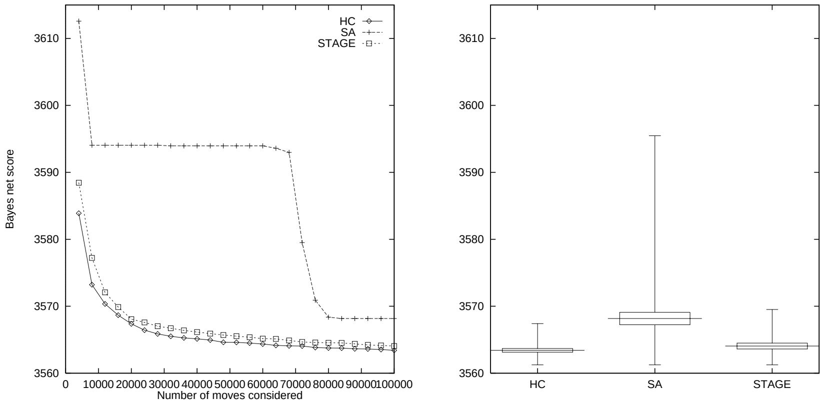


FIGURE 4.9. Bayes net performance on instance MPG

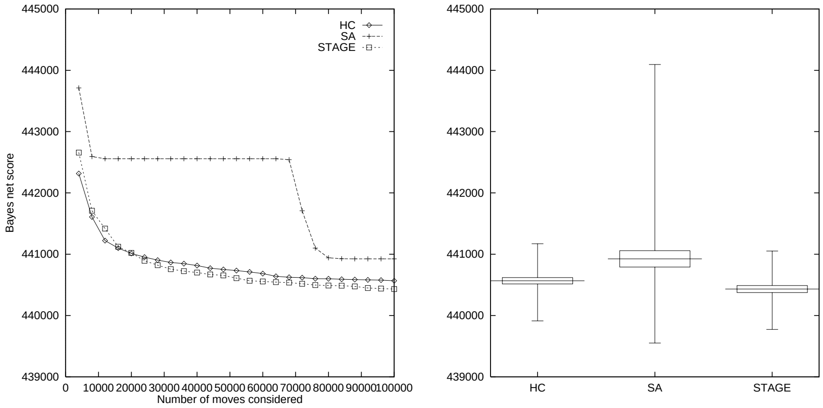


FIGURE 4.10. Bayes net performance on instance ADULT2

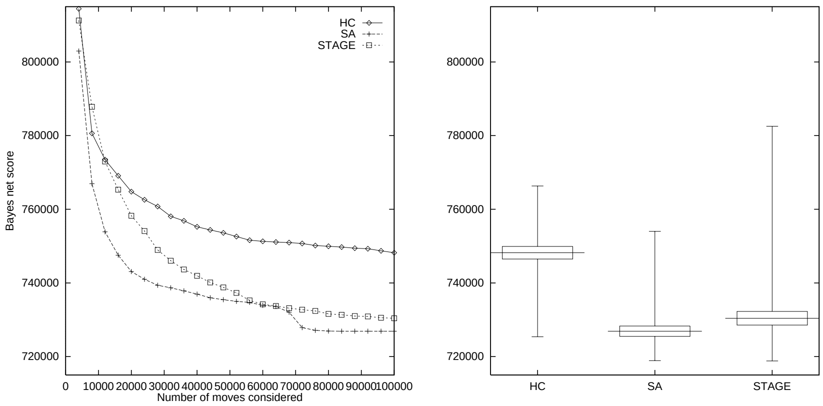


FIGURE 4.11. Bayes net performance on instance SYNTH125K

of that of the Bayes net that generated the data, and on some runs coming within 0.04%. A good solution found by STAGE is drawn in Figure 4.8. Simulated annealing slightly outperforms STAGE on average (however, Section 6.1.5 describes an extension which improves STAGE's performance on SYNTH125K). On the MPG and ADULT2 datasets, HC and STAGE performed comparably, while SA did slightly less well on average. SA's odd-looking performance curves deserve further explanation. It turns out that they are a side effect of the large scale of the objective function: when single moves incur large changes in Obj, the adaptive annealing schedule (refer to Appendix B) takes longer to raise the temperature to a suitably high initial level, which means that the initial part of SA's trajectory is effectively performing hillclimbing. During this phase SA can find quite a good solution, especially in the real datasets (MPG and ADULT2), for which the initial state (the linkless graph) is a good starting point for hillclimbing. The good early solutions, then, are never bettered until the temperature decreases late in the schedule; hence the best-so-far curve is flat for most of each run.

All algorithms require comparable amounts of total run time, except on the ADULT2 task where SA and STAGE both run slower than HC. On that task, the difference in run times appears to be caused by the types of graph structures explored during search; SA and STAGE spend greater effort exploring more complex networks with more connections, at which the objective function evaluates more slowly. STAGE's computational overhead for learning is insignificant.

In sum, STAGE's performance on the Bayes net learning task was less dominant than on the bin-packing and channel routing tasks, but it was still more consistently best or nearly best than either HC or SA on the three benchmark instances attempted.

4.5 Radiotherapy Treatment Planning

Radiation therapy is a method of treating tumors [Censor *et al.* 88]. As illustrated in Figure 4.12, a linear accelerator that 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 reasonably 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 task, then, is the following "inverse problem": given the map and the forward model, produce a treat-

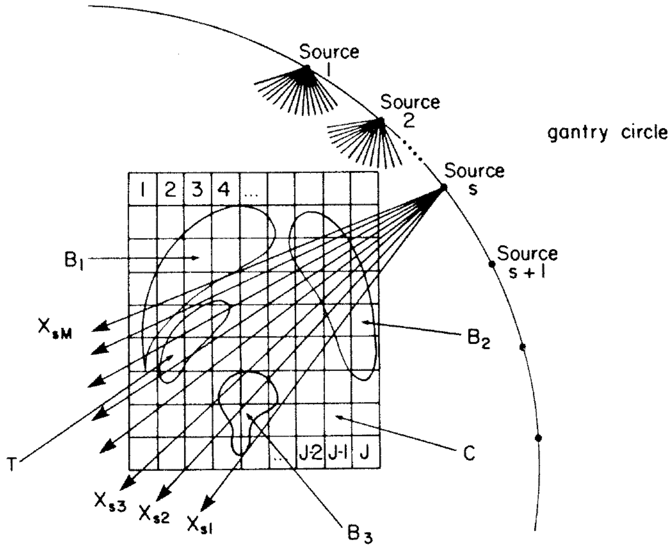


FIGURE 4.12. Radiotherapy treatment planning (from [Censor *et al.* 88])

ment plan that meets target radiation doses for the tumor while minimizing damage to sensitive nearby structures. In current practice, simulated annealing and/or linear programming are often used for this problem [Webb 91, Webb 94].

Figure 4.13 illustrates a simplified planar instance of the radiotherapy problem. The instance consists of an irregularly shaped tumor and four sensitive structures: the eyes, the brainstem, and the rest of the head. A treatment for this instance consists of a plan to turn the accelerator beam either on or off at each of 100 beam angles evenly spaced within $[-\frac{3\pi}{4}, \frac{3\pi}{4}]$. Given a treatment plan, the objective function is calculated by summing ten terms: an overdose penalty and an underdose penalty for each of the five structures. For details of the penalty terms, please refer to Appendix C.4.

I applied hillclimbing (HC), simulated annealing (SA), and STAGE to this domain. Objective function evaluations are computationally expensive here, so my experiments considered only 10,000 moves per run. The features provided to STAGE consisted of the ten subcomponents of the objective function. STAGE's parameter settings are given in Table 4.8.

Results of 200 runs of each algorithm are shown in Figure 4.14 and Table 4.9. All performed comparably, but STAGE's solutions were best on average. Note, however, that the very best solution over all 600 runs was found by a hillclimbing run. The objective function computation dominates the running time; STAGE's overhead for learning is relatively insignificant.

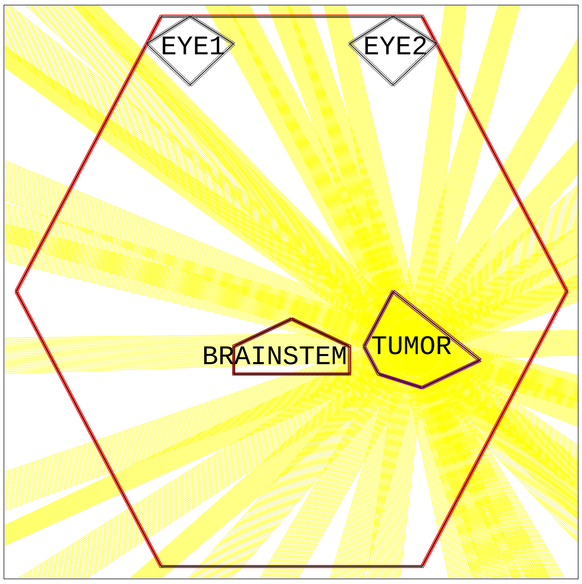


FIGURE 4.13. Radiotherapy instance 5E

Parameter	Setting
π	stochastic hillclimbing, patience=200
ObjBound	0
features	10 (overdose penalty and underdose penalty for each organ)
fitter	quadratic regression
Pat	200
TotEvals	10,000

TABLE 4.8. Summary of STAGE parameters for radiotherapy results

Instance	Algorithm	Performance (200 runs each)			$\approx$	moves
		mean	best	worst	time	accepted
5E	HC	18.822±0.030	**18.003**	19.294	550s	5.5%
	SA	18.817±0.043	18.376	19.395	460s	29%
	STAGE	**18.721**±0.029	18.294	**19.155**	530s	4.9%

TABLE 4.9. Radiotherapy results

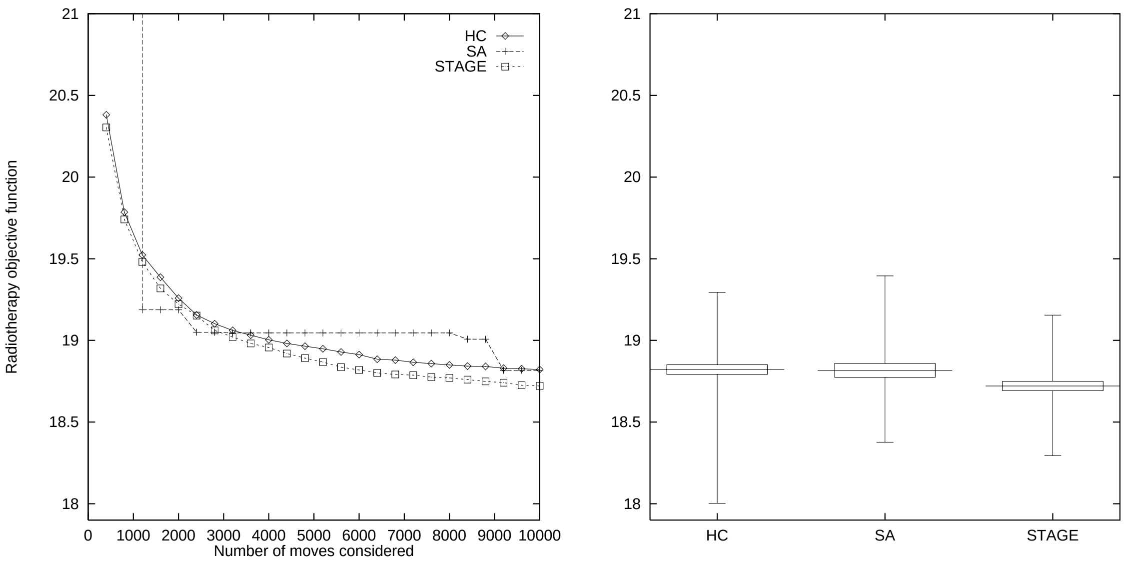


FIGURE 4.14. Radiotherapy performance on instance 5E

4.6 Cartogram Design

A "cartogram" or "Density Equalizing Map Projection" is a geographic map whose subarea boundaries have been deformed so that population density is uniform over the entire map [Dorling 94, Gusein-Zade and Tikunov 93, Dorling 96]. Such maps can be useful for visualization of, say, geographic disease distributions, because they remove the confounding effect of population density. I considered the particular instance of redrawing the map of the continental United States such that each state's area is proportional to its electoral vote for U.S. President. The goal of optimization is to best meet the new area targets for each state while minimally distorting the states' shapes and borders.

I represented the map as a collection of 162 points in $\Re^2$; each state was defined as a polygon over a subset of those points. Search begins at the original, undistorted U.S. map. The search operator consisted of perturbing a random point slightly; perturbations that would cause two edges to cross were disallowed. The objective function was defined as

$$\mathrm{Obj}(x) = \Delta\mathrm{Area}(x) + \Delta\mathrm{Gape}(x) + \Delta\mathrm{Orient}(x) + \Delta\mathrm{Segfrac}(x)$$

where $\Delta\mathrm{Area}(x)$ penalizes states for missing their new area targets, and the other three terms penalize states for differing in shape and orientation from the true U.S. map. For details of these penalty terms, please refer to Appendix C.5. Each of the

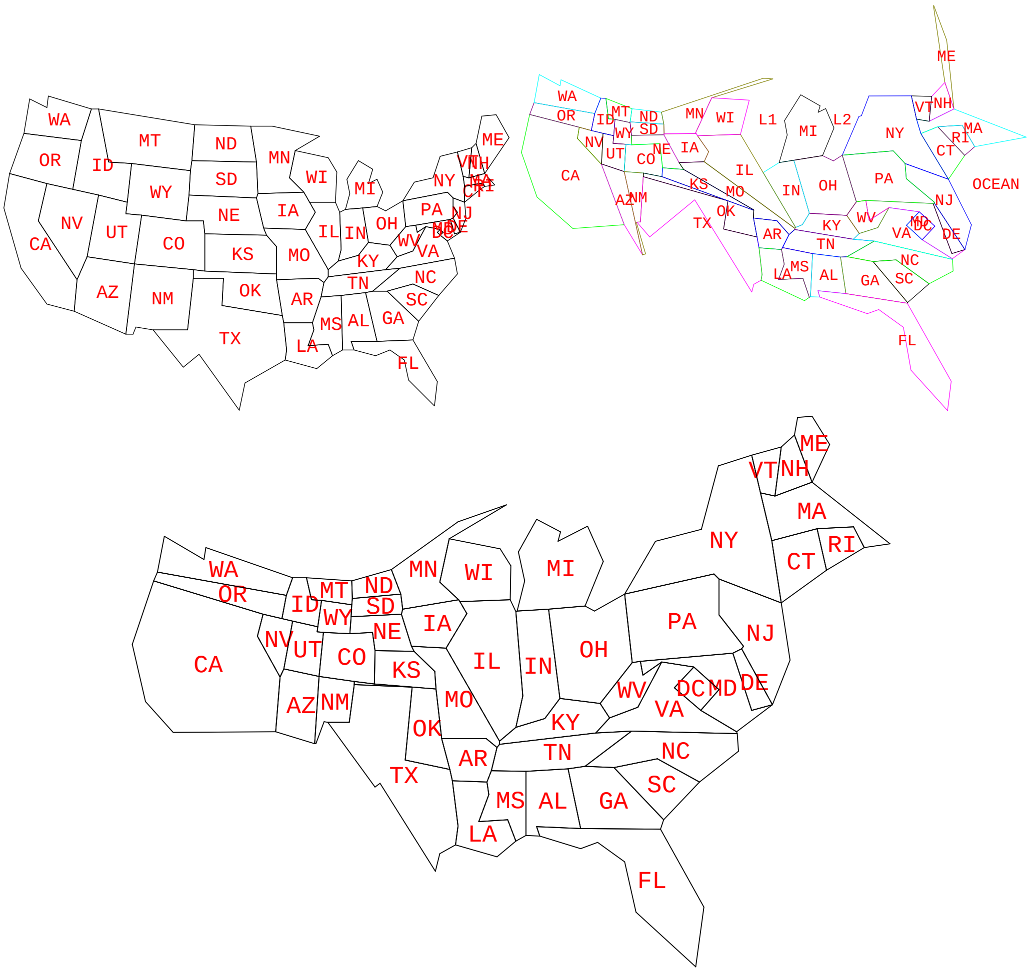


FIGURE 4.15. Cartograms of the continental U.S. Each state's target area is proportional to its electoral vote for U.S. President. The undistorted U.S. map (top left) has zero penalty for state shapes and orientations but a large penalty for state areas, so $\mathrm{Obj}(x) = 525.7$. Hillclimbing produces solutions like the one shown at top right, for which $\mathrm{Obj}(x) = 0.115$. The third cartogram, found by STAGE, has $\mathrm{Obj}(x) = 0.043$.

feature values can be updated incrementally when a single vertex is moved, so local search can be applied to optimize Obj(x) quite efficiently.

For STAGE, I represented each configuration by four features—namely, the four subcomponents of Obj. Learning a new evaluation function with quadratic regression over these features, STAGE produced a significant improvement over hillclimbing, but was outperformed by simulated annealing. Table 4.10 shows STAGE's parameters, and Table 4.11 and Figure 4.16 show the results. Later, in Sections 5.2.1–5.2.4, I report the results of further experiments on the cartogram domain using varying feature sets and function approximators.

Parameter	**Setting**
π	stochastic hillclimbing, patience=200
ObjBound	0
features	4 (subcomponents of Obj)
fitter	quadratic regression
Pat	200
TotEvals	1,000,000

Table 4.10. Summary of STAGE parameters for cartogram results

Instance	Algorithm	Performance (100 runs each)			$\approx$	moves
		mean	best	worst	time	accepted
US49	HC	0.174±0.002	0.152	0.195	190s	14%
	SA	**0.037**±0.003	**0.031**	0.170	130s	32%
	STAGE	0.056±0.003	0.038	**0.132**	172s	7%

Table 4.11. Cartogram results

4.7 Boolean Satisfiability

4.7.1 WALKSAT

Finding a variable assignment that satisfies a large Boolean expression is a fundamental—indeed, the original—NP-complete problem [Garey and Johnson 79]. In recent years, surprisingly difficult formulas have been solved by WALKSAT [Selman *et al.* 96], a simple local search method. WALKSAT, given a formula expressed in CNF (a conjunction of disjunctive clauses), conducts a random walk in assignment space that is

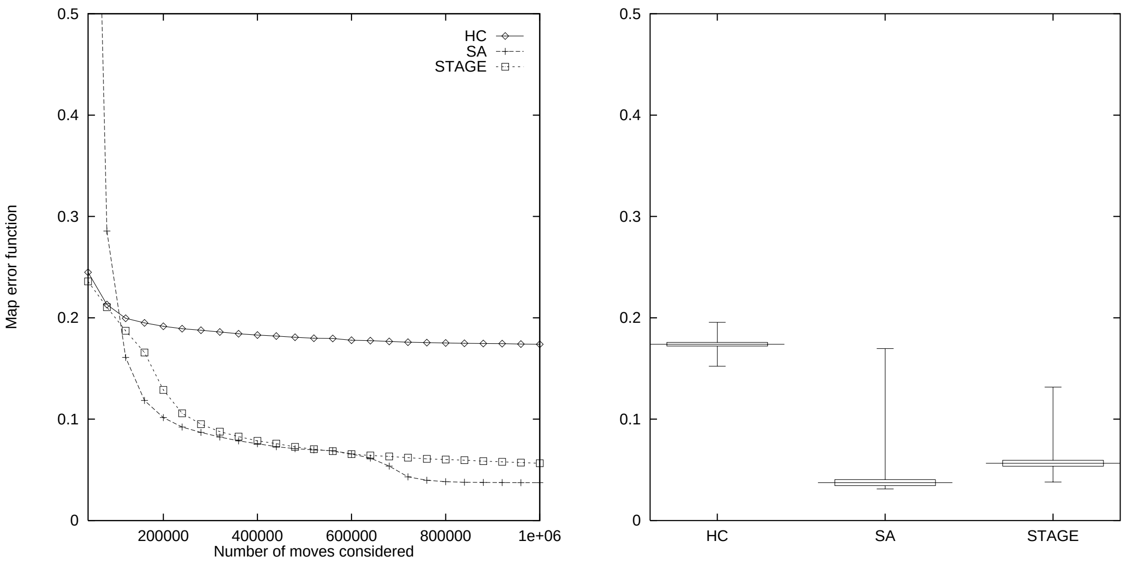


FIGURE 4.16. Cartogram performance on instance US49

biased toward minimizing

$$\text{Obj}(x) = \#\text{ of clauses unsatisfied by assignment } x.$$

When $\text{Obj}(x) = 0$, all clauses are satisfied and the formula is solved.

WALKSAT searches as follows. On each step, it first selects an unsatisfied clause at random; it will satisfy that clause by flipping one variable within it. To decide which one, it first evaluates how much overall improvement to Obj would result from flipping each variable. If the best such improvement is positive, it greedily flips a variable that attains that improvement. Otherwise, it flips a variable which worsens Obj: with probability (1-*noise*), a variable which harms Obj the least, and with probability *noise*, a variable at random from the clause. The best setting of *noise* is problem-dependent [McAllester *et al.* 97].

WALKSAT is so effective that it has rendered nearly obsolete an archive of several hundred benchmark problems collected for a 1993 DIMACS Challenge on satisfiability [Selman *et al.* 96]. Within that archive, only the "32-bit parity function learning" instances (nefariously constructed by Crawford, Kearns, Schapire, and Hirsh [Crawford 93]) are known to be solvable in principle, yet not solvable by WALKSAT. The development of an algorithm to solve these 32-bit parity instances has been listed as one of ten outstanding challenges for research in propositional reasoning and search [Selman *et al.* 97]. Most of my experiments in the satisfiability domain have focused on the first such instance in the archive, `par32-1.cnf`, a formula consisting of

10277 clauses on 3176 variables. After presenting extensive results on this instance, I will report additional results on the four other benchmark instances in the par32 family.

4.7.2 Experimental Setup

WALKSAT is generally run in a random multi-restart regime: after every *cutoff* search steps, where *cutoff* is another parameter of the algorithm, it resets the search to a random new assignment. Can STAGE, by observing WALKSAT trajectories, learn a smarter restart policy? Certainly, a variety of additional state features potentially useful for STAGE's learning are readily available in this domain. I used the following set of five simple features:

- proportion of clauses currently unsatisfied ($\propto \mathrm{Obj}(x)$)
- proportion of clauses satisfied by exactly 1 variable
- proportion of clauses satisfied by exactly 2 variables
- proportion of variables that would break a clause if flipped
- proportion of variables set to their "naive" setting[4]

All of these features can be computed incrementally in O(1) time after any variable is flipped.

For comparison, I evaluated the performance of the following algorithms, allowing each a total of 10^8 bit flips per run:

- (HC): random multi-start hillclimbing, *patience* $= 10^4$, accepting equi-cost moves. Candidate moves are generated using the same bit-flip distribution as WALKSAT, but moves that increase the number of unsatisfied clauses are rejected.
- (S/HC): STAGE applied to π = HC. Quadratic regression is used to predict hillclimbing outcomes from the five features above. For STAGE's second phase (stochastic hillclimbing on V^π), the patience is set to 1000, and candidate bits to flip are chosen uniformly at random, not with the WALKSAT distribution.

[4]Given a CNF formula F, the naive setting of variable x_i is defined to be 0 if $\neg x_i$ appears in more clauses of F than x_i, or 1 if x_i appears in more clauses than $\neg x_i$.

- (W): WALKSAT, *noise* $= 0$, *cutoff* $= 10^6$. These parameter settings were hand-tuned for best performance over the range *noise* $\in \{0, 0.05, 0.1, 0.15, \ldots, 0.5\}$, *cutoff* $\in \{10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$. Note that the chosen cutoff level of 10^6 means that WALKSAT performs exactly 100 random restarts per run.

- (S/W): STAGE applied to π = WALKSAT. This combination raises some technical issues involving WALKSAT's termination criterion, as I explain below.

Theoretically, as discussed in Section 3.4.1, STAGE can learn from any procedure π that is proper (guaranteed to terminate) and Markovian. WALKSAT's normal termination mechanism, cutting off after a pre-specified number of steps, is not Markovian: it depends on an extraneous counter variable, not just the current assignment. To apply STAGE to π = WALKSAT, I tried three modified termination criteria:

- (S/W1): use STAGE's normal patience-based mechanism for cutting off each WALKSAT trajectory, just as I do for hillclimbing. Since WALKSAT is non-monotonic, this mechanism also violates the Markov property: the probability of cutting off at state x depends not just on x but also on an extraneous counter variable and the best Obj value seen previously. However, this is easily corrected by the following adjustment.

- (S/W2): use patience-based cutoffs, but train the function approximator on only the best-so-far states of each sample WALKSAT trajectory. By Proposition 2 (presented on page 64, proven in Appendix A.1), this subsequence of states constitutes a sample trajectory from a higher-level search procedure which is proper, strictly monotonic, and Markovian, so V^π is well-defined.

- (S/W3): cut off WALKSAT's trajectory with a fixed probability $\epsilon > 0$ after every flip. This approach results in a proper Markovian trajectory, so V^π is again well-defined. A possible drawback to this approach is that termination may randomly occur during a fruitful part of the search trajectory.

For (S/W1), I hand-tuned WALKSAT's *noise* and *patience* parameters over the same ranges I used for tuning Experiment (W). Here, I found best performance at *noise* $=$ 0.25 (more random actions) and *patience* $= 10^4$ (more frequent restarting). Without further tuning, I set *patience* $= 10^4$ in (S/W2), $\epsilon = 10^{-4}$ in (S/W3), and *noise* $= 0.25$ in both. The parameters for STAGE's second phase, hillclimbing on V^π, were set as in Experiment (S/HC) above.

Serendipitously, I discovered that introducing an additional WALKSAT parameter could improve STAGE's performance. The new parameter, call it δ_w, has the

Parameter	Setting
π	(S/HC): stochastic hillclimbing, patience=10^4; OR (S/W): WALKSAT, noise=0.25, patience=10^4, $\delta_w = 10$
ObjBound	0
features	5 (Obj, % clauses with 1 true, % clauses with 2 true, % clause-breaking variables, % naive variables)
fitter	quadratic regression
Pat	1000
TotEvals	100,000,000

Table 4.12. Summary of STAGE parameters for satisfiability results

following effect: any flip that would worsen Obj by more than δ_w is rejected. Normal WALKSAT has $\delta_w = \infty$. Hillclimbing, as done in Experiment (HC) above, is equivalent to WALKSAT with $\delta_w = 0$, which performs badly. However, using intermediate settings of δ_w—thereby prohibiting only the most destructive of WALKSAT's moves—seems not to harm WALKSAT's performance, and in some cases improves it. For the (S/W) runs reported here, I set $\delta_w = 10$. STAGE's parameter settings for both Experiments (S/HC) and (S/W) are summarized in Table 4.12.

4.7.3 Main Results

The main results are shown in Figure 4.17 and in the top six lines of Table 4.13. Experiment (HC) performs quite poorly, leaving about 50 clauses unsatisfied on each run; STAGE's learning improves on this significantly (S/HC), to about 20 unsatisfied clauses. WALKSAT does better still, leaving 15 unsatisfied clauses on average. However, all the STAGE/WALKSAT runs do significantly better, leaving only about 5 clauses unsatisfied on average, and as few as 1 or 2 of the formula's 10277 clauses unsatisfied on the best runs. Although STAGE did not manage to find an assignment with 0 clauses unsatisfied, these are currently the best published results for this benchmark [Kautz 98].

I also repeated the STAGE/WALKSAT experiments using linear regression, rather than quadratic regression, as the function approximator for V^π. With only 6 coefficients being fit instead of 21, STAGE still produced about the same level of improvement over plain WALKSAT (see Table 4.13). Indeed, the fixed-probability termination criterion experiment (S/W3) performed significantly better under this simpler regression model.

Table 4.14 shows the approximate running time required by each algorithm on

Instance	Algorithm	Performance (N runs each) mean	best	worst
par32-1.cnf ($N = 100$)	HC	48.03±0.59	40	54
	S/HC (STAGE, π = hillclimbing)	21.64±0.77	11	31
	W (WALKSAT)	15.22±0.35	9	19
	S/W1 (STAGE, π = WALKSAT)	5.36±0.33	**1**	9
	S/W2	5.04±0.27	2	**8**
	S/W3	5.60±0.29	2	9
	S/W1+linear	5.27±0.37	2	14
	S/W2+linear	6.21±0.30	2	10
	S/W3+linear	**4.43**±0.28	2	**8**
par32-2.cnf ($N = 25$)	W	15.40±0.57	13	18
	S/W3+linear	**4.32**±0.48	**2**	**7**
par32-3.cnf ($N = 25$)	W	15.84±0.45	14	18
	S/W3+linear	**4.32**±0.53	**2**	**7**
par32-4.cnf ($N = 25$)	W	15.28±0.46	13	17
	S/W3+linear	**4.63**±0.60	**2**	**7**
par32-5.cnf ($N = 25$)	W	15.48±0.61	11	18
	S/W3+linear	**4.76**±0.63	**2**	9

TABLE 4.13. Satisfiability results on the 32-bit parity benchmarks

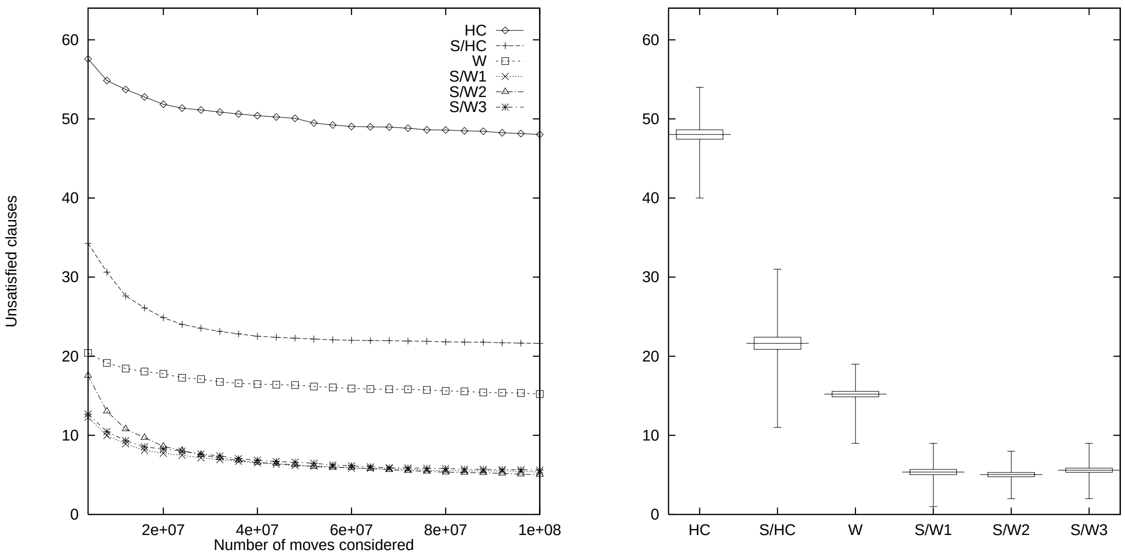


FIGURE 4.17. Satisfiability: main results on instance par32-1.cnf

instance par32-1.cnf. In this domain, STAGE takes about twice as long to complete as WALKSAT; for that matter, so does hillclimbing (HC). By profiling the executions, I identified three sources of the disparity. First, WALKSAT saves time by accepting all of the 10^8 proposed bit flips. Every time HC or STAGE rejects a move, it must unflip the modified bit and re-update the counts of violated clauses, so rejecting a move takes twice as long as accepting a move. Second, for all the STAGE runs other than S/W2, significant time is spent in memory management for storing WALKSAT trajectories. These trajectories often consist of tens of thousands of states, and my implementation is not optimized to handle such long trajectories efficiently. Since the S/W2 runs train on only the best-so-far states, they have much less data to store and do not pay this penalty. Finally, the function approximation (computing the coefficients of $\tilde{V}^\pi$ by least-squares regression) adds about 3% additional overhead to STAGE's running time.

Table 4.14 also reveals that the S/W3+linear run accepts fully 97% of its moves. This simply indicates that it is spending the bulk of its effort in the WALKSAT stage of search (during which 100% of moves are accepted), and relatively little time on the stage of hillclimbing on $\tilde{V}^\pi$. A closer look shows that this happens because the linear approximation is quite inaccurate, with an RMS error of 9.5 on its training set, and STAGE's ObjBound parameter cuts off hillclimbing as soon as the predicted $\tilde{V}^\pi(x)$ falls below zero—typically after only a few hundred moves on each iteration. Though these runs reject fewer moves than their S/W3+quadratic counterparts, their extra WALKSAT runs mean extra memory management for trajectory storage, so their running time is not significantly lessened.

Despite STAGE's overhead in this domain, it is clear from the plot of Figure 4.17 that even if running times were equalized by halving STAGE's allotted number of moves, STAGE's performance would still significantly exceed pure WALKSAT's.

4.7.4 Follow-up Experiments

I conducted three additional follow-up experiments. First, I compared pure WALKSAT with STAGE/WALKSAT on the other four 32-bit parity instances from the DIMACS archive. The results, shown in Table 4.13, corroborate the results on par32-1: STAGE consistently leaves 2/3 fewer unsatisfied clauses than WALKSAT on these problems.

Second, I studied each of the WALKSAT parameter differences between Experiments (W) and (S/W3) in isolation. Specifically, I ran eight head-to-head experimental comparisons of WALKSAT and STAGE/WALKSAT with the WALKSAT parameters fixed to be the same in both algorithms. The eight experiments corre-

Algorithm	≈ time	moves accepted	extra time to undo moves	extra time for memory management
HC	3200s	61%	⏱	
S/HC (STAGE, π = hillclimbing)	5000s	38%	⏱⏱	⏱
W (WALKSAT)	1900s	100%		
S/W1 (STAGE, π = WALKSAT)	4100s	69%	⏱	⏱
S/W2	3200s	54%	⏱	
S/W3	3900s	70%	⏱	⏱
S/W1+linear	4100s	55%	⏱	⏱
S/W2+linear	3100s	55%	⏱	
S/W3+linear	3600s	97%		⏱⏱

TABLE 4.14. Approximate running times on instance par32-1.cnf. The stopwatch icons indicate which aspects of search caused running time to exceed WALKSAT's.

sponded to all combinations of the following settings:

$$(\mathit{cutoff}, \mathit{noise}, \delta_w) \in \{10^6, 10^4\} \times \{0, 0.25\} \times \{\infty, 10\}$$

The results were as follows. In two of the eight comparisons, namely, where $\mathit{cutoff} = 10^6$ and $\mathit{noise} = 0$, WALKSAT and STAGE performed statistically equivalently; STAGE's adaptive restarting performed neither better nor worse than WALKSAT's random restarting. In the other six comparisons, however, STAGE improved performance dramatically over plain WALKSAT.

Third, in an attempt to get STAGE to satisfy *all* the clauses of formula par32-1.cnf, I ran eight additional runs of (S/W3+linear) with an extended limit of TOTEVALS $= 2 \times 10^9$ bit flips. These runs used about 24 hours of computation each. The result was that one run left 1 clause unsatisfied, five runs left 2 clauses unsatisfied, and two runs left 3 clauses unsatisfied. None solved the formula. Future work will pursue this further: I hope a more insightful set of features, a less hasty job of parameter-tuning, or longer runs will enable STAGE to "cross the finish line." In any event, STAGE certainly shows promise for hard satisfiability problems—perhaps especially for MAXSAT problems where near-miss solutions are useful [Jiang *et al.* 95].

4.8 Boggle Board Setup

In the game of Boggle, 25 cubes with letters printed on each face are shaken into a 5×5 grid (see Figure 4.18).[5] The object of the game is to find English words

[5] Boggle is published by Parker Brothers, Inc. The 25-cube version is known as "Big Boggle" or "Boggle Master."

that are spelled out by connected paths through the grid. A legal path may include horizontal, vertical, and/or diagonal steps; it may not include any cube more than once. Long words are more valuable than short ones: the scoring system counts 1 point for 4-letter words, 2 points for 5-letter words, 3 points for 6-letter words, 5 points for 7-letter words, and 11 points for words of length 8 or greater.

G	R	R	W	Y
H	W	X	V	P
Z	K	Y	T	W
D	J	G	D	D
Y	A	D	S	Y

R	S	T	C	S
D	E	I	A	E
G	N	L	R	P
E	A	T	E	S
M	S	S	I	D

FIGURE 4.18. A random Boggle board (8 words, score=10, Obj=−0.010) and an optimized Boggle board (2034 words, score=9245, Obj=−9.245). The latter includes such high-scoring words as *depreciated*, *distracting*, *specialties*, *delicateness* and *desperateness*.

Given a fixed board setup x, finding *all* the English words in it is a simple computational task; by representing the dictionary[6] as a prefix tree, Score(x) can be computed in about a millisecond. It is a difficult optimization task, however, to identify what fixed board x^* has the highest score. For consistency with the other domains of this chapter, I pose the problem as a minimization task, where $\text{Obj}(x) = -\text{Score}(x)/1000$. (I used the scaling factor of 1/1000 to avoid a possible repeat of the flat simulated annealing performance curves found and explained in Section 4.4.) Note that I allow any letter to appear in any position of x, rather than constraining them to the faces of real 6-sided Boggle cubes. Exhaustive search of 26^{25} Boggle boards is intractable, so local search is a natural approach.

I set up the search space as follows. The initial state is constructed by choosing 25 letters uniformly at random. Then, to generate a neighboring state, either of the following operators is applied with probability 0.5:

- Select a grid square at random and choose a new letter for it. (The new letter is selected with probability equal to its unigram frequency in the dictionary.)

[6]My experiments make use of the 126,468-word Official Scrabble Player's Dictionary.

- Or, select a grid square at random, and swap the letter at that position with the letter at a random adjacent position.

The following features of each state x were provided for STAGE's learning:

1. The objective function, $\mathrm{Obj}(x) = -\mathrm{Score}(x)/1000$.
2. The number of vowels on board x.
3. The number of distinct letters on board x.
4. The sum of the unigram frequencies of the letters of x. (These frequencies, computed directly from the dictionary, range from $\mathrm{Freq}(e) = 0.1034$ to $\mathrm{Freq}(q) = 0.0016$.)
5. The sum of the bigram frequencies of all adjacent pairs of x.

These features are cheap to compute incrementally after each move in state space, and intuitively should be helpful for STAGE in learning to distinguish promising from unpromising boards.

Parameter	**Setting**
π	stochastic hillclimbing, patience=1000
ObjBound	$-\infty$
features	5 (Obj, # vowels, # distinct, $\sum$ unigram, $\sum$ bigram)
fitter	quadratic regression
Pat	200
TotEvals	100,000

Table 4.15. Summary of STAGE parameters for Boggle results

However, STAGE's results on Boggle were disappointing. STAGE's parameters are shown in Table 4.15 and comparative results are shown in Figure 4.19 and Table 4.16. Average runs of hillclimbing (patience=1000), simulated annealing, and STAGE all reach the same Boggle score, about 8400–8500 points.

Boggle is the only domain I have tried on which STAGE's learned smart restarting does not improve significantly over random-restart hillclimbing. This provides an interesting opportunity to compare the properties of a domain that is poor for STAGE with the other domains; I do so in Section 5.1.4.

Instance	Algorithm	Performance (100 runs each)			≈	moves
		mean	best	worst	time	accepted
5 × 5	HC	**-8.413**±0.066	-9.046	-7.473	2235s	2.4%
	SA	**-8.431**±0.086	-9.272	**-7.622**	1720s	33%
	STAGE	**-8.480**±0.077	**-9.355**	-7.570	2450s	1.3%

TABLE 4.16. Boggle results. The mean performances do not differ significantly from one another.

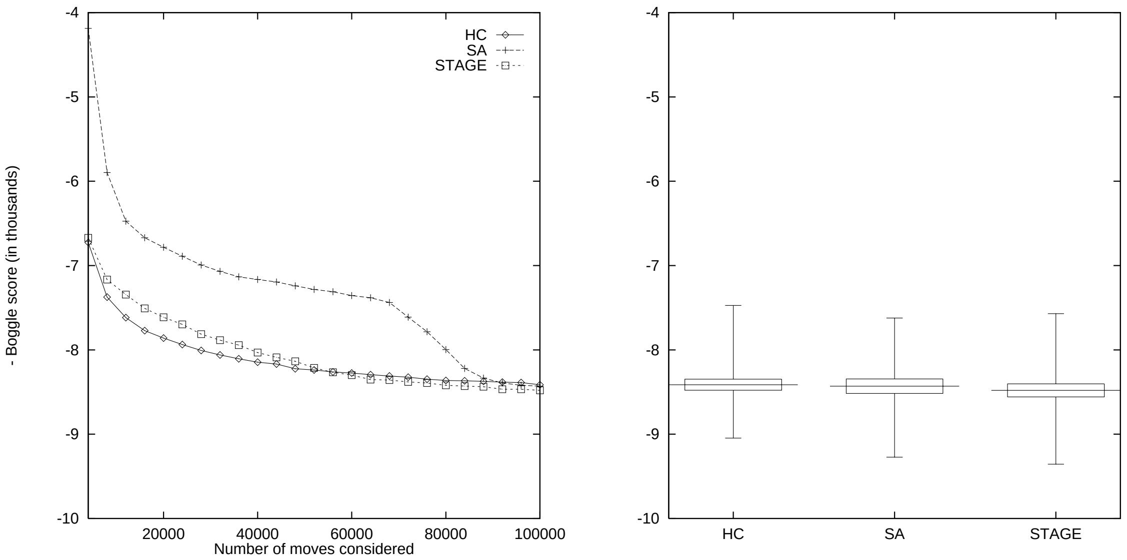


FIGURE 4.19. Performance on the Boggle domain

4.9 Discussion

This chapter has demonstrated that STAGE may be profitably applied to a wide variety of global optimization problems. In addition to the domains reported here, an algorithm closely related to STAGE has also been successfully applied to the "Dial-a-Ride" problem, a variant of the Travelling Salesperson Problem, by Moll et al. [97]. Empirically, in both discrete domains (e.g., channel routing, Bayes net structure-finding, satisfiability) and continuous domains (e.g., cartogram design), STAGE is able to learn from and improve upon the results of local search trajectories. In the next chapter, I will probe more deeply into the reasons for STAGE's success.

Chapter 5
STAGE: Analysis

In the preceding chapter, I gave empirical evidence that STAGE is an effective optimization technique for a wide variety of domains. In this chapter, I break down the causes for that effectiveness. Does STAGE's power derive from its use of machine learning, per its design? Or are incidental aspects of the way it organizes its search just as responsible for STAGE's success? How robust is the algorithm to varying choices of feature sets, function approximators, and other user-controllable parameters?

This chapter reports the results of a series of experiments that explore these questions empirically. Most of the experiments are performed in the domains of VLSI channel routing, as described in Section 4.3, and cartogram design, as described in Section 4.6.

5.1 Explaining STAGE's Success

STAGE performs superbly on the channel routing domain, not only outperforming hillclimbing as it was trained to do, but also finding better solutions on average than the best simulated annealing runs. How can we explain its success? There are at least three possibilities:

Hypothesis A: STAGE works according to its design. Gathering data from a number of hillclimbing trajectories, it learns to predict the outcome of hillclimbing starting from various state features, and it exploits these predictions to reach improved local optima.

Hypothesis B: Since STAGE alternates between simple hillclimbing and another policy, it simply benefits from having more random exploration. STAGE uses hillclimbing on the learned $\tilde{V}^{\pi}$ as its secondary policy, but alternative policies would do just as well.

Hypothesis C: The function approximator may simply be smoothing the objective function, which helps eliminate local minima and plateaus.

In this section, I first describe experiments which reject the latter two hypotheses. I then describe experiments giving further evidence for Hypothesis A, that STAGE does indeed work as designed. Finally, I analyze an instance where STAGE failed.

5.1.1 $\tilde{V}^\pi$ versus Other Secondary Policies

Hypothesis B attributes STAGE's success to its unusual regime of alternating between hillclimbing and a secondary search policy. Perhaps any secondary policy which perturbed the search away from its current local optimum would be just as effective as STAGE's secondary policy of hillclimbing on the learned evaluation function $\tilde{V}^\pi$.

To test this, I ran experiments with several alternative choices for STAGE's secondary policy:

Policy B0 (normal STAGE): Hillclimb on $\tilde{V}^\pi$.

Policy B1: Perform a random walk of fixed length ω. (I tried setting ω to 1, 3, 10, and 40.)

Policy B2: Hillclimb on the *inverse* of $\tilde{V}^\pi$. In other words, move stochastically to a state which $\tilde{V}^\pi$ predicts to be the *worst* place from which to begin a search.

Policy B3: Hillclimb on a corrupted version of $\tilde{V}^\pi$, trained by replacing every target value in its training set with a random value.

Each of these policies was alternated with standard hillclimbing (patience=250), so as to imitate STAGE's normal regime. Experiments B0, B2 and B3 approximated $\tilde{V}^\pi$ using linear regression over the three channel routing features $\langle w, p, U \rangle$, each scaled to the range $[0, 1]$. I ran each resulting algorithm 50 times on channel routing instance YK4, limiting each run to $\text{TotEvals} = 10^5$ total moves considered. (Note that the results of Section 4.3 were tabulated over longer runs of length $\text{TotEvals} = 5 \cdot 10^5$.)

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
YK4	B0 (STAGE)	**13.54**±1.13	**12**	41
	B1 ($\omega = 1$)	16.94±0.23	15	**19**
	B1 ($\omega = 3$)	17.16±0.24	16	**19**
	B1 ($\omega = 10$)	19.12±0.27	17	21
	B1 ($\omega = 40$)	22.64±0.26	20	25
	B2	37.86±3.52	14	52
	B3	15.38±0.64	13	27

TABLE 5.1. Results of different secondary policies on channel routing instance YK4. In each case, $\text{TotEvals} = 10^5$.

The results, given in Figure 5.1 and Table 5.1, show conclusively that the choice of secondary policy does matter. STAGE's policy significantly outperformed all others,

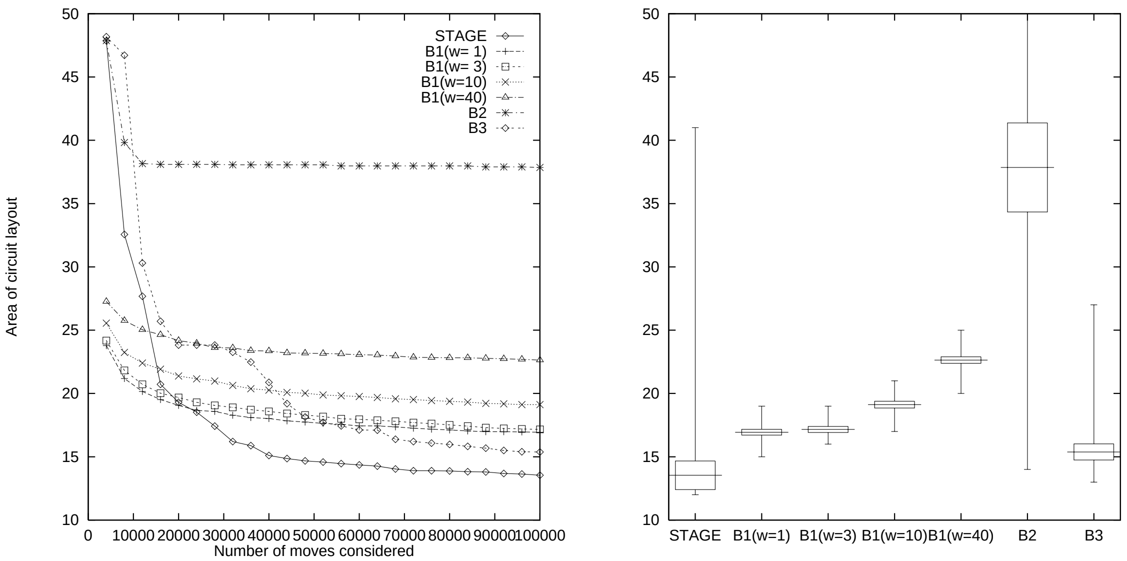


FIGURE 5.1. Performance of different secondary policies on channel routing instance YK4

with 48 of its 50 runs producing a solution circuit of quality between 12 and 14. (One outlier run did no better than 41; I analyze this in Section 5.1.3.) The random walk policies (B1) were consistently inferior. Indeed, the pattern of the B1 results shows clearly that performance degraded more and more as additional undirected exploration was allowed. Experiment B2, in which the search was purposely led against $\tilde{V}^\pi$ toward states judged *unpromising*, performed much worse than even the random-walk policies. This provides further evidence that $\tilde{V}^\pi$ has learned useful predictive information about the features.

Finally, Experiment B3 performed better than the random-walk policies, but still significantly worse than STAGE. Why did assigning random target values to the training set for $\tilde{V}^\pi$ result in even moderately good performance? The answer is that in this experiment, the linear regression model has only four coefficients to fit, and one of these (the coefficient of the constant term) has no effect when $\tilde{V}^\pi$ is used as an evaluation function to guide search. Thus, even choosing random values for the 3 meaningful coefficients would sometimes lead search in the same useful direction as the true V^π. This analysis is supported by the plot of Figure 5.2. This plot compares the successive local minima visited by a typical run of STAGE and a typical run of Experiment B3. Clearly, STAGE learns a policy which keeps it in a high-quality part of the space during most of its search, while B3 only occasionally "lucks into" a good solution. Still, these results highlight the potential for an alternative algorithm to

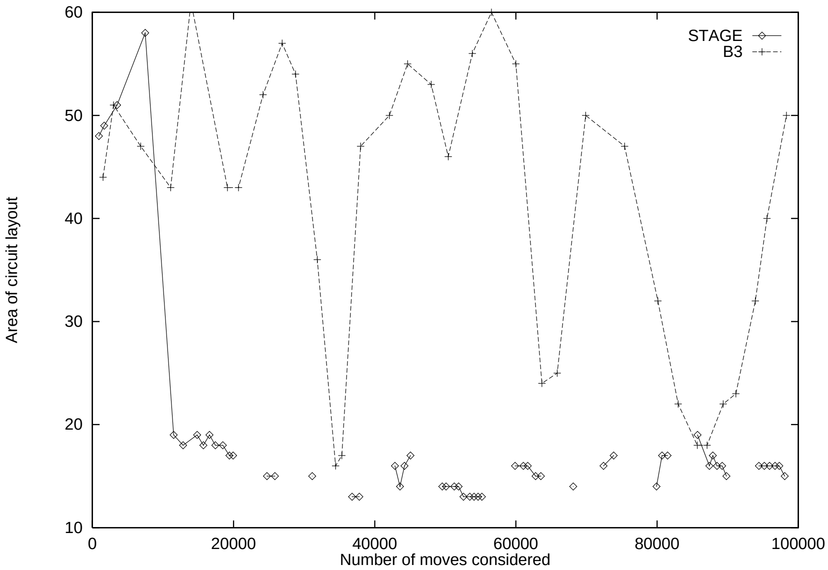


FIGURE 5.2. Quality of local minima reached on each successive iteration of a STAGE run and a B3 run. Gaps in the STAGE plot correspond to iterations on which a restart occurred.

STAGE, which works by optimizing the coefficients of a $\tilde{V}^\pi$-like function directly (see Section 8.2.3).

The results of this series of experiments do not prove that hillclimbing on $\tilde{V}^\pi$ is the only policy which can productively lead search away from a local minimum and into the attracting basin of a potentially better local minimum. However, they do at least demonstrate that not every secondary policy will be effective—and that STAGE's learned policy is more effective than most.

5.1.2 $\tilde{V}^\pi$ versus Simple Smoothing

STAGE performs "predictive smoothing" of the original objective function Obj: the function $\tilde{V}^\pi$ is a continuous mapping from the features of state x to the Obj value that policy π is predicted to eventually reach from x. But perhaps, as suggested by Hypothesis C above, the predictive aspect of STAGE's smoothing is irrelevant. Perhaps STAGE's success could be replicated by simply smoothing Obj directly over the feature space, which would eliminate the original objective function's local minima and plateaus.

One flaw of this hypothesis is immediately apparent: in channel routing as in most of the other domains described in Chapter 4, the objective function value Obj(x) is provided to STAGE as one of the features of the input feature space. Therefore, any function approximator could fit Obj perfectly over the feature space by simply copying the $\mathrm{Obj}(x)$ feature from input to output. Clearly, this perfect "smoothing" of Obj would not eliminate any local minima from Obj; STAGE would reduce to standard multi-restart hillclimbing. Experiment C0, described below, empirically demonstrates precisely this effect.

However, perhaps an imperfect smoothing of Obj would produce the hypothesized good outcome. I performed the following series of experiments. For all experiments except HC, the basic STAGE regime is unchanged; only the training set's target values for the function approximator are modified.

Experiment HC: Multi-restart hillclimbing, accepting equi-cost moves, $patience = 500$.

Experiment STAGE (same as Experiment B0): Normal STAGE, modelling $V^\pi(\mathrm{x})$ by linear regression over the three features $\langle w(x), p(x), U(x)\rangle$.

Experiment C0 (perfect "smoothing"): For each state in STAGE's training set, represented by the features $\langle w(x), p(x), U(x)\rangle$, train the function approximator to model not $V^\pi(\mathrm{x})$ but the objective function $\mathrm{Obj}(x) = w(x)$. Here, the function approximator can simply copy $w(x)$ from input to output, so no real smoothing occurs. Results are shown in Table 5.2 and Figure 5.3: as expected, this policy performed similarly to multi-start hillclimbing.

Experiment C1: I eliminated the w feature from the training set, so the function approximator had to model $\mathrm{Obj}(x)$ as a linear function of only $\langle p(x), U(x)\rangle$. The results were extremely poor. A closer look explained why. Recall that Wong defined

$$U(x) = \sum_{i=1}^{w(x)} u_i(x)^2$$

where $u_i(x)$ is the fraction of track i that is unoccupied [Wong *et al.* 88]. In poor-quality solutions such as those visited at the beginning of a search, $u_i(x)$ is close to 1 for each track, so $U(x) \approx \mathrm{Obj}(x)$. Thus, even a linear function approximator can model Obj quite accurately by simply copying $U(x)$ from input to output. But this smoothed objective function, $U(x)$, makes a terrible evaluation function for optimization. Starting from a local optimum of Obj, where no single move can reduce $w(x)$ further, the only way a greedy search can reduce U is to reduce the variance of the $u_i(x)$ values, that is, to distribute the wires evenly throughout the available tracks. That is exactly the opposite of the successful strategy that STAGE discovers: to distribute the wires as unevenly as possible, thereby creating some near-empty tracks which hillclimbing can more readily eliminate.

Experiment C2: Using the insight gained from the last experiment, I replaced $U(x)$ by a closely related feature:

$$V(x) \stackrel{\text{def}}{=} \sum_{i=1}^{w(x)} (1 - u_i(x))^2$$

With a bit of algebra, it can be shown that $V(x) \equiv U(x) - w(x) + C$, where C is a constant. With $w(x)$ subtracted from the feature, the regression model can no longer fit Obj well by simply copying a feature through. Indeed, after 10^5 steps of STAGE, the linearly smoothed fit of Obj over $\langle p(x), V(x) \rangle$ has an RMS error around 13, compared with an RMS error of less than 0.1 in Experiment C1. Thus, this fit does perform significant smoothing. The smoothed fit assigns a negative coefficient to $V(x)$, so searching on it tends to *increase* the variance of the u_i, resulting in improved overall performance relative to both C0 and C1. Nevertheless, C2's performance is still much worse than STAGE's. (In Section 5.2.1 below, I show that STAGE's performance with the p and V features is even better than STAGE with w, p and U.)

Instance	Algorithm	Performance (50 runs each)			RMS
		mean	best	worst	of fit
YK4	HC	21.32±0.22	19	23	—
	STAGE	**13.54**±1.13	**12**	41	2.1
	C0	21.06±0.74	15	27	0.0
	C1	40.00±0.18	38	41	0.05
	C2	19.34±1.00	15	28	13.6

TABLE 5.2. Performance comparison of STAGE with STAGE-like algorithms that simply smooth Obj. For each run, TOTEVALS = 10^5. The RMS column gives a typical root-mean-square error of the learned evaluation function over its training set at the end of the run.

In addition to the results plotted above, I also repeated experiments C1 and C2 using quadratic rather than linear regression to smooth Obj. This did decrease the RMS error of the approximations, to approximately 0.04 for C1 and 3.9 for C2.

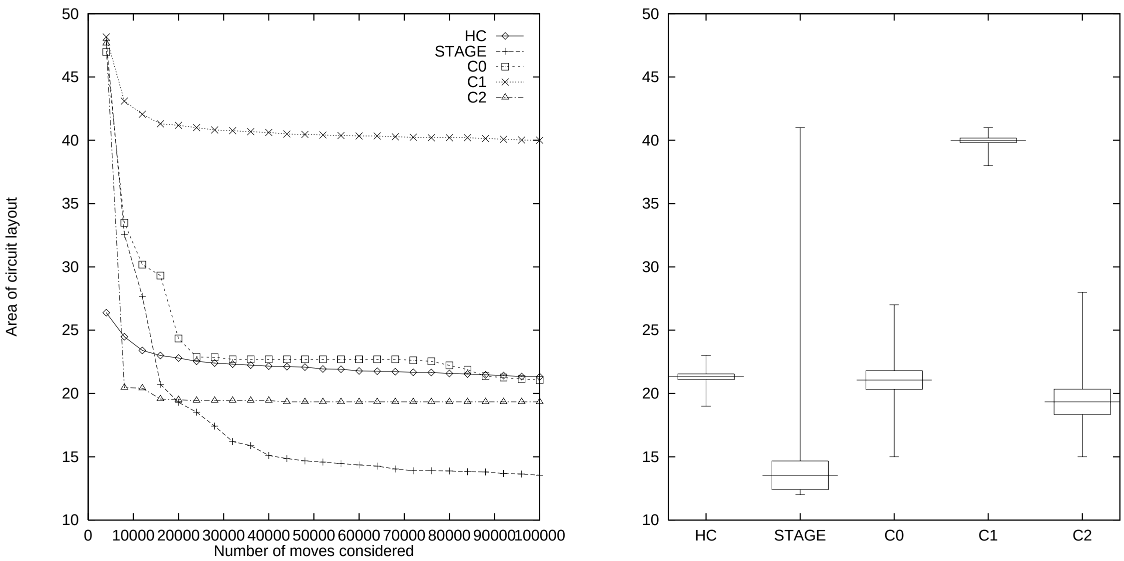


FIGURE 5.3. Performance on the experiments of Section 5.1.2

However, it did not improve optimization performance in either case. From this series of experiments, I conclude that, at least for this problem, STAGE's success cannot be attributed to simple smoothing of the objective function. STAGE's predictive smoothing works significantly better.

5.1.3 Learning Curves for Channel Routing

I have presented evidence that STAGE's secondary policy, searching on $\tilde{V}^\pi$, is more helpful to optimization than other reasonable policies based on randomness or smoothing. This evidence contradicts Hypotheses B and C outlined on page 109. I now examine STAGE's success on channel routing more closely, to support Hypothesis A: that STAGE's leverage is indeed due to machine learning.

Figure 5.4 illustrates three short runs of STAGE on routing instance YK4. In each row of the figure, the left-hand graph illustrates the successive local minima visited over the course of STAGE's run: a diamond symbol ($\diamondsuit$) marks a local minimum of Obj, and a plus symbol (+) marks a local minimum of the learned evaluation function $\tilde{V}^\pi$. There is a gap in the plot each time STAGE resets search to the poor initial state. Observe that the first two runs, (A) and (B), perform well, both attaining a best solution quality of 12; whereas run (C) performs uncommonly poorly, with a best solution quality of 34.

The right-hand graphs track the evolution of $\tilde{V}^\pi$ over the course of each run, plotting the coefficients β_w, β_p, and β_U on each iteration, where

$$\tilde{V}^\pi(x) = \beta_w \cdot w(x) + \beta_p \cdot p(x) + \beta_U \cdot U(x) + \beta_1$$

For clarity, the constant coefficient β_1 is omitted from the plots: its value, typically around -80, has no effect on search since it leaves the relative ordering of states unchanged.

The coefficient plots show that all runs converge rather quickly to similar $\tilde{V}^\pi$ functions. The learned $\tilde{V}^\pi$ functions have a high positive coefficient on $w(x)$, a negative coefficient of nearly equal magnitude on $U(x)$, and a coefficient near zero on $p(x)$. The assignment of a negative coefficient to U is surprising, because U measures the sparseness 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 search toward terrible solutions in which each subnet occupies its own track. Indeed, the hand-tuned evaluation function built by Wong et al. for this problem assigned $\beta_U = +10$ [Wong *et al.* 88].

However, the positive coefficient on w cancels out this bias. Recall from Experiment C2 of the previous section that the following relation holds:

$$V(x) = U(x) - w(x) + C$$

where $V(x)$ is a feature measuring the variance in track fullness levels, and C is a constant. By assigning $\beta_w \approx -\beta_U$, STAGE builds the evaluation function

$$\begin{aligned}\tilde{V}^\pi(x) &\approx -\beta_U \cdot w(x) + \beta_p \cdot p(x) + \beta_U \cdot U(x) + \beta_1 \\ &= \beta_U(U(x) - w(x)) + \beta_p \cdot p(x) + \beta_1 \\ &= \beta_U \cdot V(x) + \beta_p \cdot p(x) + \beta_1'\end{aligned}$$

Thus, in order to predict search performance, STAGE learns to extract the variance feature $V(x)$. The coefficient β_U is negative, so minimizing STAGE's learned evaluation function biases search toward increasing $V(x)$—that is, toward creating solutions with an uneven distribution of track fullness levels. Although this characteristic is not itself the mark of a high-quality solution, it does help lead hillclimbing search to high-quality solutions.

Given that all three STAGE runs of Figure 5.4 learned quickly to extract the variance feature $U - w$, why are their performance curves so different? A close look reveals that the third feature, $p(x)$, matters: performance breaks through to the "excellent" level when the coefficient β_p becomes positive. In run (A), which is most typical, β_p is positive throughout the entire run, and STAGE reaches excellent solutions on every iteration. In run (B), β_p becomes consistently positive only around

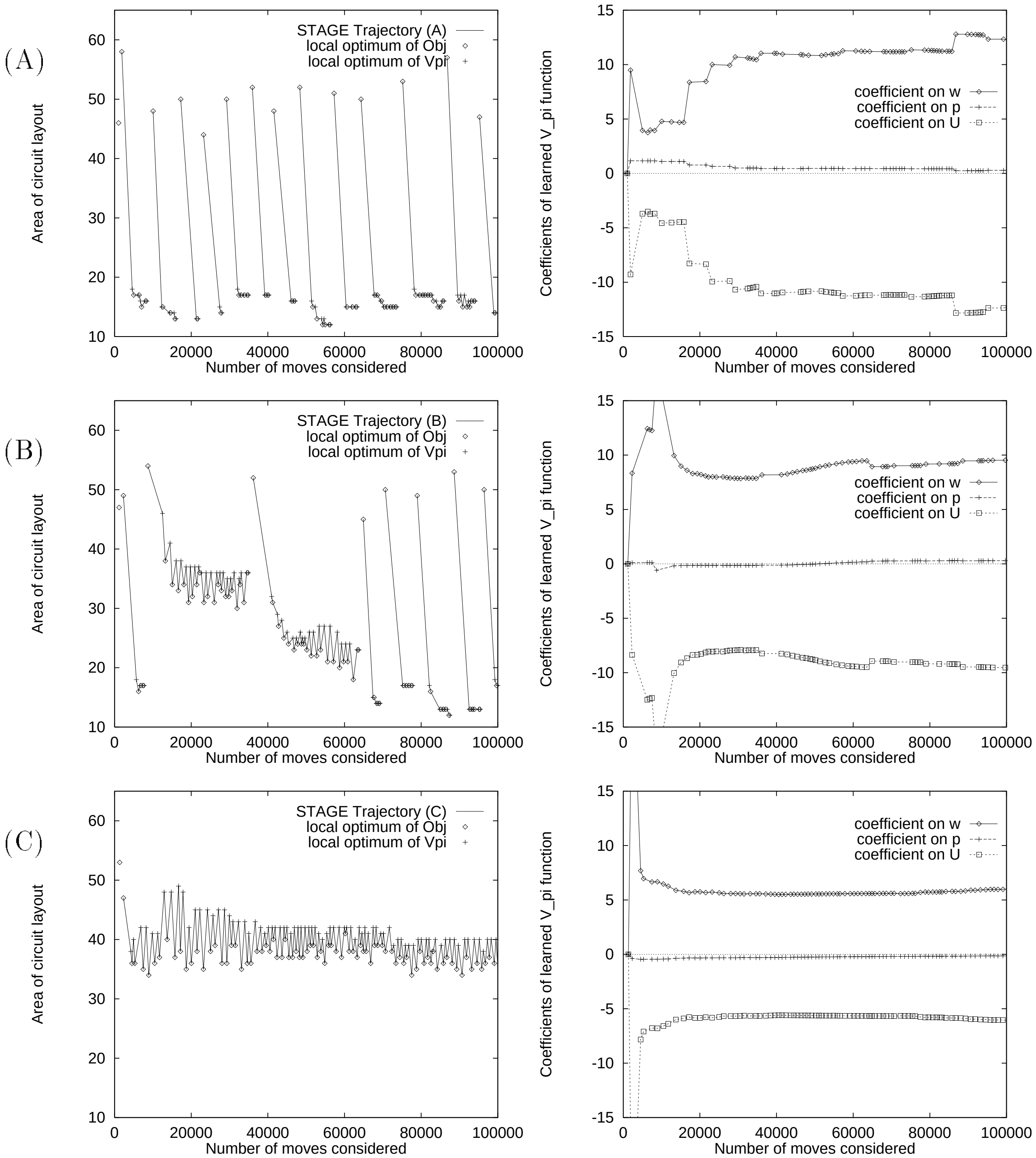
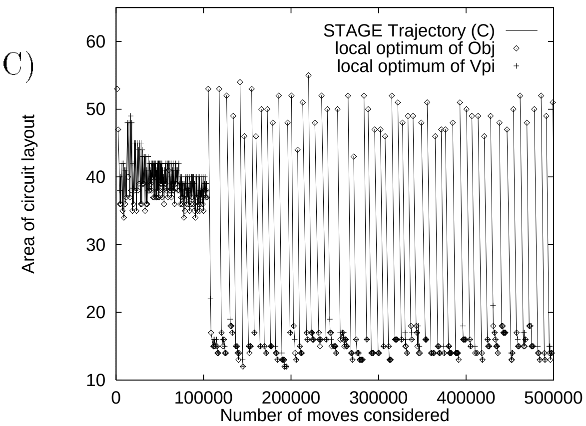


FIGURE 5.4. Three runs of STAGE on channel routing instance YK4. Left-hand graphs plot the Obj value reached at the end of each search trajectory within STAGE; right-hand graphs plot the evolution of the coefficients of $\tilde{V}^{\pi}(x)$ over the run.

(C)



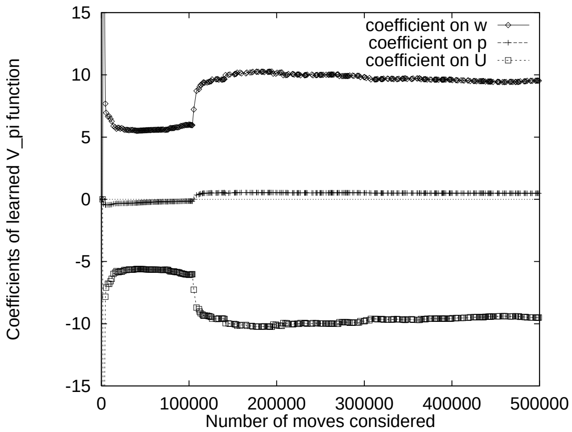


FIGURE 5.5. Run (C) of Figure 5.4, extended to TOTEVALS = 500,000.

move 60000, which corresponds to STAGE's breakthrough. Finally, in the outlier run (C), β_p remains negative throughout the first 100,000 moves. However, it is slowly increasing, and in fact, if the same run is allowed to extend to TOTEVALS = 500,000, then β_p soon becomes positive and performance improves (see Figure 5.5). This run reaches a solution quality of 12 after 145,000 moves.

These runs illustrate that STAGE's convergence rate may depend on the particular regions that STAGE happens to explore. In run (C), STAGE apparently becomes "stuck" in a region of poor local optima until the β_p coefficient becomes positive, triggering a random restart from which the learned $\tilde{V}^\pi$ can be effectively exploited. This suggests that faster convergence may be attained by more frequent random restarts; I investigate this in Section 5.2.4.

Despite the varying convergence rates, all the STAGE runs on instance YK4 do eventually converge to an approximation of V^π close to the following:

$$\tilde{V}^\pi(x) \approx 10 \cdot w(x) + 0.5 \cdot p(x) - 10 \cdot U(x) - 80 \tag{5.1}$$

The coefficients on all three features are significant. As discussed above, the component $10(w(x) - U(x))$ biases search toward solutions with an uneven distribution of track fullness levels. As for $p(x)$, recall from Section 4.3 that it provides a lower bound on all solutions derived from x by future merging of tracks. Thus, the coefficient of $+0.5$ on $p(x)$ helps keep the search from straying onto unpromising trajectories. In Section 6.2 I show that STAGE learns similar coefficients on a number of other instances of the channel routing problem.

5.1.4 STAGE's Failure on Boggle Setup

STAGE improved significantly over multi-restart hillclimbing on six of the seven large-scale optimization domains presented in Chapter 4. On the "Boggle setup" domain of Section 4.8, however, STAGE failed to produce a significant improvement. What explains this failure?

According to Hypothesis A, STAGE succeeds when it can identify and exploit trends in the mapping from starting state to hillclimbing outcome. As it turns out, the Boggle domain illustrates the converse: when the results of hillclimbing from a variety of starting states show no discernible trend, then STAGE will fail.

The following experiment makes this clear. I ran 50 restarts of hillclimbing for each of six different restarting policies:

random: Reassign all 25 tiles in the grid randomly on each restart.

EEE: Start with each tile in the grid to the letter 'E'.

SSS: Start with each tile in the grid to the letter 'S'.

ZZZ: Start with each tile in the grid to the letter 'Z'.

ABC: Start with the grid set to ABCDE/FGHIJ/KLMNO/PQRST/UVWXY.

cvcvc: Assign the first, third, and fifth rows of the grid to random consonants, and the second and fourth rows of the grid to random vowels. High-scoring grids often have a pattern similar to this (or rotations of this).

The boxplot in Figure 5.6 compares the performance of hillclimbing from these sets of states. Apparently, the restarting policy is irrelevant to hillclimbing's mean performance: on average, hillclimbing leads to a Boggle score near 7000 no matter which of the above types of starting states is chosen. Thus, STAGE cannot learn useful predictions of V^π, and its failure to outperform multi-restart hillclimbing is consistent with our understanding of how STAGE works.

5.2 Empirical Studies of Parameter Choices

We have shown that STAGE learns to predict search performance as a function of starting state features, and exploits these predictions to improve optimization results. However, STAGE's performance clearly depends on the user's choice of features $F(x)$, the function approximator used to model $\tilde{V}^\pi$, the policy π being learned, and other parameters. Section 3.4 analyzed these choices theoretically; this section continues the analysis from an empirical perspective.

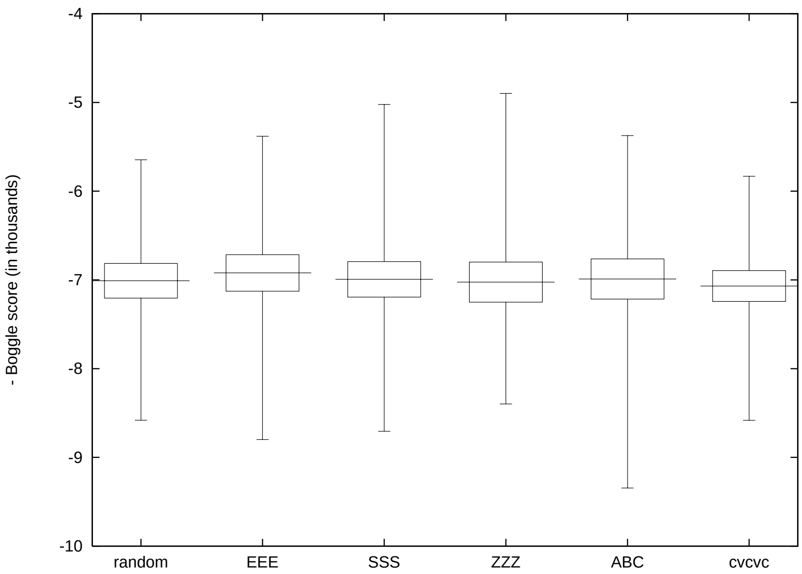


FIGURE 5.6. Boggle: average performance of 50 restarts of hillclimbing from six different sets of starting states

5.2.1 Feature Sets

Practical domains are generally abundant in potentially useful state features. But which of these should be given to STAGE for its learning? Providing many detailed state features, perhaps even a complete description of the state, gives STAGE the opportunity to model V^π very accurately; however, the extra features require more parameters to be fit, which increases the complexity of the fitter's task and may slow STAGE down intolerably. Conversely, using only a few coarse features results in efficient fitting, but limits the prediction accuracy that STAGE can achieve.

I compared STAGE's performance with a wide variety of feature sets on two domains: channel routing (instance YK4) and cartogram design (instance US49). These results are consistent with my informal experience with STAGE on many other domains: smaller feature sets learn faster and work better.

On the channel routing benchmark, I compared seven sets of features for representing any given state x. The other parameters to STAGE are detailed in Table 5.3. The results of 50 runs with each feature set are summarized in Figure 5.7 and Table 5.4. The table also gives the average final RMS error for $\tilde{V}^\pi$ on each experiment, but note that these numbers are not directly comparable, since the training set distributions can be markedly different from experiment to experiment. The feature sets tried were as follows:

WPU: $\langle w(x), p(x), U(x)\rangle$, the original features used in Section 4.3 and analyzed above.

PU: $\langle p(x), U(x)\rangle$ only, dropping the objective function $\text{Obj}(x) = w(x)$ from the input space. This performs badly, for the same reasons as Experiment C1 of Section 5.1.2 above. From this and other (unreported) experiments, I conclude that $\text{Obj}(x)$ should generally be included as a feature.

PV: $\langle p(x), V(x)\rangle$, where $V(x) \stackrel{\text{def}}{=} \sum_{i=1}^{w(x)}(1 - u_i(x))^2$, as suggested by Experiment C2 of Section 5.1.2 above. This performs well and, as can be seen in Figure 5.7, reaches good solutions very quickly! This shows that knowing what features are good for prediction and explicitly providing them leads to great performance. But it was through STAGE's success with the original features that this more compact feature set was discovered.

rrr: $\langle r_1(x), r_2(x), r_3(x)\rangle$, where $r_i(x)$ is a pseudorandom number in $[0, 1]$ chosen deterministically as a function of x and i. That is, these are three features of the state which are completely irrelevant for predicting V^π. As expected, this performs poorly, similar to the random-walk experiments of Section 5.1.1 above.

WPUrrr: $\langle w(x), p(x), U(x), r_1(x), r_2(x), r_3(x)\rangle$. This experiment tests whether performance degrades in the presence of irrelevant features. The result is encouraging: there is only a very slight performance degradation relative to WPU. Early on in each run, $\tilde{V}^\pi$ learns to assign near-zero coefficients to the irrelevant features.

WPU2: $\langle w(x), w(x)^2, p(x), p(x)^2, U(x)\rangle$. This includes the two quadratic terms that were used in the hand-tuned evaluation function for simulated annealing of [Wong *et al.* 88]. Apparently, these two terms are not useful to STAGE, as they are assigned relatively small coefficients and do not improve performance. (An outlier run in this series is responsible for the increased standard error of the mean.)

WPU2++: $\langle w(x), w(x)^2, p(x), p(x)^2, U(x), f_{t(1)}(x), f_{t(2)}(x), \ldots f_{t(144)}(x)\rangle$. Here, $t(i)$ denotes the current track $(1 \ldots w(x))$ on which subnet i has been placed, and $f_k(x) = 1 - u_k(x)$ denotes the fullness of track k. These 149 features provide much detail of state x, though not enough to reproduce x completely. The result: learning occurs slowly, both in terms of running time per iteration (since a 150×150 matrix must be inverted after each hillclimbing trajectory) and in terms of number of iterations to reach good performance. However, when these runs are allowed to extend ten times longer, to TOTEVALS $= 5{\cdot}10^6$, performance does reach the same excellent level as the other good STAGE runs.

Parameter	**Setting**
π	stochastic hillclimbing, rejecting equi-cost moves, patience=250
OBJBOUND	$-\infty$
features	between 2 and 149, depending on experiment
fitter	linear regression
PAT	250
TOTEVALS	500,000

TABLE 5.3. Summary of STAGE parameters for channel routing feature comparisons

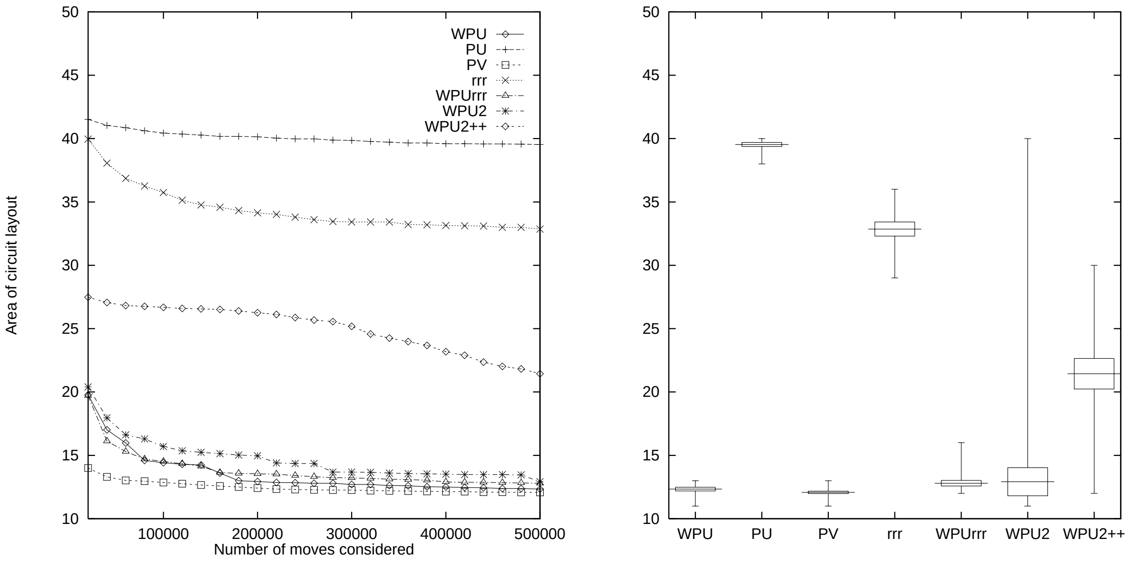


FIGURE 5.7. Performance of different feature sets on channel routing instance YK4

The second experiment with feature sets is from the U.S. cartogram domain. Recall from Section 4.6 that the objective is to deform the map's states so that their areas meet new prescribed targets (in this instance, proportional to electoral vote), but their shapes and connectivity remain similar to the un-deformed map. The objective function was defined as

$$\mathrm{Obj}(x) = \Delta\mathrm{Area}(x) + \Delta\mathrm{Gape}(x) + \Delta\mathrm{Orient}(x) + \Delta\mathrm{Segfrac}(x)$$

Instance	Algorithm	Performance (50 runs each)			RMS
		mean	best	worst	of fit
YK4	WPU	12.34±0.15	**11**	**13**	2.7
	PU	39.54±0.16	38	40	2.8
	PV	**12.08**±0.10	**11**	**13**	2.8
	rrr	32.86±0.56	29	36	3.7
	WPUrrr	12.80±0.22	12	16	2.7
	WPU2	12.92±1.11	**11**	40	2.7
	WPU2++	21.44±1.20	12	30	1.7

TABLE 5.4. Results with different feature sets on channel routing instance YK4

and STAGE used those four subcomponents of the objective function as its features for learning. Many other features can be imagined for this domain; I examine STAGE's performance with several of them here. These experiments used linear regression to fit $\tilde{V}^{\pi}$. Other STAGE parameters are detailed in Table 5.5, and the results are shown in Figure 5.8 and Table 5.6. The feature sets compared are as follows:

Asp.1: $\langle \mathrm{Asp}(x) \rangle$, a single feature describing the aspect ratio of the current map's bounding rectangle. Note that most moves leave this feature unchanged. STAGE learns a coefficient near zero for this feature, and its performance of Obj ≈ 0.16 is about the same as that of multi-restart hillclimbing.

Peri.1: $\langle \mathrm{Peri}(x) \rangle$, a single feature which sums the perimeter over all the states. STAGE works surprisingly well with this feature. It learns a positive coefficient on $\mathrm{Peri}(x)$; that is, it learns that hillclimbing performs better when it begins from a map which is "scrunched."

Cop.2: the horizontal and vertical coordinates of map x's center of population. I thought these would be a good pair of features for the US49 domain, since a good cartogram should have its population center near the geometric center of the map. However, these features were ineffective, under both linear regression (shown in graphs) and quadratic regression (not shown).

Obj.4: $\langle \mathrm{Area}(x), \mathrm{Gape}(x), \mathrm{Orient}(x), \mathrm{SegFrac}(x) \rangle$, the four subcomponents of the objective function. These features worked quite well, though not as well as Peri.1 on average. Note that the better STAGE results of Section 4.6 used quadratic regression over these four features.

ObjPeri.5: Given the good performance of Peri.1 and Obj.4, it is natural to combine them into a 5-feature representation for STAGE. This feature set performed comparably to experiment Obj.4.

St.15: This larger feature set concentrates on five "important" states of the map: New York, Texas, Florida, California, and Illinois. For each of these it provides three features: the horizontal and vertical coordinates of the state's current centroid (expressed relative to the map's current bounding rectangle) and the state's area. Note that this is the first feature set I have tailored specifically to this instance. However, STAGE's performance with these features, while better than multi-restart hillclimbing, is worse than with most of the simple instance-independent feature sets discussed above.

St.147: This experiment includes the three centroid and area features for all 49 of the map's states. With this many features, STAGE is significantly slowed down. For a fixed number of moves, however, solution quality is significantly better than multi-restart hillclimbing.

This series of experiments shows that for almost any natural choice of domain features, STAGE is able to discover a structure over those features which can be exploited to improve performance significantly over multi-restart hillclimbing.

Parameter	**Setting**
π	stochastic hillclimbing, rejecting equi-cost moves, patience=200
ObjBound	0
features	between 1 and 147, depending on experiment
fitter	linear regression
Pat	200
TotEvals	1,000,000

Table 5.5. Summary of STAGE parameters for cartogram feature comparisons.

From the experiments reported here on the channel routing and cartogram domains, and from other informal experiments not shown, I conclude that STAGE works best with small, simple sets of features. Small feature sets not only are speediest for STAGE to work with, but can also, in cases such as cartogram Experiment Peri.1, provide a bias toward good extrapolation by $\tilde{V}^{\pi}$, enabling successful exploration. Features associated with subcomponents of the objective function, or the variance of such subcomponents, were often helpful. With the function approximators tested here, STAGE is not overly sensitive to the inclusion of extra random features; however, it

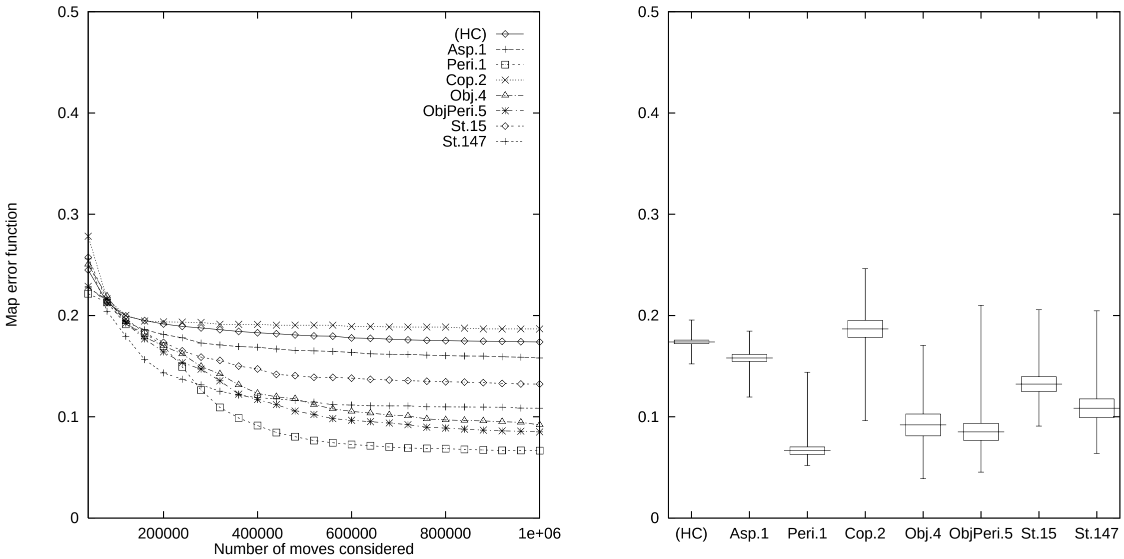


FIGURE 5.8. Cartogram performance with different feature sets

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
US49	Asp.1	0.158±0.004	0.120	0.185
	Peri.1	**0.067**±0.004	0.052	**0.144**
	Cop.2	0.187±0.008	0.096	0.246
	Obj.4	0.092±0.011	**0.039**	0.170
	ObjPeri.5	0.085±0.008	0.045	0.210
	St.15	0.132±0.007	0.091	0.206
	St.147	0.109±0.009	0.064	0.205

TABLE 5.6. Cartogram results with different feature sets

can sometimes be drawn into abysmal performance even worse than multi-restart hill-climbing by a particularly bad set of features, as in channel routing Experiment PU.

5.2.2 Fitters

Another key parameter to STAGE is the function approximator or "fitter" used to model $V^\pi(x)$. In Section 3.4.3, I argued that fitters having a linear architecture, such as polynomial regression (of any degree), are best suited to STAGE. Here, I empirically investigate the performance of STAGE with a range of polynomial fitters, ranging from degree 1 to degree 5, on both the channel routing and cartogram domains.

The table below lists the eight fitters I tested. Let $\langle f_1, f_2, \ldots, f_D \rangle$ denote the features of each domain given to STAGE; note that $D = 3$ in the channel routing domain and $D = 4$ in the cartogram domain. For each fitter, the table also gives the total number of coefficients being fit, both as a function of arbitrary D and in the case where $D = 4$.

Key	Description	# params
F1:	linear regression	$D + 1 = 5$
F2:	quadratic regression	$\binom{D+2}{2} = 15$
F3:	cubic regression	$\binom{D+3}{3} = 35$
F4:	quartic regression	$\binom{D+4}{4} = 70$
E2:	quadratic regression *without cross-terms*. For each domain feature f_i, this model includes the terms f_i and f_i^2 but no cross-terms involving the product of more than one feature.	$2D + 1 = 9$
E3:	cubic regression without cross-terms	$3D + 1 = 13$
E4:	quartic regression without cross-terms	$4D + 1 = 17$
E5:	quintic regression without cross-terms	$5D + 1 = 21$

Results are given in the usual form in Tables 5.7 and 5.8 and Figures 5.9 and 5.9. On both domains, the results indicate that the choice of fitter has a relatively minor impact on performance. On the channel routing domain, slightly better results and fewer poor outlier runs were obtained from the most highly biased models, linear regression and crossterm-less quadratic regression. This is consistent with our earlier observation that the linear approximation $V^\pi(x) \approx 10w(x) + 0.5p(x) - 10U(x) - 80$ suffices for excellent performance here. On the cartogram domain, however, somewhat more complex models performed slightly better.

In these experiments, STAGE's computational overhead for function approximation was not a significant component of running time, even for the most complex model here, quartic regression with all cross-terms. However, since the expense of least-squares quartic regression increases as $O(D^{12})$, this overhead would certainly become prohibitive for more than $D = 4$ features. In my experience, quadratic regression (either with or without cross terms) provides sufficient model flexibility, enough bias to enable aggressive extrapolation, and efficient computational performance.

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
YK4	F1 (linear)	**12.26**±0.17	**11**	14
	F2 (quadratic)	14.30±1.16	12	37
	F3 (cubic)	13.08±0.35	12	19
	F4 (quartic)	12.86±0.42	**11**	19
	E2 (crossterm-less quadratic)	**12.30**±0.14	**11**	**13**
	E3 (crossterm-less cubic)	**12.80**±1.07	12	39
	E4 (crossterm-less quartic)	13.54±1.15	**11**	39
	E5 (crossterm-less quintic)	15.02±0.71	12	24

TABLE 5.7. Channel routing results with different polynomial function approximators over the 3 features $\langle w, p, U\rangle$

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
US49	(HC)	0.174±0.002	0.152	0.195
	F1 (linear)	0.092±0.011	0.039	0.170
	F2 (quadratic)	**0.057**±0.004	**0.038**	0.103
	F3 (cubic)	**0.057**±0.004	0.038	0.126
	F4 (quartic)	0.069±0.003	0.045	**0.096**
	E2 (crossterm-less quadratic)	0.078±0.010	0.041	0.174
	E3 (crossterm-less cubic)	0.067±0.005	0.038	0.135
	E4 (crossterm-less quartic)	**0.058**±0.004	0.040	0.111
	E5 (crossterm-less quintic)	0.069±0.005	0.041	0.106

TABLE 5.8. Cartogram performance with different polynomial function approximators over the 4 features $\langle\Delta\text{Area}, \Delta\text{Gape}, \Delta\text{Orient}, \Delta\text{Segfrac}\rangle$. All the STAGE runs significantly outperform multi-restart hillclimbing (HC).

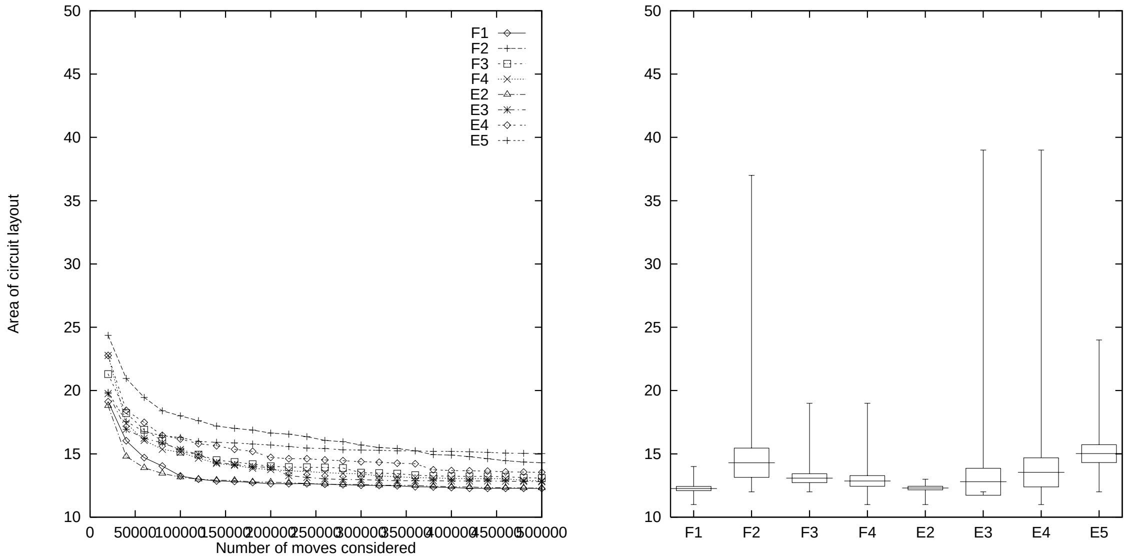


FIGURE 5.9. Performance of different function approximators on channel routing instance YK4

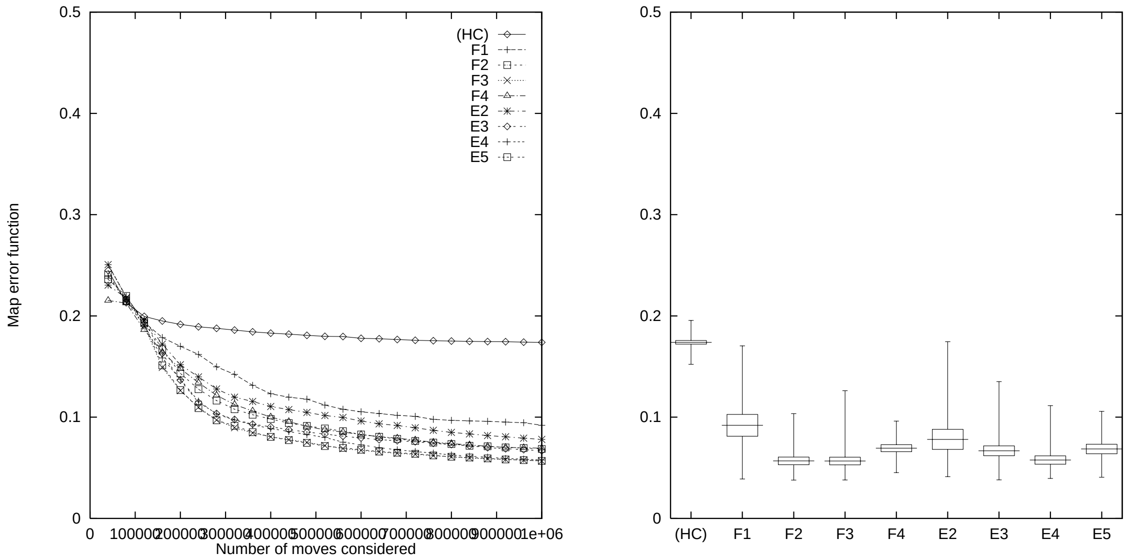


FIGURE 5.10. Cartogram performance with different polynomial function approximators

5.2.3 Policy π

Thus far, I have discussed how the user's choice of features and fitters affects STAGE's ability to learn and exploit $\tilde{V}^{\pi}(F(x))$. But what about the choice of π itself? In every domain of Chapter 4 besides Boolean satisfiability, π was chosen to be a very simple search procedure: stochastic first-improvement hillclimbing, rejecting equi-cost moves. But clearly, stronger local search procedures are available. For example, stochastic hillclimbing *accepting* equi-cost moves often performs better, and simulated annealing performs better still. Can STAGE learn to improve upon these, and achieve yet better performance?

Theoretically, even if a procedure π_1 is significantly better than another procedure π_2, it is by no means guaranteed that STAGE on π_1 will outperform STAGE on π_2. For example, π_2 may produce more diverse outcomes as a function of starting state, enabling useful extrapolation in $\tilde{V}^{\pi_2}$; whereas if π_1 is expected to reach the same solution quality no matter what state it starts from—as was the case with the Boggle example of Section 5.1.4—then $\tilde{V}^{\pi_1}$ will be flat, and STAGE will not produce improvement.

In this section, I consider the following three policies on the cartogram domain:

- π_1: regular first-improvement hillclimbing, rejecting equi-cost moves, patience=200.

- π_2: hillclimbing as above, but modified so that equi-cost and some slightly harmful moves are accepted. Specifically, a move is rejected if and only if it worsens Obj by more than δ=0.0001. (This value of δ was the most effective of a wide range tried on this domain.) On average, π_2 performs significantly better than π_1.

- π_3: simulated annealing with a shortened schedule length of 20,000 moves. The short schedule allows it to be used in the context of a multi-restart procedure.

For each of these three policies, I sought to compare the default restarting method to a "smart" restarting method learned by STAGE. In the cases of π_1 and π_2, the default restarting method is to reset to the domain's initial state (the original undeformed U.S. map). In the case of π_3, I found that a better default restarting method is to start each new annealing schedule in the same state where the previous trajectory finished. These procedures are labelled PI1, PI2 and PI3 in the results presented below.

In applying STAGE to π_2 and π_3, a complication arises: neither of these procedures is Markovian. Methods for coping with this theoretical difficulty were discussed in detail at the end of Section 3.4.1 (p. 63), and in the context of WALKSAT in Section 4.7.2. In the case of π_2, a simple fix is to train $V^{\pi_2}(x)$ on only those states which are the best-so-far on their trajectory. (Note that on the pure hillclimbing trajectories generated by π_1, every state is the best-so-far.) These procedures are labelled STAGE(PI1) and STAGE(PI2) in the results. In the case of π_3, even training only on best-so-far states does not make $V^{\pi_3}(x)$ theoretically well-defined (though results for STAGE(PI3) are given below nonetheless). Instead, giving up on the Markov property, I resort to training V^{π_3} on just the single starting state of each simulated annealing trajectory. This variant of STAGE is well-defined for any proper policy π; its results for all three of our policies are shown as STAGE0(PI1), STAGE0(PI2), and STAGE0(PI3).

The results, displayed in Table 5.9 and Figures 5.11–5.13, point to several conclusions. First, as demonstrated by Experiment STAGE(PI2), STAGE can successfully learn from a non-Markovian policy by training only on best-so-far states. Second, from Experiment STAGE(PI3) we can conclude that STAGE may not be effective at learning evaluation functions from simulated annealing; this is unsurprising, since SA's initial period of random search makes the outcome quite unpredictable from the starting state. Third, from the set of STAGE0 experiments, it is apparent that STAGE learns much more quickly when it is able to train on entire trajectories, rather than just starting states. Thus, STAGE is able to exploit the Markov property of V^π for efficient performance.

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
US49	PI1 = hillclimbing	0.174±0.002	0.152	0.192
	STAGE(PI1)	**0.057**±0.004	**0.038**	**0.103**
	STAGE0(PI1)	0.099±0.013	0.042	0.174
	PI2 = hillclimbing/δ=0.0001	0.140±0.003	0.115	0.167
	STAGE(PI2)	**0.052**±0.003	**0.040**	**0.083**
	STAGE0(PI2)	0.077±0.009	0.045	0.136
	PI3 = simulated annealing	**0.049**±0.001	0.044	**0.070**
	STAGE(PI3)	**0.050**±0.002	**0.042**	0.091
	STAGE0(PI3)	0.060±0.005	0.043	0.142

TABLE 5.9. Cartogram performance with STAGE learning $\tilde{V}^\pi$ from a variety of different choices of policy π. All algorithms were limited to considering TOTEVALS = 10^6 moves.

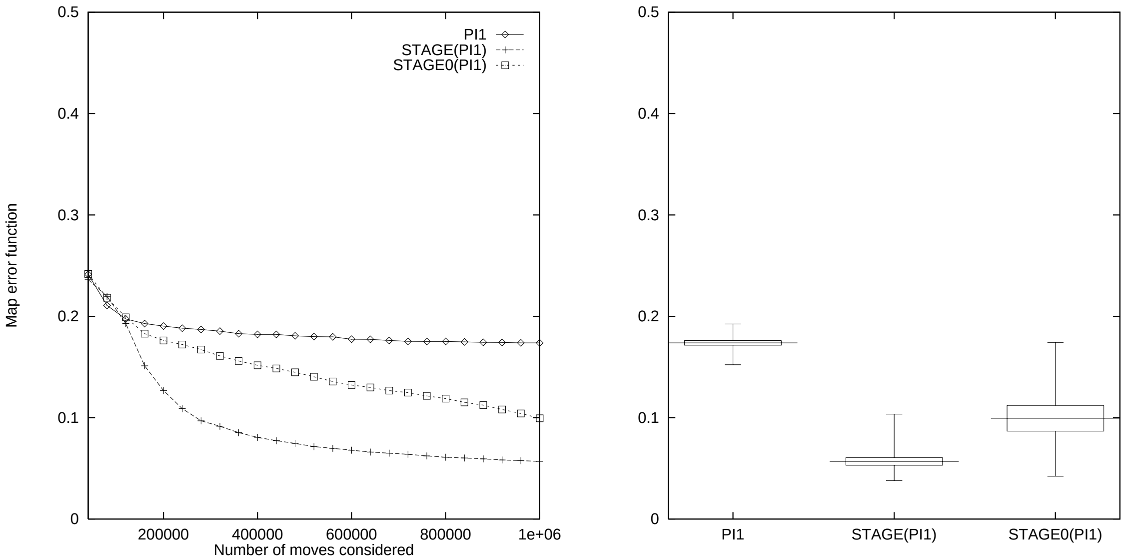


FIGURE 5.11. Cartogram performance of STAGE using π_1 = hillclimbing

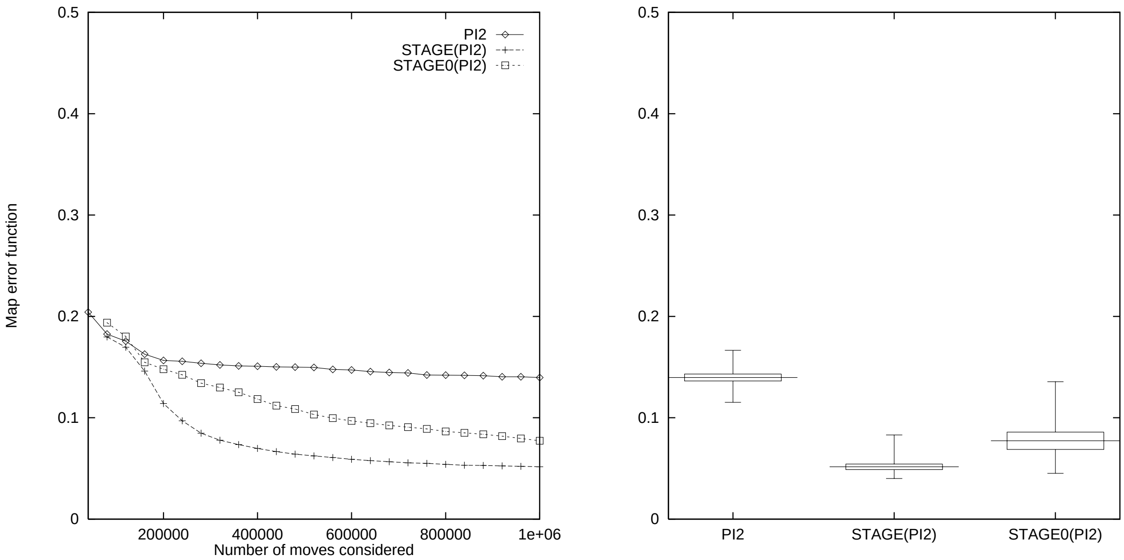


FIGURE 5.12. Cartogram performance of STAGE using π_2 = hillclimbing accepting equi-cost and slightly harmful moves

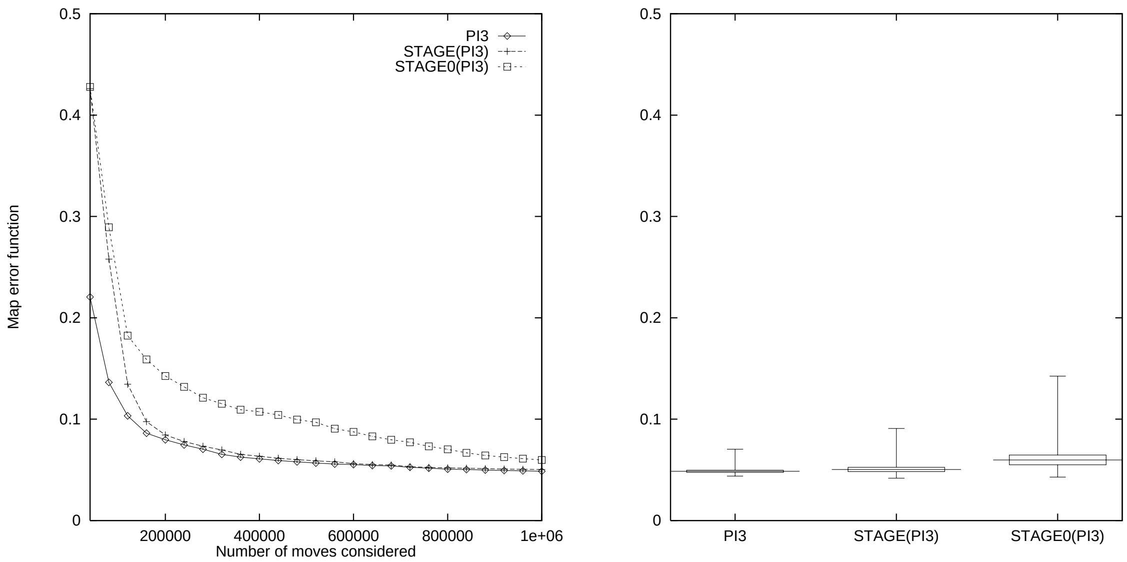


FIGURE 5.13. Cartogram performance of STAGE using π_3 = simulated annealing

5.2.4 Exploration/Exploitation

In any time-bounded search process there arises a tradeoff between *exploitation*, i.e., pursuing what appears to be the best path given the limited observations made thus far, and *exploration*, i.e., trying paths about which little is yet known. Treating this tradeoff optimally is in general intractable, and various heuristics have been proposed in the reinforcement learning literature [Thrun 92, Moore and Atkeson 93, Dearden *et al.* 98]. For its exploration, STAGE relies primarily on extrapolation by the function approximator to guide search to unvisited regions, and secondarily on random restarts when search stalls.

However, there is another aspect of STAGE's search strategy than impinges on the exploration/exploitation dilemma. On each iteration, STAGE runs π to generate a trajectory $(x_0 \ldots x_T)$, uses this trajectory to update $\tilde{V}^\pi$, and then searches for a good starting state of π by hillclimbing on the new $\tilde{V}^\pi$. This section investigates the design decision of where to begin each search of $\tilde{V}^\pi$:

Continue: This is STAGE's normal policy—namely, to begin each search of $\tilde{V}^\pi$ by simply continuing from the current local optimum x_T.

Re-init: Begin each search of $\tilde{V}^\pi$ from a random initial state. This promotes greater global *exploration* of the space, at the cost of losing the benefit of the work just done by π to reach a local optimum of Obj.

Compromises between **Continue** and **Re-init** are possible. Let **Re-init**(K) denote the policy of jumping to a random initial state if and only if K consecutive STAGE iterations have failed to improve on the best optimum yet seen. Thus, **Re-init**(0) always starts from a random state, and **Re-init**(∞) is the same as the **Continue** policy. I tested a range of settings for K.

Best-ever: Begin each search of $\tilde{V}^\pi$ from the best state seen so far on this run. This promotes greater *exploitation* of a known excellent solution, at the cost of possibly becoming over-focused on one area of search space.

Again, compromises are possible. In this case, I implemented **Best-ever**(p), which on each iteration either jumps to the best-so-far state with probability p or continues from the current state x_T with probability $1-p$. Note that **Best-ever**(0), like **Re-init**(∞), is equivalent to **Continue**.

I compared **Continue**, **Re-init**(0), **Re-init**(5), **Re-init**(20), **Best-ever**(0.1), **Best-ever**(0.4), **Best-ever**(0.7), and **Best-ever**(1) on cartogram domain US49. The results are given in Table 5.10 and Figure 5.10. The results show that frequent jumping to a random state, as **Re-init**(K) does for small K, does negatively affect solution quality, whereas frequent jumping to the best-so-far state, as done by **Best-ever**(p), neither helps performance nor hurts it significantly. STAGE's **Continue** policy seems to strike an acceptable balance between exploration and exploitation.

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
US49	**Continue** (normal STAGE)	**0.057**±0.003	**0.037**	0.103
	Re-init(0)	0.171±0.003	0.150	0.197
	Re-init(2)	0.072±0.004	0.045	0.112
	Re-init(5)	0.059±0.004	0.040	0.109
	Re-init(20)	**0.056**±0.005	0.037	0.114
	Best-ever(0.1)	**0.056**±0.004	0.037	**0.100**
	Best-ever(0.4)	0.062±0.006	0.038	0.147
	Best-ever(0.7)	0.060±0.005	0.041	0.129
	Best-ever(1.0)	0.062±0.004	0.039	0.113

Table 5.10. Cartogram performance under various modifications of STAGE's policy for initializing search of $\tilde{V}^\pi$ on each round

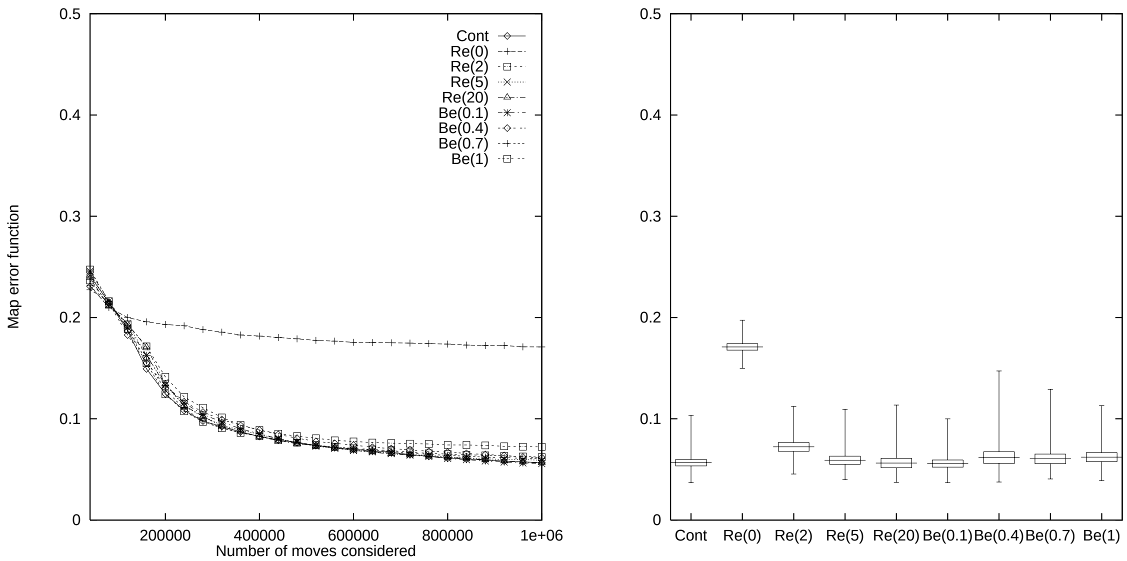


FIGURE 5.14. Cartogram performance under various modifications of STAGE's policy for initializing search of $\tilde{V}^\pi$ on each round

5.2.5 Patience and ObjBound

Finally, I investigated the effect of the two remaining STAGE parameters: PAT and OBJBOUND. Recall from the STAGE algorithm (p. 54) that these parameters jointly determine when STAGE's second phase of search, stochastic hillclimbing on $\tilde{V}^\pi$, should terminate. OBJBOUND corresponds to a known lower bound on the objective function; it cuts the search off when $\tilde{V}^\pi(F(x))$ predicts that x is an impossibly good starting state, i.e., $\tilde{V}^\pi(F(x)) <$ OBJBOUND. This cutoff may be disabled by setting OBJBOUND $= -\infty$. The other parameter, PAT, cuts the search off when too many consecutive moves have failed to improve $\tilde{V}^\pi$. If either of these parameters is set too aggressively (too tight a bound, too-low patience), then STAGE will fail to reach the best starting points predicted by its function approximator. If, on the other hand, these are set too loosely, then STAGE may waste valuable time that could instead have been spent on the first phase of search, namely, running π and gathering new training data. How sensitive is STAGE to the precise settings used?

Experimental results of varying these parameters on the cartogram domain are presented in Table 5.11 and Figures 5.15 and 5.16. STAGE's other parameters were set as in the cartogram experiments of Section 4.6 (p. 97). The results show that STAGE's performance suffered when PAT was very low or very high, but worked well for a wide range of settings between 64 and 1024. As for the OBJBOUND parameter, STAGE performed best when it was set to its true bound of 0, but degraded gracefully when it was set to -1 (too loose a bound) or $+0.1$ (too tight). Performance was, however, significantly worse for ObjBound=∞, demonstrating that the ObjBound parameter is useful. By cutting off search on $\tilde{V}^\pi$ when the function approximator is provably inaccurate, ObjBound saves computation time and prevents search from wandering too far astray.

Instance	Algorithm	Performance (50 runs each)		
		mean	best	worst
US49	Pat=16	0.161±0.015	0.079	0.285
	Pat=32	0.095±0.008	0.040	0.147
	Pat=64	0.065±0.006	0.040	0.151
	Pat=128	**0.057**±0.004	0.039	**0.114**
	Pat=256	**0.059**±0.007	**0.038**	0.190
	Pat=512	0.062±0.006	0.041	0.134
	Pat=1024	0.070±0.005	0.044	0.126
	Pat=2048	0.100±0.006	0.046	0.159
	Pat=4096	0.126±0.008	0.074	0.187
	Pat=8192	0.122±0.007	0.057	0.172
	Pat=16384	0.132±0.006	0.068	0.178
	Pat=32768	0.129±0.007	0.079	0.189
	Pat=65536	0.135±0.007	0.081	0.184
	ObjBound=$-\infty$	0.083±0.011	**0.036**	0.189
	ObjBound=-1	0.060±0.006	0.037	0.143
	ObjBound=0	**0.057**±0.004	0.038	**0.103**
	ObjBound=0.1	0.061±0.006	0.040	0.171

Table 5.11. Cartogram performance with varying settings of Pat and ObjBound

5.3 Discussion

This chapter gives depth to the results of the previous chapter, demonstrating that STAGE does indeed obtain its success by exploiting the power of reinforcement learning, and that it works reliably over a wide range of parameter settings. The chapter's empirical conclusions may be summarized as follows:

- Evaluation functions other than STAGE's learned $\tilde{V}^\pi$, built at random or by simply smoothing the objective function, perform much worse than $\tilde{V}^\pi$ in the context of multi-start optimization. **STAGE successfully learns and exploits the predictive power of value functions.**

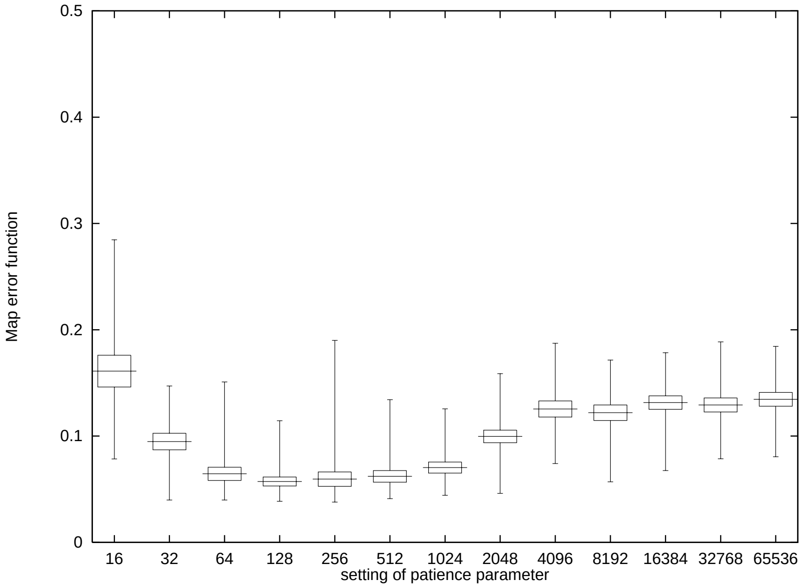


FIGURE 5.15. Cartogram performance with various patience settings

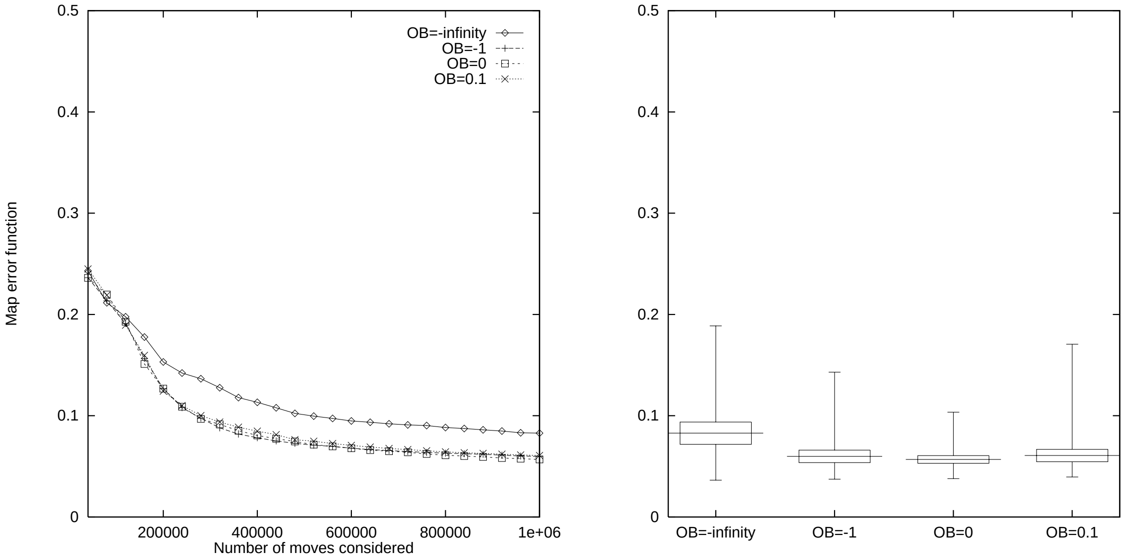


FIGURE 5.16. Cartogram performance with different OBJBOUND levels

- When the baseline policy π does not demonstrate predictable trends as a function of state features, then $\tilde{V}^\pi$ will not be able to guide STAGE to promising new starting states, so STAGE will not improve (nor hurt) performance compared to multi-start π. This was demonstrated for hillclimbing in the Boggle domain and for simulated annealing in the cartogram domain.

- For the policy of π=stochastic hillclimbing, STAGE can empirically learn a useful global structure from many different natural choices of feature sets. STAGE is robust in the presence of irrelevant features, though it works most efficiently and effectively with small feature sets. As a choice of function approximator, quadratic regression provides sufficient model flexibility, enough bias to enable aggressive extrapolation, and efficient computational performance.

- Empirically, STAGE makes reasonable tradeoffs between exploration and exploitation, and it performs robustly over a wide range of settings for the parameters PAT and OBJBOUND.

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 $x_t \neq$ 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 $x_t \neq$ 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 69). 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

$$\mathbf{Nv} = \mathbf{s} + \mathbf{Cv},$$

    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 $\mathbf{s}$:

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

where $\boldsymbol{\Phi}$ 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 $\tilde{V}^\pi$:

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

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 $\boldsymbol{\Phi}$ to the lookup-table value function:

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

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. 33). 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.

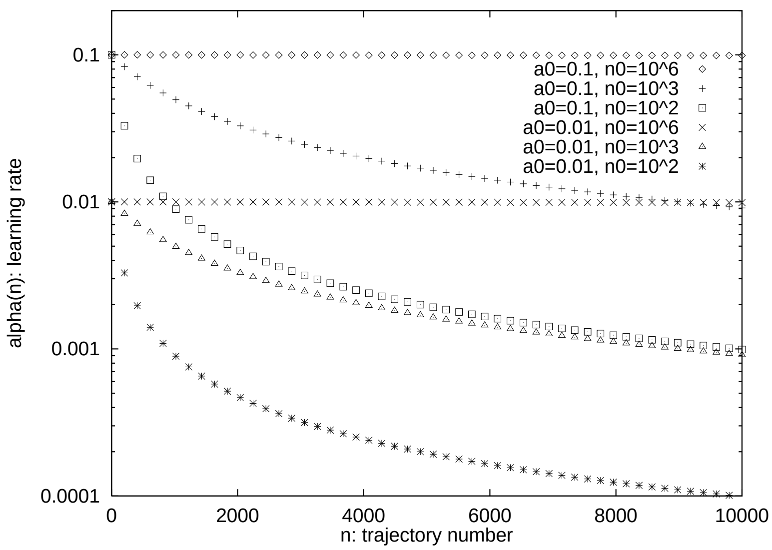


FIGURE 6.1. The six different *stepsize schedules* used in the experiments with TD(λ). The schedules are determined by Equation 6.7 with various settings for α_0 and n_0.

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.

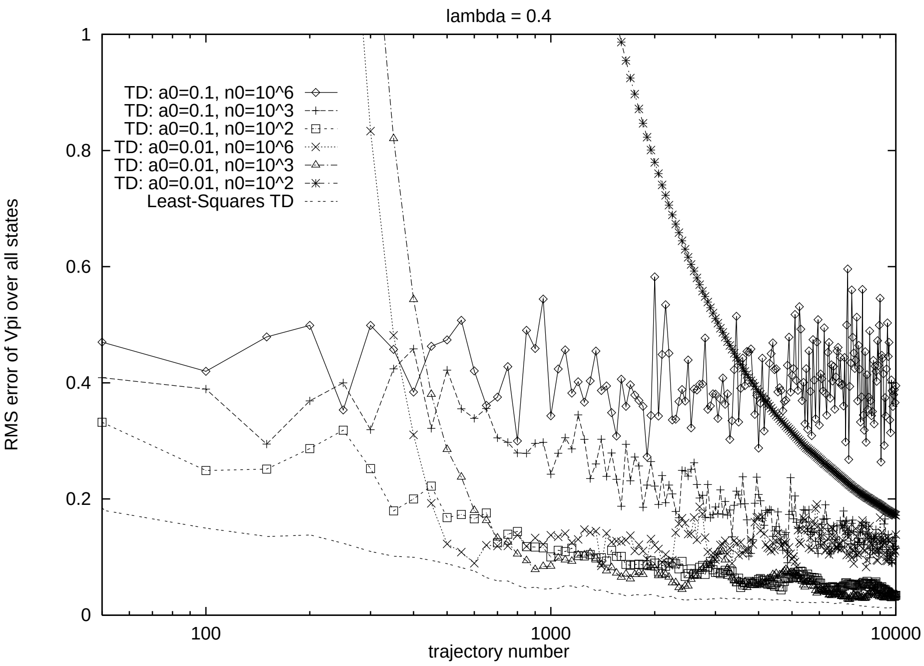


FIGURE 6.2. Performance of TD(0.4) and LSTD(0.4) on the Hopworld domain. Note the logarithmic scale of the x-axis.

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. 61). 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.

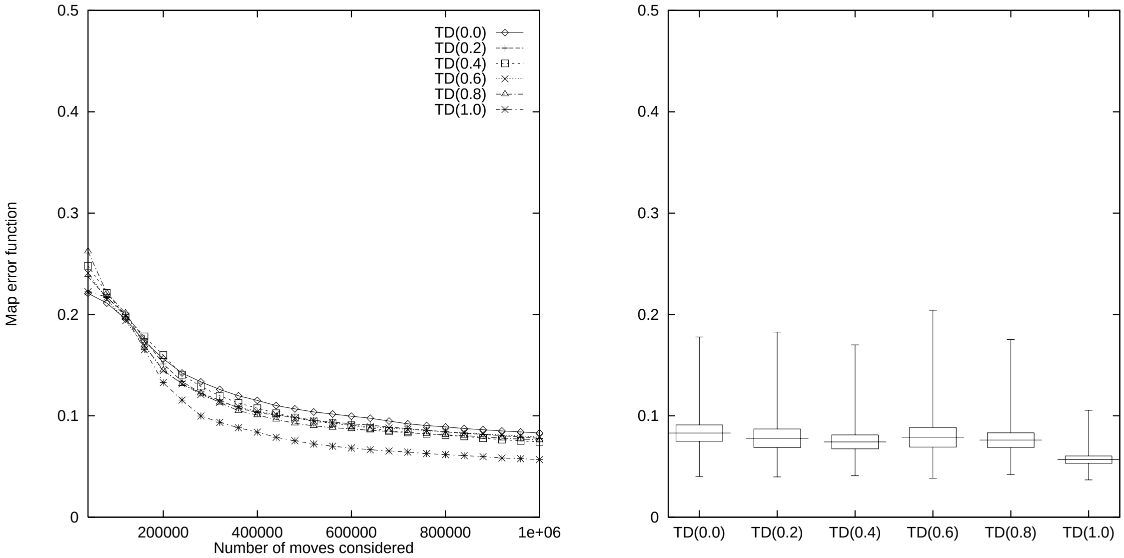


FIGURE 6.3. Cartogram performance with LSTD(λ)

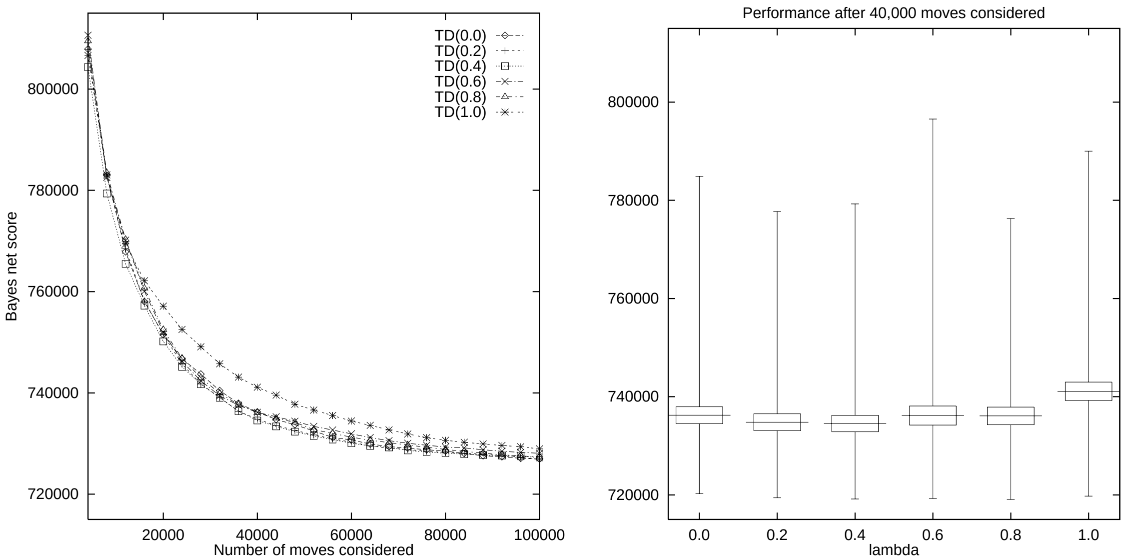


FIGURE 6.4. Performance of STAGE with LSTD(λ) on Bayes net structure-finding instance SYNTH125K. After 40,000 moves, LSTD(1.0) significantly lagged all other values for λ.

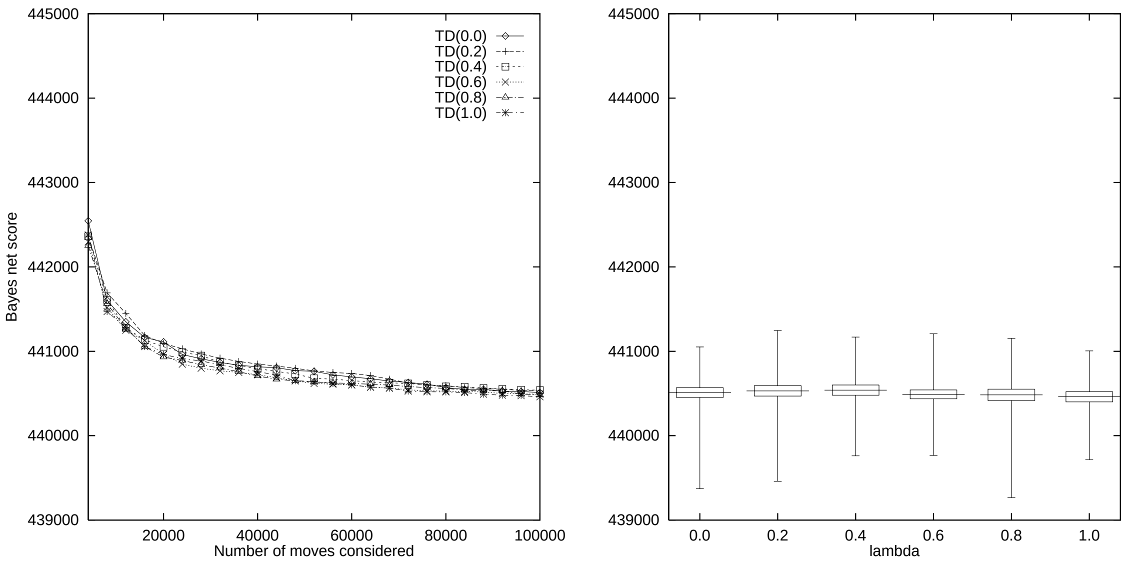


FIGURE 6.5. Performance of STAGE with LSTD(λ) on Bayes net structure-finding instance ADULT2

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 118), 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. 77), 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. 54). It is assumed that each instance's featurizer $F : X \rightarrow \Re^D$ 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. 54)—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.

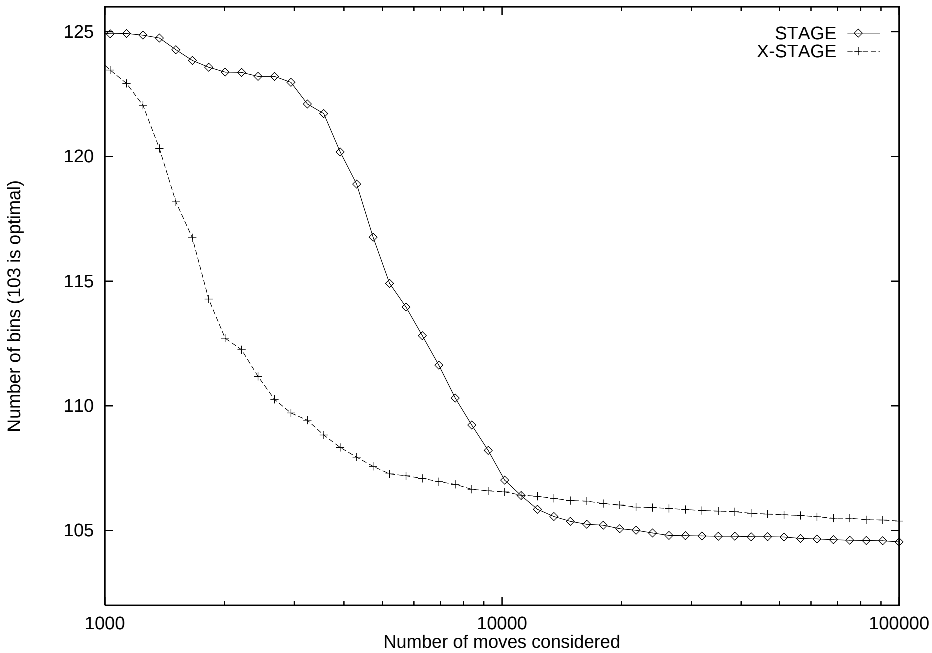


FIGURE 6.6. Bin-packing performance on instance u250_13 with transfer (X-STAGE) and without transfer (STAGE). Note the logarithmic scale of the x-axis.

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. 83). 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

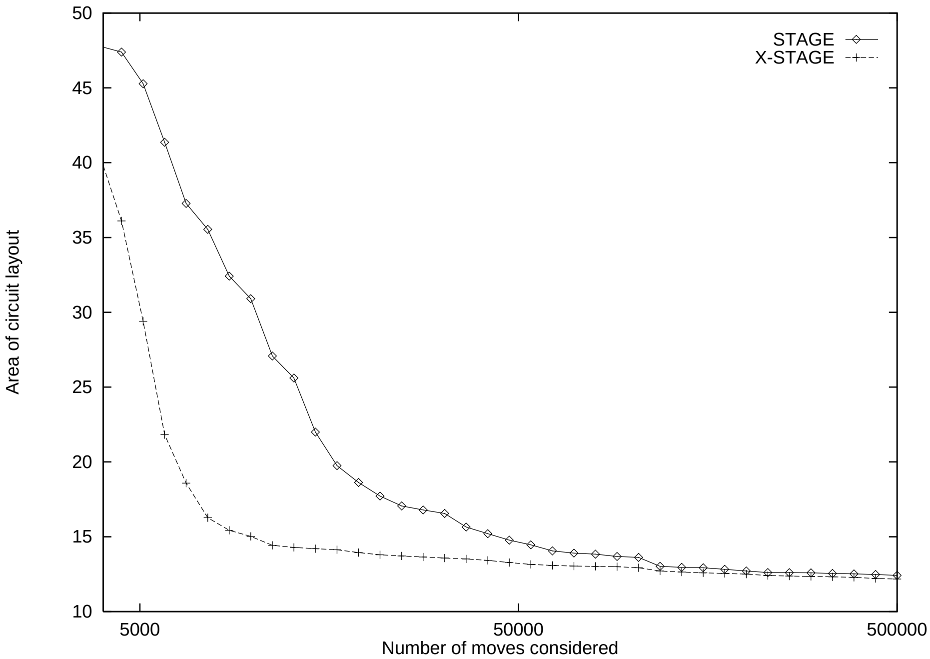


FIGURE 6.7. Channel routing performance on instance YK4 with transfer (X-STAGE) and without transfer (STAGE). Note the logarithmic scale of the x-axis.

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.

Chapter 7

Related Work

Heuristic methods for global optimization have assuredly *not* been overlooked in the literature. Their practical applications are important and numerous, and the literature is correspondingly immense. There is also a significant, if not quite as overwhelming, literature on learning evaluation functions for heuristic search in game-playing and problem-solving. In this chapter, I review the prior work most relevant to STAGE from both these literatures. The reviewed topics are organized as follows:

- §7.1: adaptive multi-restart techniques for local search;
- §7.2: reinforcement learning for combinatorial optimization, especially the study of Zhang and Dietterich [95];
- §7.3: simulation-based methods for improving AI search in game-playing and problem-solving domains, including techniques based on "rollouts" and on learning evaluation functions; and
- §7.4: genetic algorithms.

I defer a high-level discussion of STAGE's novel contributions to the next, concluding chapter.

7.1 Adaptive Multi-Restart Techniques

An iteration of hillclimbing typically reaches a local optimum very quickly. Thus, in the time required to perform a single iteration of (say) simulated annealing, one can run many hillclimbing iterations from different random starting points (or even from the same starting point, if move operators are sampled stochastically) and report the best result. Empirically, random multi-start hillclimbing has produced excellent solutions on practical computer vision tasks [Beveridge *et al.* 96], outperformed simulated annealing on the Traveling Salesman Problem (TSP) [Johnson and McGeoch 95], and outperformed genetic algorithms and genetic programming on several large-scale testbeds [Juels and Wattenberg 96].

Nevertheless, the effectiveness of random multi-start local search is limited in many cases by a "central limit catastrophe" [Boese *et al.* 94]: random local optima in large problems tend to all have average quality, with little variance [Martin and Otto 94]. This means the chance of finding an improved solution diminishes quickly from one iteration to the next. To improve on these chances, an *adaptive multi-start* approach—designed to select restart states with better-than-average odds of finding an improved solution—seems appropriate. Indeed, in the theoretical model of local search proposed by Aldous and Vazirani [94], a given performance level that takes $O(n)$ restarts to reach by a random starting policy can instead be reached with as few as $O(\log n)$ restarts when an adaptive policy, which uses successful early runs to seed later starting states, is used.

Many adaptive multi-start techniques have been proposed. One particularly relevant study has recently been conducted by Boese [95, 96]. On a fixed, well-known instance of the TSP, he ran local search 2500 times to produce 2500 locally optimal solutions. Then, for each of those solutions, he computed the average distance to the other 2499 solutions, measured by a natural distance metric on TSP tours. The results showed a stunning correlation between solution quality and average distance: high-quality local optima tended to have small average distance to the other optima—they were "centrally" located—while worse local optima tended to have greater average distance to the others; they were at the "outskirts" of the space. Similar correlations were found in a variety of other optimization domains, including circuit/graph partitioning, satisfiability, number partitioning, and job-shop scheduling. Boese concluded that many practical optimization problems exhibit a "globally convex" or so-called "big valley" structure, in which the set of local optima appears convex with one central global optimum. Boese's intuitive diagram of the big valley structure is reproduced in Figure 7.1.

The big valley structure is auspicious for a STAGE-like approach. Indeed, Boese's intuitive diagram, motivated by his experiments on large-scale complex problems, bears a striking resemblance to the 1-D wave function of Figure 3.4 (p. 50), which I contrived as an example of the kind of problem at which STAGE would excel. Working from the assumption of the big valley structure, Boese recommended the following two-phase adaptive multi-start methodology for optimization:

Phase One: Generate R random starting solutions and run Greedy_Descent from each to determine a set of corresponding *random local minima.*

Phase Two: Based on the local minima obtained so far, construct *adaptive* starting solutions and run Greedy_Descent A times from each one to yield corresponding *adaptive local minima.*

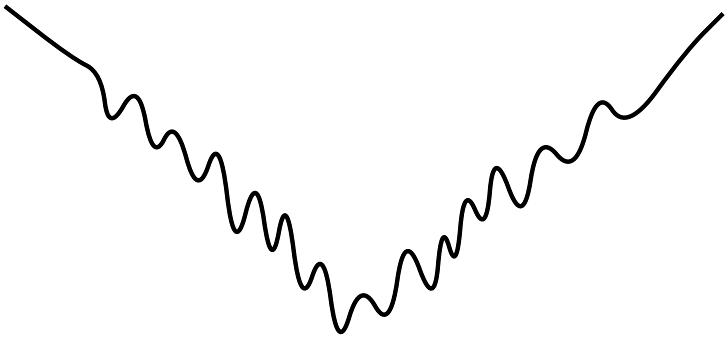


FIGURE 7.1. Intuitive picture of the "big valley" solution space structure. (Adapted from [Boese 95].)

> Intuitively, the two phases respectively develop, then exploit, a structural picture of the cost surface. [Boese *et al.* 94]

At this level of description, the two phases of Boese's recommended search regime correspond almost exactly with the two alternating phases of STAGE. The main difference is that Boese hand-builds a problem-specific routine for adaptively constructing new starting states, whereas STAGE uses machine learning to do the same automatically. Another difference is that STAGE's learned heuristic for constructing starting states is based on full hillclimbing trajectories, not just the local minima obtained so far.

A similar methodology underlies the current best heuristic for solving large Traveling Salesman problems, "Chained Local Optimization" (CLO) [Martin and Otto 94]. CLO performs ordinary hillclimbing to reach a local optimum z, and then applies a special large-step stochastic operator designed to "kick" the search from z into a nearby but different attracting basin. Hillclimbing from this new starting point produces a new local optimum z'; if this turns out to be much poorer than z, then CLO returns to z, undoing the kick. In effect, CLO constructs a new high-level search space: the new operators consist of large-step kick moves, and the new objective function is calculated by first applying hillclimbing in the low-level space, then evaluating the resulting local optimum. (A similar trick is often applied with genetic algorithms, as I discuss in Section 7.4 below.) In the TSP, the kick designed by Martin and Otto [94] is a so-called "double-bridge" operation, chosen because such moves cannot be easily found nor easily undone by Lin-Kernighan local search moves [Johnson and McGeoch 95]. Like Boese's adaptive multi-start, CLO relies on manually designed kick steps for finding a good new starting state, as opposed to STAGE's learned restart policy.

Furthermore, STAGE's "kicks" place search in not just a random nearby basin, but one specifically predicted to produce an improved local optimum.

The big valley diagram, like my 1-D wave function, conveys the notion of a global structure over the local optima. Unlike the wave function, it also conveys one potentially misleading intuition: that starting from low-cost solutions is necessarily better than starting from high-cost solutions. In his survey of local search techniques for the TSP, Johnson [95] considered four different randomized heuristics for constructing starting tours from which to begin local search. He found significant differences in the quality of final solutions. Interestingly, the heuristic that constructed the best-quality starting tours (namely, the "Clarke-Wright" heuristic) was also the one that led search to the *worst*-quality final solutions—even worse than starting from a very poor, completely random tour. Such "deceptiveness" can cause trouble for simulated annealing and genetic algorithms. Large-step methods such as CLO may evade some such deceits by "stepping over" high-cost regions. STAGE confronts the deceit head-on: it explicitly detects when features other than the objective function are better predictors of final solution quality, and can learn to ignore the objective function altogether when searching for a good start state.

Many other sensible heuristics for adaptive restarting have been shown effective in the literature. The widely applied methodology of "tabu search" [Glover and Laguna 93] is fundamentally a set of adaptive heuristics for escaping local optima, like CLO's kick steps. Hagen and Kahng's "Clustered Adaptive Multi-Start" achieves excellent results on the VLSI netlist partitioning task [Hagen and Kahng 97]; like CLO, it alternates between search with high-level operators (constructed *adaptively* by clustering elements of previous good solutions) and ordinary local search. Jagota's "Stochastic Steep Descent with Reinforcement Learning" heuristically rewards good starting states and punishes poor starting states in a multi-start hillclimbing context [Jagota *et al.* 96, Jagota 96]. The precise reward mechanism is heuristically determined and appears to be quite problem-specific, as opposed to STAGE's uniform mechanism of predicting search outcomes by value function approximation. As such, a direct empirical comparison would be difficult.

7.2 Reinforcement Learning for Optimization

Value function approximation has previously been applied to a large-scale combinatorial optimization task: the Space Shuttle Payload Processing domain [Zhang and Dietterich 95, Zhang 96]. I will discuss this application in some detail, because both the similarities and differences to STAGE are instructive. Please refer back to Section 2.1 for details on the algorithms and notation of value function approximation.

The Space Shuttle Payload Processing (SSPP) domain is a form 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 [94], Zhang and Dietterich defined a search over the space of fully specified schedules that 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 considered only operations which would repair the schedule's earliest constraint violation. Search terminates as soon as a violation-free schedule is reached.

Zhang and Dietterich applied reinforcement learning to this domain in order to obtain *transfer*: their goal 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 instance-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. (The voting-based approach to transfer introduced in Section 6.2.2 of this thesis allows such normalization to be avoided.) Trained on many scheduling instances, the neural network could then be applied to guide search on a novel scheduling instance, hopefully producing a good solution quickly.

Unlike STAGE, which seeks to learn V^π, the predicted outcome of a prespecified optimization policy π—the Zhang and Dietterich approach seeks to learn V^*, the predicted outcome of the *best possible* optimization policy. Before discussing how they approached this ambitious goal, we must address how the "best possible optimization policy" is even defined, since optimization policies face two conflicting objectives: "produce good solutions" and "finish quickly." In the SSPP domain, Zhang and Dietterich measured the total cost of a search trajectory $(x_0, x_1, \ldots, x_N, \text{END})$ by

$$\mathrm{Obj}(x_N) + 0.001N$$

Effectively, since $\mathrm{Obj}(x)$ is near 1.0 in this problem, this cost function means that a 1% improvement in final solution quality is worth about 10 extra search steps [Zhang and Dietterich 98]. The goal of learning, then, was to produce a policy π^* to optimize this balance between trajectory length and final solution quality.

Following Tesauro's methodology for learning V^* on backgammon, Zhang and Dietterich applied optimistic TD(λ) to the SSPP domain. Rather than training only a single neural network, they trained a pool of 8–12 networks, using different parameter settings for each. They trained the networks on a set of small problem instances. Training continued until performance stopped improving on a validation set of other problem instances. Then, the N best-performing networks were saved and used in a round-robin fashion for comparisons against Zweben's iterative-repair system [Zweben *et al.* 94], the previously best scheduler. The results showed that searches with the learned evaluation functions produced schedules as good as Zweben's in less than half the CPU time.

Getting these good results required substantial tuning. One complication involves state-space cycles. Since the move operators are deterministic, a learned policy may easily enter an infinite loop, which makes its value function undefined. Loops are fairly infrequent in the SSPP domain because most operators repair constraint violations, lengthening the schedule; still, Zhang and Dietterich had to include a loop-detection and escape mechanism, clouding the interpretation of V^*. To attack other combinatorial optimization domains with their method, they suggest that "it is important to formulate problem spaces so that they are acyclic" [Zhang and Dietterich 98]—but such formulations are unnatural for most local search applications, in which the operators typically allow any solution to be reached from any other solution. STAGE finesses this issue by fixing π to be a proper policy such as hillclimbing.

STAGE also manages to finesse three other algorithmic complications which Zhang and Dietterich found it necessary to introduce: experience replay [Lin 93], random exploration (slowly decreasing over time), and random-sample greedy search. Experience replay, i.e., saving the best trajectories in memory and occasionally retraining on them, is unnecessary in STAGE because the regression matrices always maintain the sufficient statistics of *all* historical training data. Adding random exploration is unnecessary because empirically, STAGE's baseline policy π (e.g., stochastic hillclimbing or WALKSAT) provides enough exploration inherently. This is in contrast to the SSPP formulation, where actions are deterministic. Finally, STAGE does not face the branching factor problem which led Zhang and Dietterich to introduce random-sample greedy search (RSGS). Briefly, the problem is that when hundreds or thousands of legal operators are available, selecting the greedy action, as optimistic TD(λ) requires, is too costly. RSGS uses a heuristic to select an approximately greedy move from a subset of the available moves. Again, this clouds the interpretation of V^*. In STAGE, each decision is simply whether to accept or reject a single available move, and the interpretation of V^π is clear.

To summarize, STAGE avoids most of the algorithmic complexities of Zhang and Dietterich's method because it is solving a fundamentally simpler problem: estimating V^π from a fixed stochastic π, rather than discovering an optimal deterministic policy π^* and value function V^*. It also avoids many issues of normalizing problem instances and designing training architectures by virtue of the fact that it applies in the context of a single problem instance. However, an advantage of the Zhang and Dietterich approach is that it holds out the potential of identifying a truly optimal or near-optimal policy π^*. STAGE can only claim to learn an improvement over the prespecified policy π.

Another published work, the "Ant-Q" system [Dorigo and Gambardella 95], is also billed as a reinforcement learning approach to combinatorial optimization. Based on an extensive metaphor with the behavior of ant colonies, it has been applied only to the TSP; it is unclear how it would be applied to other optimization domains.

7.3 Rollouts and Learning for AI Search

Ordinary hillclimbing, simulated annealing and genetic algorithms evaluate the quality of each neighboring state x' by its objective function value Obj(x'). STAGE evaluates x' by how promising its features make it appear, $\tilde{V}^\pi(F(x'))$. A third possibility is to evaluate x' by performing one or more actual sample runs of hillclimbing starting from x'. This is the principle of so-called "rollout algorithms" [Bertsekas *et al.* 97, Tesauro and Galperin 97], a term borrowed from the backgammon-strategy literature [Woolsey 91]. Like STAGE, rollout methods evaluate each neighbor based upon its long-term promise as a starting state for a fixed policy π. Selecting actions in this way is tantamount to performing a single round of the policy iteration algorithm [Howard 60] and can be guaranteed to improve upon π [Bertsekas *et al.* 97].

Unlike STAGE, rollout methods do not cache the results of their lookahead searches in a function approximator $\tilde{V}^\pi$. In this way they avoid the biases that feature representations and function approximation introduce, perhaps allowing more accurate moves. However, the computational cost of replacing function approximator evaluations by full sample runs is extremely high. This cost may be tolerable in a game-playing scenario where several seconds are available for each move selection; but would probably be deadly for practical optimization algorithms, where (in the words of Buntine [97]) a general maxim applies: "speed over smarts."

In fact, an earlier study of Abramson [90] demonstrated the power of rollout methods for game-playing, although he was apparently unaware of the connection to Markov decision processes and policy iteration. In his "expected-outcome" model, moves in the game of Othello were made by computing, via multiple rollouts for each legal move, the expected probability of winning *assuming both sides play randomly from that point on.* Though this assumption is clearly inaccurate, Abramson reported empirical good play at Othello:

> Given no expert information, the ability to evaluate only leaves, and a good deal of computation time, [expected-outcome functions] were able to play better than a competent (albeit non-masterful) function that had been handcrafted by an expert. [Abramson 90]

This improvement is consistent with the experience of Tesauro and Galperin [97], who found that a rollout-based player (implemented on a parallel IBM supercomputer) dramatically improved the performance of both weak and strong evaluation functions.

Abramson [90] went on to propose and test learning the expected-outcome function with linear regression over state features, just as STAGE does. Reinterpreted in the language of reinforcement learning, Abramson's method fixed a policy π (both players play randomly), inducing a Markov chain on the game space. He collected samples of the policy value function $V^\pi(x)$ by running π multiple times from each of 5000 random starting states; trained a linear predictor from the samples; and finally, used the predictor as a static evaluation function $\tilde{V}^\pi(F(x))$ to select moves. STAGE can be seen as extending Abramson's work from game-playing into the realm of combinatorial optimization. A key difference is that STAGE uses a high-quality baseline policy π, such as hillclimbing or WALKSAT, in place of Abramson's random policy. STAGE also interleaves its training and decision-making phases, which enables it to adapt its training distribution over time to focus on high-quality states.

Abramson's work is one of many examples in the Artificial Intelligence literature of learning evaluation functions for game-playing. Samuel's pioneering work on checkers [Samuel 59, Samuel 67], Christensen's work on chess [Christensen 86], and Lee's work on Othello [Lee and Mahajan 88] fall into this category; I have already discussed these in Section 2.1.2 of this thesis. More recently, the successful application of reinforcement learning to backgammon [Tesauro 92, Boyan 92] has inspired similar investigations in chess [Thrun 95, Baxter *et al.* 97], Go [Schraudolph *et al.* 94], and other games.

Moving from game-playing to problem-solving domains, a study by Rendell [83] addressed evaluation function learning in the sliding-tiles puzzle (15-puzzle). His method, when abstracted of many details, bears significant similarities to STAGE: it learns to approximate a function which measures the promise of each state as a starting state for a given policy. In particular, the policy π is a *best-first search* (with backtracking) guided by a given evaluation function $f(x)$; and the measurement being approximated is not a value function but a so-called "penetrance" measure at each state, as defined by Doran and Michie [66]. In STAGE-like notation, the penetrance is defined as

$$P^{\pi}(x) \stackrel{\text{def}}{=} \frac{\text{length of discovered path from } x \text{ to solution}}{\text{total nodes expanded by } \pi \text{ during search from } x}$$

For example, in the 15-puzzle, given a constant evaluation function $f(x) \equiv 1$, the policy π reduces to breadth-first search, and $P^{\pi}(x)$ is on the order of 10^{-9} for random starting states x; whereas a perfect evaluation function $f(x) \equiv V^{*}(x)$ would give rise to a backtracking-free policy π with penetrance $P^{\pi}(x) = 1$ everywhere. Rendell fits a linear approximation to the penetrance, then applies the fit to improve search control using a complex bootstrapping and normalization procedure.

7.4 Genetic Algorithms

Genetic algorithms (GAs)—algorithms based on metaphors of biological evolution such as natural selection, mutation, and recombination—represent another heuristic approach to combinatorial optimization [Goldberg 89]. Translated into the terminology of local search, "natural selection" means rejecting high-cost states in favor of low-cost states, like hillclimbing; "mutation" means a small-step local search operation; and "recombination" means adaptively creating a new state from previously good solutions. GAs have much in common with the adaptive multi-start hillclimbing approaches discussed above in Section 7.1. In broad terms, the GA population carries out multiple restarts of hillclimbing in parallel, culling poor-performing runs and replacing them with new adaptively constructed starting states.

To apply GAs to an optimization problem, the configuration space X must be represented as a space of discrete feature vectors—typically fixed-length bitstrings $\{0,1\}^{L}$—and the mapping must be a bijection, so that a solution bitstring in the feature space can be converted back to a configuration in X. (This contrasts to STAGE, where features can be any real-valued vector function of the state, and the mapping need not be invertible.) Typically, a GA mutation operator consists of flipping a single bit, and a recombination operator consists of merging the bits of two "parent" bitstrings into the new "child" bitstring. The effectiveness of GA search depends critically on the suitability of these operators to the particular bitstring representation chosen for the problem.

Of course, the effectiveness of *any* local search algorithm depends on the neighborhood operators available; but genetic algorithms generally allow less flexibility in designing the neighborhood, since mutations are represented as bit-flips. Hillclimbing and simulated annealing, by contrast, allow sophisticated, domain-specific search operators, such as the partition-graph manipulations used in simulated annealing applications of VLSI channel routing [Wong *et al.* 88]. On the other hand, genetic algorithms have a built-in mechanism for combining features of previously discovered good solutions into new starting states. STAGE can be seen as providing the best of both worlds: sophisticated search operators *and* adaptive restarts based on arbitrary domain features.

Some GA implementations do manage to take advantage of local search operators more sophisticated than bit-flips, using the trick of embedding a hillclimbing search into each objective function evaluation [Hinton and Nowlan 87]. That is, the GA's population of bitstrings actually serves as a population not of final solutions but of starting states for hillclimbing. The most successful GA approaches to the Traveling Salesman Problem all work this way so that they can exploit the sophisticated Lin-Kernighan local search moves [Johnson and McGeoch 95]. Here, the GA operators play a role analogous to the large-step "kick moves" of Chained Local Optimization [Martin and Otto 94], as described in Section 7.1 above. Depending on the particular implementation, the next generation's population may consist of not only the best starting states from the previous generation, but also the best final states found by hillclimbing runs—a kind of Lamarckian evolution in which learned traits are inheritable [Ackley and Littman 93, Johnson and McGeoch 95].

In such a GA, the population may be seen as implicitly maintaining a global predictive model of where, in bitstring-space, the best starting points are to be found. The COMIT algorithm of Baluja and Davies [97], a descendant of PBIL [Baluja and Caruana 95] and MIMIC [de Bonet *et al.* 97], makes this viewpoint explicit: it generates adaptive starting points not by random genetic recombination, but rather by first building an explicit probabilistic model of the population and then sampling that model. COMIT's learned probability model is similar in spirit to STAGE's V^π function. Differences include the following:

- COMIT is restricted to bijective bitstring-like representations, whereas STAGE can use any feature mapping; and

- COMIT's model is trained from only the set of best-quality states found so far, ignoring the differences between their outcomes; whereas STAGE's value function is trained from all states seen on all trajectories, good and bad, paying attention to the outcome values. Boese's experimental data and "big valley structure" hypothesis (see page 165) indicate that there is often useful information to be gained by modelling the weaker areas of the solution space, too [Boese *et al.* 94]. In particular, this gives STAGE the power for directed extrapolation beyond the support of its training set.

In preliminary experiments in the Boolean satisfiability domain, on the same 32-bit parity instances described in Section 4.7, COMIT (using WALKSAT as a subroutine) did not perform as well as STAGE [Davies and Baluja 98].

7.5 Discussion

The studies described in the last four sections make it clear that STAGE bears close relationships to previous work done in the optimization, reinforcement learning, and AI problem-solving communities. A concise summary of this chapter might read as follows:

> STAGE takes its main idea—learn an evaluation function to improve search performance—from the AI literature on problem-solving. It grounds that idea in the theory of value function approximation. And it applies that idea in the successful framework of adaptive multi-start approaches to global optimization.

STAGE unifies these lines of research in an algorithm which is nonetheless quite simple to explain and to implement. In the next, concluding chapter, I will summarize the novel contributions made by STAGE and suggest a number of future directions for integrating reinforcement learning into effective optimization algorithms.

Chapter 8

Conclusions

Reinforcement learning offers the tantalizing promise of software systems that autonomously improve their performance on sequential decision-making tasks, according to the following methodology:

1. Collect data by observing simulations of the task.

2. Statistically learn to predict the long-term outcomes of the chosen decisions.

3. Use those predictions as an evaluation function to guide future decisions with great foresight.

In this dissertation, I have described and analyzed a practical method for applying this methodology to general combinatorial optimization tasks. I have also introduced several new algorithms for efficient reinforcement learning in large state spaces.

This concluding chapter first reviews the dissertation's scientific contributions to the state of the art in reinforcement learning, heuristic search, and global optimization. It then outlines a number of promising directions for future research in learning evaluation functions.

8.1 Contributions

The principal contributions of this thesis may be summarized as follows:

- (Chapter 2) I have introduced **two novel value-function-approximation algorithms**: Grow-Support for deterministic problems (§2.2) and ROUT for stochastic acyclic problems (§2.3). Their primary innovation is that, without falling prey to the curse of dimensionality, they are able to explicitly represent which states are already solved and which are not yet solved. Using this information, they "work backwards," computing accurate V^* values at targeted unsolved states. By treating solved and unsolved states differently, Grow-Support and ROUT eliminate the possibility of divergence caused by repeated value re-estimation.

- (§3.1.3) In the context of global optimization, I have recognized that **additional features** of each state, other than the state's objective-function value, can provide useful information for decision making in search. Traditional algorithms either ignore this information or incorporate it in an *ad hoc* manner. STAGE provides a principled, automatic mechanism for exploiting additional state features.

- (§3.2) I have defined the predictive **value function of a local search procedure**, $V^\pi(x)$; described the conditions under which it corresponds to the value function of a Markov chain; and described how it may be learned from simulation data by a function approximator.

- (§3.2) I have introduced **STAGE**, a straightforward algorithm for exploiting the learned approximation of V^π to guide future search. STAGE is general: it can be applied to any optimization problem to which hillclimbing applies. It may be viewed as an adaptive multi-restart approach to optimization; the adaptive component is automatically learned from simulation data on each problem instance.

- (§3.3) I have presented **two illustrative domains**—the 1-D wave minimization example and the bin-packing example—and demonstrated how STAGE succeeds on them. These examples provide clear intuitions of how learning evaluation functions can improve search performance.

- (§3.4) I have analyzed the **theoretical conditions** under which STAGE is well-defined and efficient. Specifically, I have shown that V^π is well-defined for *any* local search procedure π, as long as the objective function is bounded below; however, STAGE learns to approximate V^π most efficiently if π is proper, Markovian, and monotonic. I have provided several methods for converting improper and nonmonotonic procedures into a form suitable for STAGE.

- (Chapters 4 and 5) I have contributed **empirical evidence** that STAGE is applicable, practical and effective on a wide variety of large-scale optimization tasks. On most tested instances, STAGE outperforms both multi-start hillclimbing and a good implementation of simulated annealing. On challenging instances of the Boolean satisfiability task, STAGE successfully learned from WALKSAT, a non-greedy local search procedure—and produced the best published solutions to date. The empirical analyses of Chapter 5 demonstrate STAGE's overall robustness with various parameter settings, and also explain under what circumstances STAGE will fail to improve optimization performance.

- (§6.1) I have introduced a **least-squares formulation of the TD($\boldsymbol{\lambda}$) algorithm**, extending the work of Bradtke and Barto [96]. I empirically demonstrate the improved data efficiency of this formulation, and give a new intuitive explanation for the source of this efficiency: the statistics kept by LSTD(λ) amount to a compressed *model* of the underlying Markov process.

- (§6.2) I have motivated, described and tested a new approach to **transferring learned information** from previously solved optimization problem instances to new ones. By using a voting mechanism, the X-STAGE algorithm avoids having to normalize the objective function across disparate instances.

- (Chapter 7) I have **surveyed the literatures** of several related areas: value function approximation (§2.1.2, §7.2); adaptive multi-restart techniques for local search (§7.1), including genetic algorithms (§7.4); and simulation-based learning methods for improving AI search (§7.3). Taken together, these surveys provide a useful collection of references for researchers interested in automatic learning and tuning of evaluation functions.

8.2 Future Directions

The main conclusion of this thesis is that learning evaluation functions can improve global optimization performance. STAGE is a simple, practical technique that demonstrates this. STAGE's simplicity enables many potentially useful extensions; I suggest some of these in Section 8.2.1. Beyond STAGE, there are at least two conceptually different ways of utilizing value function approximation in global optimization; I discuss these in Section 8.2.2. Finally, in Section 8.2.3, I consider the potential for learning evaluation functions by non-VFA-based, *direct meta-optimization* methods.

8.2.1 Extending STAGE

Many modifications to and extensions of STAGE, such as varying the regression model and training technique used to approximate V^π, have been investigated in Chapters 5 and 6 of this thesis. However, many further interesting modifications remain untried. I describe several of these here:

Non-polynomial function approximators. My study of Section 5.2.2 was limited to first- through fifth-order polynomial models of V^π. It would be interesting to see whether other linear architectures—such as CMACs, radial basis function networks, and random-representation neural networks—could produce better fits and better performance.

A more ambitious study could investigate efficient ways to use *nonlinear architectures*, such as multi-layer perceptrons or memory-based fitters, with STAGE. In the context of transfer, the training speed of the function approximator is less crucial. One intriguing possibility is to learn a nonlinear representation from a set of training instances, then freeze the nonlinear components so that fast least-squares methods can be used on the test instances. For example, STAGE could learn a neural network representation of V^π from training instances, then freeze the input-to-hidden weights of the network to allow linear learning on a new test instance. This could be a useful way to construct a feature set for a linear architecture automatically. (Related ideas are discussed in [Utgoff 96].)

More aggressive optimization of $\tilde{\mathbf{V}}^\pi$. On each iteration, in order to find a promising new starting state for the baseline procedure π, STAGE optimizes $\tilde{V}^\pi$ by performing first-improvement hillclimbing. A more aggressive optimization technique, such as simulated annealing, could instead be applied at that stage; and that may well improve performance.

Steepest descent. With the exception of the WALKSAT results of Section 4.7 and the experiments of Section 5.2.3, STAGE has been trained to predict and improve upon the baseline procedure of π = first-improvement hillclimbing. However, in some optimization problems—particularly, those with relatively few moves available from each state—*steepest-descent* (best-improvement) search may be more effective. Steepest-descent is proper, Markovian, and monotonic, so STAGE applies directly; and it would be interesting to compare its effectiveness with first-improvement hillclimbing's.

Continuous optimization. This dissertation has focused on discrete global optimization problems. However, STAGE applies without modification to continuous global optimization problems (i.e., find $\mathbf{x}^* = \operatorname{argmin} \mathrm{Obj} : \Re^K \to \Re$) as well. The cartogram design problem of Section 4.6 is an example of such a problem; however, much more sophisticated neighborhood operators than the point perturbations I defined for that domain are available. For example, the downhill simplex method of Nelder and Mead (described in [Press *et al.* 92, §10.4]) provides an effective set of local search moves for continuous optimization. Downhill simplex reaches a local optimum quickly, and Press et al. [92] recommend embedding it within a multiple-restart or simulated-annealing framework. STAGE could provide an effective learning framework for multi-restart simplex search.

Confidence intervals. STAGE identifies good restart points by optimizing $\tilde{V}^\pi(x)$, the predicted expected outcome of search from x. However, in the context of a long run involving many restarts, it may be better to start search from a state with worse expected outcome but higher outcome *variance*. After all, what we really want to minimize is not the outcome of any one trajectory, but the minimum outcome over the whole collection of trajectories STAGE generates. One possible heuristic along these lines would be to exploit confidence intervals on $\tilde{V}^\pi$'s predictions to guide search. For example, STAGE could evaluate the promise of a state x by, instead of the expected value of $\tilde{V}^\pi(x)$, a more *optimistic* measure such as

- the 25th-percentile prediction of $\tilde{V}^\pi(x)$, or
- the probability that $\tilde{V}^\pi(x)$ exceeds the best value seen so far on this run.

Such strategies could have the effect of both encouraging exploration of state-space regions where $\tilde{V}^\pi$ is poorly modeled (similar to the Interval-Estimation [Kaelbling 93] and IEMAX [Moore and Schneider 96] algorithms) and encouraging repeated visits to states that promise to lead occasionally to excellent solutions.

8.2.2 Other Uses of VFA for Optimization

STAGE exploits the value function V^π for the purpose of guiding search to new starting points for π. However, value functions can also aid optimization in at least two further ways: *filtering* and *sampling*.

Filtering refers to the early cutoff of an unpromising search trajectory—before it even reaches a local optimum—to conserve time for additional restarts and better trajectories. Heuristic methods for filtering have been investigated by, e.g., [Nakakuki and Sadeh 94]. Perkins et al. [97] have suggested that reinforcement-learning methods could provide a principled mechanism for deciding when to abort a trajectory. In the context of STAGE, filtering could be implemented simply as follows: cut off any π-trajectory when its predicted eventual outcome $\tilde{V}^\pi(x)$ is worse than, say, the mean of all π-outcomes seen thus far. This technique would allow STAGE to exploit its learned predictions during both stages of search.

Sampling refers to the selection of candidate moves for evaluation during search. In this dissertation, I have assumed that candidate moves are generated with a probability distribution that remains stationary throughout the optimization run. In optimization practice, however, it is often more effective to modify the candidate distribution over the course of the search—for example, to generate large-step candidate moves more frequently early in the search process, and to generate small-step, fine-tuning moves more frequently later in search. Cohn reviews techniques for adapting the sampling distribution of candidate moves, including one "based on their probability of success and on their effect on improving the cost function" [Cohn 92, §2.4.4]. In order to estimate these quantities without having to invoke the (presumably expensive) objective function, Cohn's move generator maintains statistics for each category of move that has been tried recently in the run—a simple kind of reinforcement learning.

A more sophisticated approach has recently been proposed by Su et al. [98]. Their method learns, over multiple simulated-annealing runs, to predict the long-term outcome achieved by starting search at state $\mathbf{x}$ and with initial action a.[1] In reinforcement-learning terminology, their method learns to approximate the task's *state-action value function* $Q^\pi(\mathbf{x}, a)$ [Watkins 89]. This form of value function allows the effects of various actions a to be predicted without having to actually apply the action or invoke the objective function. Their method uses the learned value function to preselect the most promising out of five random candidate moves before each step of simulated annealing, thereby saving time that would have been spent evaluating bad candidate moves. In optimization domains where objective function evaluations are costly, the Q^π value-function formulation offers the potential for significant speedups. It remains for future research to determine how best to combine filtering, sampling, and search-guiding uses of value functions in optimization.

8.2.3 Direct Meta-Optimization

All the approaches discussed in this thesis have built evaluation functions by approximating a value function V^π or V^*, functions which predict the long-term outcomes of a search policy. However, an alternative approach not based on value function approximation, which I call *direct meta-optimization*, also applies. Direct meta-optimization methods assume a fixed parametric form for the evaluation function and optimize those parameters directly with respect to the ultimate objective, sampled by Monte

[1] Note that the state vector $\mathbf{x}$ for simulated annealing consists of both the current configuration x and the current temperature T.

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 evaluation functions $\tilde{V}$ learned by such methods are not constrained by the Bellman equations: the values they produce for any given state have no semantic interpretation in terms of long-term predictions. 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. However, there are several reasons to believe that a direct approach may be effective:

- Not having to meet the Bellman constraints may actually make learning easier. For example, even if a domain's true value function is very jagged, meta-optimization may discover a quite different, smooth $\tilde{V}$ that performs well. (This point was also made in [Utgoff and Clouse 91].)

- Direct approaches do not depend on the Markov property. Since they treat the baseline search procedure as a black box, they can be applied to optimize the evaluation function for backtracking search algorithms such as A*, or in sequential decision problems involving hidden state.

- Similarly, extra algorithmic parameters such as hillclimbing's *patience* level and simulated annealing's temperature schedule can be included along with the evaluation function coefficients in the meta-optimization.

Further arguments supporting the direct approach are given in [Moriarty *et al.* 97].

Direct meta-optimization methods have been applied to learning evaluation functions before, particularly in the game-playing literature. Genetic-algorithm approaches to game learning generally fall into this category (e.g., [Tunstall-Pedoe 91]). Recently, Pollack et al. attacked backgammon by hillclimbing over the 3980 weights of a neural network [Pollack *et al.* 96]. Surprisingly, this procedure developed a good backgammon player, though not on the level of Tesauro's TD-Gammon networks. 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 94]. These and 10 other parameters of the annealer were optimized using Powell's method as described in [Press *et al.* 92]. 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.

I believe that the computational requirements of direct meta-optimization can be significantly reduced by the use of new memory-based stochastic optimization techniques [Moore and Schneider 96, Moore *et al.* 98]. 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 the baseline stochastic search procedure and reporting the final result. An important future direction for reinforcement-learning research is to carefully compare the empirical performance of direct meta-optimization and value function approximation methods.

8.3 Concluding Remarks

In the decade since the deep connection between AI heuristic search and Markov decision process theory was first identified, the field of reinforcement learning has made much progress. Algorithms for learning sequential decision-making from simulation data are now well understood for tasks in which the value function can be represented exactly. This thesis has addressed the more difficult case in which the value function must be represented compactly by a function approximator. Its primary contribution is a practical algorithm, built on reinforcement-learning foundations, that efficiently learns and exploits a predictive model of a search procedure's performance. Ultimately, I believe such methods will lead to more effective solutions to large sequential decision problems in industry, science, and government, and thereby improve society.

Appendix A

Proofs

A.1 The Best-So-Far Procedure Is Markovian

Here, I prove Proposition 2 of Section 3.4.1. This proposition gives us a way to apply a natural patience-based termination criterion to a nonmonotonic search procedure such as WALKSAT—yet still maintain the Markov property that makes the target of STAGE's learning well-defined—by using the device of the *best-so-far abstraction* $\text{BSF}(\pi)$, as defined in Definition 6 on page 63. The statement is as follows:

Proposition (**#2**, p. 64). *If local search procedure π is Markovian over a finite state space X, and π' is the procedure that results by adding patience-based termination to π, then procedure $\text{BSF}(\pi')$ is proper, Markovian, and strictly monotonic.*

In outline, the proof follows these three steps:

1. The procedure π' is proper, but not necessarily Markovian in the state space X. However, π' is proper and Markovian in an augmented state space Z.
2. We apply a lemma that the Y*-abstraction* (defined below) of any proper and Markovian procedure is also proper and Markovian. $\text{BSF}(\pi')$ is such an abstraction, which proves that it is proper and Markovian in the augmented state space Z.
3. Finally, we show that the Markov property still holds when trajectories of $\text{BSF}(\pi')$ are projected back down to the original state space X. The property of strict monotonicity also follows trivially.

Before explaining these steps in detail, I present the definition of a Y-abstraction and the lemma that Step 2 requires.

Definition 7. Let π be a proper local search procedure defined over a state space $Z \cup \{\text{END}\}$, and let $Y \subset Z$ be an arbitrary subset of states. Then the Y*-abstraction* of procedure π is a new policy π', defined over the smaller state space $Y \cup \{\text{END}\}$, which is derived from π as follows: starting from any state $y_0 \in Y$, π' generates trajectories by following π *but filtering out all states belonging to* $Z \setminus Y$ (see Figure A.1).

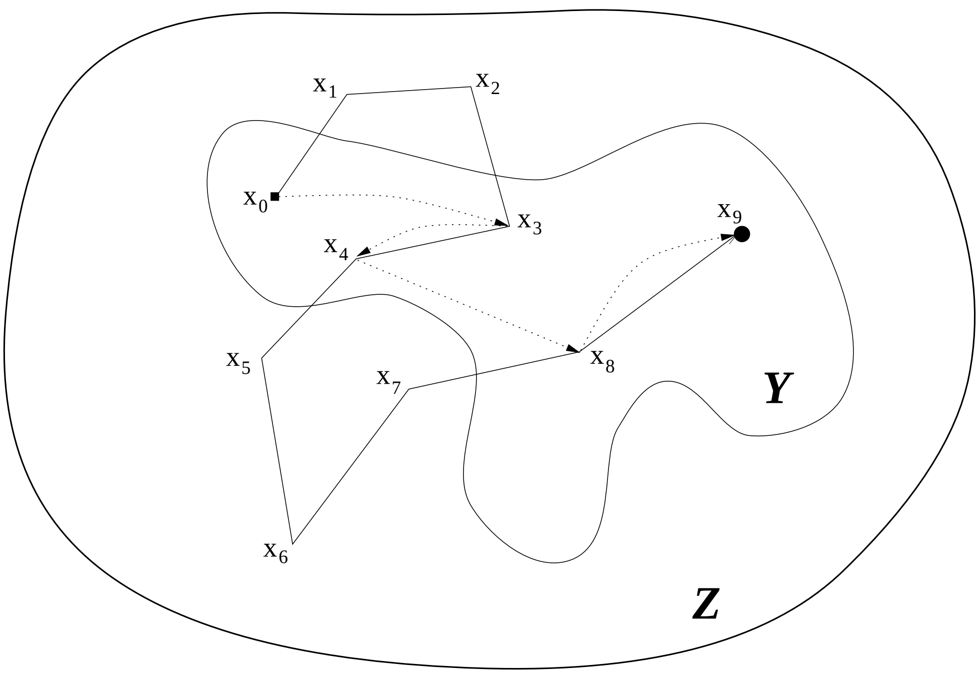


FIGURE A.1. Illustration of the Y-abstraction procedure. Given a trajectory $(x_0, \ldots, x_9)$ (straight solid lines) generated by procedure π in state space Z, the Y-abstraction produces a corresponding shorter trajectory (dotted lines) that is restricted to the subspace Y.

Lemma 1. *Suppose procedure π is proper and Markovian over state space $Z \cup \{\text{END}\}$, and that procedure π' is the Y-abstraction of Z for any given subset $Y \subset Z$. Then π' is proper and Markovian over the subspace $Y' = Y \cup \{\text{END}\}$.*

Proof of Lemma 1. First, note that if we apply π starting from any state $x_0 \in Z$, we get a trajectory τ which must eventually visit a state $y \in Y'$, since π is assumed proper and Y' includes the terminal state. Let $\text{ENTER}(\tau, Y')$ be the first state in Y' that τ visits after leaving x_0. Then, over the full distribution of trajectories that π may generate from x_0, the probability that y is the first state subsequently visited in Y' is given by $p(y|x_0)$, computed as follows:

$$\forall x_0 \in Z, y \in Y': \quad p(y|x_0) \stackrel{\text{def}}{=} \sum_{\{\tau \in \mathcal{T}:\text{ENTER}(\tau, Y')=y\}} P(\tau | x_0^\tau = x_0) \tag{A.1}$$

Now consider a trajectory of the Y-abstraction policy π', starting at a state $y_0 \in Y$. By definition of π', this trajectory will be built by applying the original policy π and filtering out the states in $Z \backslash Y$. Thus, the first transition will be to some state $y_1 \in Y'$

with probability given precisely by $p(y_1|y_0)$ as defined in Equation A.1. Furthermore, by the Markov assumption on π, the future transitions from y_1 will be independent of the trajectory history and obey the same probabilities $p(y_{i+1}|y_i)\ \forall i \geq 0$. Thus, π' is Markovian over Y'. π' is also clearly proper, by inheritance from π. □

I now fill in the details of the proof of Proposition 2, following the three-part outline sketched above.

Proof of Proposition 2.

1. In the statement of the proposition, π' is defined to be the procedure that results when patience-based termination is imposed on a procedure π that is Markovian, but not necessarily proper. Recall that patience-based termination means, for a given patience level $\textsc{Pat} \geq 1$, that any trajectory will end deterministically as soon as $\textsc{Pat}$ consecutive steps are taken with no improvement over the best state found so far. That is, for all trajectories τ produced by π':

$$(t \geq \textsc{Pat}) \wedge (x^\tau_{t-\textsc{Pat}} = \min_{\{j:0\leq j\leq t\}} \mathrm{Obj}(x^\tau_j)) \;\Rightarrow\; x^\tau_{t+1} = \textsc{end}.$$

Since the state space X is assumed finite, π' is certainly proper: the Obj values cannot keep decreasing forever. However, π' is not necessarily Markovian. Because of the new termination condition, the transition probability

$$\mathrm{P}(x^\tau_{i+1} = x_{i+1} \mid x^\tau_0 = x_0, x^\tau_1 = x_1, \ldots, x^\tau_i = x_i)$$

is *not* independent of the history $x_0, \ldots, x_{i-1}$ as required by Definition 3 (page 59). In particular, the probability of termination depends not just on the current state, but also on how recently the best-so-far state was found, and what the Obj value at that state was.

However, the procedure π' can be made Markovian by augmenting the state space with these relevant extra variables. Define the augmented state space $Z = X \times \Re \times \mathbb{N}$, where the three components of a state $z \in Z$ are given by

$a(z) \in X$, representing a state x in the original state space

$b(z) \in \Re$, representing the best Obj value seen on the current trajectory

$c(z) \in \mathbb{N}$, representing the count of non-improving steps since finding $b(z)$.

In this augmented space, a trajectory of procedure π' begins at the state $z^\tau_0 = \langle x^\tau_0, \mathrm{Obj}(x^\tau_0), 0\rangle$. All future state transitions depend only on the current state

z_i^τ. There are three kinds of possible transitions: (1) the patience counter may expire, causing deterministic termination; (2) a new best-so-far state may be discovered, in which case $b(z)$ is updated and $c(z)$ is reset to zero; or (3) an ordinary transition to a non-best-so-far state occurs, in which case $c(z)$ is simply incremented. These give rise to the following Markovian transition probabilities: $\forall z \in Z, z' \in Z \cup \{\text{END}\}$,

$$\mathrm{P}(z_{i+1}^\tau = z' \mid z_i^\tau = z) =$$
$$\begin{cases} 1 & \text{if } (c(z) = \text{PAT}) \wedge (z' = \text{END}) \\ p(a(z')|a(z)) & \text{if } (\text{Obj}(a(z')) < b(z)) \wedge (b(z') = \text{Obj}(a(z'))) \wedge (c(z') = 0) \\ p(a(z')|a(z)) & \text{if } (\text{Obj}(a(z')) \geq b(z)) \wedge (b(z') = b(z)) \wedge (c(z') = c(z) + 1) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, π' is proper and Markovian over the space $Z \cup \{\text{END}\}$, concluding the first segment of our proof.

2. We now apply Lemma 1 to show that the best-so-far abstraction of π', denoted $\text{BSF}(\pi')$, is also proper and Markovian in the augmented state space Z. Let the subset Y be defined by

$$Y = \{z \in Z : (b(z) = \text{Obj}(a(z))) \wedge (c(z) = 0)\}.$$

Y consists of precisely the best-so-far states on any trajectory of π'. By Lemma 1, $\text{BSF}(\pi')$ is proper and Markovian over the subspace $Y \cup \{\text{END}\}$. That is, following procedure $\text{BSF}(\pi')$, the transitions from any state y are given by fixed probabilities $p(y'|y)$ independent of any past history of the procedure.

3. Finally, we note that every state $y \in Y$ has the form $\langle x, \text{Obj}(x), 0\rangle$. All the information about the state y is determined by the first component, the underlying state in the un-augmented space. It follows that the procedure $\text{BSF}(\pi')$ is Markovian in the underlying space $X \cup \{\text{END}\}$, with transition probabilities given simply by

$$\mathrm{P}(x_{i+1}^\tau = x' \mid x_i^\tau = x) = p(\langle x', \text{Obj}(x'), 0\rangle \mid \langle x, \text{Obj}(x), 0\rangle).$$

We note that the property of strict monotonicity follows trivially from the definition of BSF, completing the proof of Proposition 2. □

A.2 Least-Squares TD(1) Is Equivalent to Linear Regression

This section demonstrates that the incremental algorithm LSTD(1), which is a special case of the LSTD(λ) procedure introduced in Section 6.1.2, produces an approximate value function which is equivalent to that which would be generated by standard, non-incremental, least-squares linear regression.

To be precise, assume we are given a sample trajectory $(x_0, x_1, \ldots, x_L, \text{END})$ of a Markov chain, with one-step rewards of $R(x_j, x_{j+1})$ on each step. From this trajectory, a supervised learning system would generate the training pairs:

$$\begin{aligned}
\boldsymbol{\phi}_0 &\mapsto R(x_0, x_1) + R(x_1, x_2) + \cdots + R(x_L, \text{END}) \\
\boldsymbol{\phi}_1 &\mapsto R(x_1, x_2) + \cdots + R(x_L, \text{END}) \\
\vdots &\mapsto \vdots \\
\boldsymbol{\phi}_L &\mapsto R(x_L, \text{END})
\end{aligned}$$

where $\boldsymbol{\phi}_i$ is the vector of features representing state x_i. Performing standard least-squares linear regression on the above training set, as described for STAGE in Chapter 3 (Equation 3.8, page 69), produces the following regression matrices:

$$\mathbf{A}_{\text{LR}} = \sum_{i=0}^{L} \boldsymbol{\phi}_i \boldsymbol{\phi}_i^{\mathsf{T}} \qquad\qquad \mathbf{b}_{\text{LR}} = \sum_{i=0}^{L} \boldsymbol{\phi}_i y_i$$

$$\text{where } \; y_i = \sum_{j=i}^{L} R(x_j, x_{j+1})$$

I now show that, thanks to the algebraic trick of the eligibility vectors $\mathbf{z}_t$, LSTD(1) builds the equivalent $\mathbf{A}$ and $\mathbf{b}$ fully incrementally—without having to store the trajectory while waiting to observe the eventual outcome y_i. Please refer to Table 6.1.1 (p. 142) for the definition of the algorithm.

Proof. With simple algebraic manipulations, the sums built by LSTD(1)'s $\mathbf{A}$ and $\mathbf{b}$ telescope neatly into $\mathbf{A}_{\text{LR}}$ and $\mathbf{b}_{\text{LR}}$, as follows:

$$\begin{aligned}
\mathbf{A} &= \sum_{i=0}^{L} \mathbf{z}_t \big(\boldsymbol{\phi}_i - \boldsymbol{\phi}_{i+1}\big)^{\mathsf{T}} \\
&= \sum_{i=0}^{L} \Big(\sum_{j=0}^{i} \boldsymbol{\phi}_j\Big)\big(\boldsymbol{\phi}_i - \boldsymbol{\phi}_{i+1}\big)^{\mathsf{T}} && \textit{(by definition of } \mathbf{z}_t\textit{)} \\
&= \sum_{i=0}^{L} \sum_{j=0}^{i} \boldsymbol{\phi}_j \boldsymbol{\phi}_i^{\mathsf{T}} - \sum_{i=0}^{L} \sum_{j=0}^{i} \boldsymbol{\phi}_j \boldsymbol{\phi}_{i+1}^{\mathsf{T}}
\end{aligned}$$

$$
\begin{aligned}
&= \big(\boldsymbol{\phi}_0\boldsymbol{\phi}_0^\intercal + \sum_{i=1}^{L}\sum_{j=0}^{i}\boldsymbol{\phi}_j\boldsymbol{\phi}_i^\intercal\big) - \big(\sum_{k=1}^{L+1}\sum_{j=0}^{k-1}\boldsymbol{\phi}_j\boldsymbol{\phi}_k^\intercal\big) && \textit{(substituting } k \equiv i+1\textit{)} \\
&= \boldsymbol{\phi}_0\boldsymbol{\phi}_0^\intercal + \sum_{i=1}^{L}\sum_{j=0}^{i}\boldsymbol{\phi}_j\boldsymbol{\phi}_i^\intercal - \sum_{k=1}^{L}\sum_{j=0}^{k-1}\boldsymbol{\phi}_j\boldsymbol{\phi}_k^\intercal && \textit{(since } \boldsymbol{\phi}_{L+1} \stackrel{\text{def}}{=} \boldsymbol{0}\textit{)} \\
&= \boldsymbol{\phi}_0\boldsymbol{\phi}_0^\intercal + \sum_{i=1}^{L}\big(\sum_{j=0}^{i}\boldsymbol{\phi}_j\boldsymbol{\phi}_i^\intercal - \sum_{j=0}^{i-1}\boldsymbol{\phi}_j\boldsymbol{\phi}_i^\intercal\big) && \textit{(substituting } i \equiv k\textit{)} \\
&= \sum_{i=0}^{L}\boldsymbol{\phi}_i\boldsymbol{\phi}_i^\intercal \\
&= \mathbf{A}_{\text{LR}}, \quad \text{as desired;}
\end{aligned}
$$

and

$$
\begin{aligned}
\mathbf{b} &= \sum_{i=0}^{L}\mathbf{z}_i R(x_i, x_{i+1}) \\
&= \sum_{i=0}^{L}\big(\sum_{j=0}^{i}\boldsymbol{\phi}_j\big)R(x_i, x_{i+1}) && \textit{(by definition of } \mathbf{z}_t\textit{)} \\
&= \sum_{i=0}^{L}\sum_{j=0}^{L}1(j \le i)\boldsymbol{\phi}_j R(x_i, x_{i+1}) && \textit{(where } 1(\textsf{True}) \stackrel{\text{def}}{=} 1,\ 1(\textsf{False}) \stackrel{\text{def}}{=} 0\textit{)} \\
&= \sum_{j=0}^{L}\sum_{i=0}^{L}1(j \le i)\boldsymbol{\phi}_j R(x_i, x_{i+1}) \\
&= \sum_{j=0}^{L}\boldsymbol{\phi}_j\sum_{i=j}^{L}R(x_i, x_{i+1}) \\
&= \mathbf{b}_{\text{LR}}, \quad \text{as desired.}
\end{aligned}
$$

These reductions prove that the contributions to $\mathbf{A}$ and $\mathbf{b}$ by any single trajectory are identical in LSTD(1) and least-squares linear regression. In both algorithms, contributions from multiple trajectories are simply summed into the matrices. Thus, LSTD(1) and linear regression compute the same statistics and, ultimately, the same coefficients for the approximated value function. □

Appendix B

Simulated Annealing

B.1 Annealing Schedules

Simulated annealing (SA) is described by the template in Table B.1. The main implementation challenge is to choose an effective annealing schedule for the temperature parameter. The temperature t_i controls the probability of accepting a step to a worse neighbor: when $t_i = +\infty$, SA accepts any step, acting as a random walk; whereas when $t_i = 0$, SA rejects all worsening steps, acting as stochastic hillclimbing with equi-cost moves.

Simulated-Annealing(X, S, N, Obj, TotEvals, *Schedule*):

Given:

- a state space X
- starting states $S \subset X$
- a neighborhood structure $N : X \to 2^X$
- an objective function, $\mathrm{Obj} : X \to \Re$, to be minimized
- TotEvals, the number of state evaluations allotted for this run
- an annealing *Schedule* which determines the temperature on each iteration

1. Let $x_0 \in S$ be a random starting state for search;
   let $t_0 :=$ the initial temperature of *Schedule*
2. For $i := 0$ to $\mathrm{TotEvals} - 1$, do:
   (a) Choose $x' :=$ a random element from $N(x_i)$
   (b) $x_{i+1} := \begin{cases} x' & \text{if } \mathrm{RAND}[0,1) < e^{[\mathrm{Obj}(x_i) - \mathrm{Obj}(x')]/t_i} \\ x_i & \text{otherwise} \end{cases}$
   (c) Update temperature t_{i+1} according to *Schedule*
3. Return the best state found.

Table B.1. A template for the simulated annealing algorithm

The temperature annealing schedule has been the subject of extensive theoretical and experimental analysis in the simulated annealing literature [Boese 96]. The possibilities include the following:

- Logarithmic: $t_i = \gamma / \log(i+2)$. For sufficiently high γ, a schedule of this form guarantees that SA will converge with probability one to the globally optimal solution [Mitra *et al.* 86, Hajek 88]. Unfortunately, this guarantee only applies in the limit as TotEvals $\to \infty$. Logarithmic schedules are generally too slow for practical use.

- Geometric: for a fixed *initial temperature* t_0, *cooling rate* $\alpha \in (0,1)$ and *round length* $L \geq 1$, define $t_i = t_0 \cdot \alpha^{\lfloor i/L \rfloor}$. This is the original schedule proposed by Kirkpatrick [83], and is still widely used in practice [Johnson *et al.* 89, Johnson *et al.* 91, Press *et al.* 92]. However, the parameters t_0, L, and α must be tuned for each problem instance.

- Adaptive: a variety of adaptive schedules have been proposed, with both theoretical and practical motivations. [Boese 96] summarizes the most notable of these, including the schedules of [Aarts and van Laarhoven 85], [Huang *et al.* 86], and [Lam and Delosme 88]. According to [Ochotta 94],

  > Results for the Lam schedule are quite impressive. When compared with other general-purpose annealing schedules such as [Huang *et al.* 86] and even hand-crafted schedules like the one in [Sechen and Sangiovanni-Vincentelli 86], it often provides speed-ups of 50% while actually improving the quality of the final answers [Lam and Delosme 88]. Simulated annealing with the Lam schedule even compares favorably with heuristic combinatorial optimization methods tuned to specific problems like partitioning and the travelling salesman problem [Lam 88].

B.2 The "Modified Lam" Schedule

For the purposes of comparing against STAGE, I sought to implement simulated annealing with a cooling schedule which would both perform very well and would require little tuning from one problem instance to the next. After some experimentation, I settled on an adaptive schedule similar to that used in [Ochotta 94], which in turn was based on Swartz's modification of the Lam schedule [Swartz and Sechen 90]. Swartz observed that for each of a large number of annealing runs using the Lam schedule, the accept ratio—that is, the ratio of moves accepted to moves considered—followed an almost identical pattern with respect to the move counter i [Ochotta 94, p. 137]:

The accept rate starts at about 100%, decreases exponentially until it stabilizes (about 15% of the way through the run) at about 44%, and remains there until about 65% of the way through the run. It then continues its exponential decline until the end of the annealing process. Based on this observation, Swartz duplicated the effect of the Lam schedule by using a simple feedback loop to control the temperature and *force* the accept ratio to follow the curve in Figure [B.1]. In comparing the modified schedule to the original schedule, Swartz reported almost no difference in the quality of the final answers [Swartz and Sechen 90]. One additional benefit of the modified Lam schedule is that, in contrast to the original schedule, the total number of moves in the annealing run can be specified by the user.

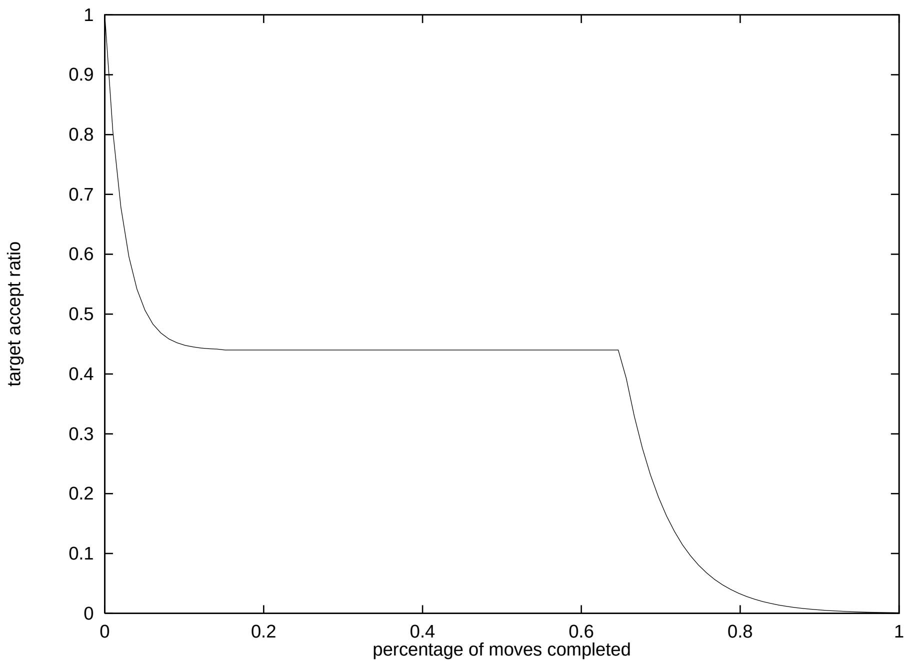


FIGURE B.1. Accept rate targets for the modified Lam schedule

All details of my implementation of Swartz's modified Lam schedule are given below in Table B.2. I do not dynamically readjust the neighborhood structure $N(x)$ over the course of search, as the original Lam schedule did, since such readjustments cannot be specified in a problem-independent manner.

Simulated-Annealing(X, S, N, Obj, TotEvals, **Modified-Lam-Schedule**):

1. Let $x_0 \in S$ be a random starting state for search;
   let AcceptRate:=0.5, $t_0 := 0.5$.
2. For i := 0 to TotEvals $- 1$, do:
   (a) Choose $x' :=$ a random element from $N(x_i)$
   (b) $x_{i+1} := \begin{cases} x' & \text{if } \textsc{rand}[0,1) < e^{[\text{Obj}(x_i) - \text{Obj}(x')]/t_i} \\ x_i & \text{otherwise} \end{cases}$
   (c) Update temperature t_{i+1} as follows:
      i. let AcceptRate $:= \begin{cases} \frac{1}{500}(499 \cdot \textsc{AcceptRate} + 1) & \text{if } x' \text{ was accepted} \\ \frac{1}{500}(499 \cdot \textsc{AcceptRate}) & \text{if } x' \text{ was rejected} \end{cases}$
      ii. let $d := i/$TotEvals
      iii. let TargRate $:= \begin{cases} 0.44 + 0.56 \cdot 560^{-d/0.15} & \text{if } 0 \le d < 0.15 \\ 0.44 & \text{if } 0.15 \le d < 0.65 \\ 0.44 \cdot 440^{-(d-0.65)/0.35} & \text{if } 0.65 \le d < 1 \end{cases}$
      iv. let $t_{i+1} := \begin{cases} t_i \cdot 0.999 & \text{if } \textsc{AcceptRate} > \textsc{TargRate} \\ t_i / 0.999 & \text{otherwise} \end{cases}$
3. Return the best state found.

Table B.2. Details of the "modified Lam" adaptive simulated annealing schedule, instantiating the template of Figure B.1. The TargRate function is plotted in Figure B.1.

B.3 Experiments

I certainly do not claim that the modified Lam schedule is perfectly optimized for every problem attempted in Chapter 4. Getting the best performance from simulated annealing on any given problem is an art that involves refining the temperature schedule, dynamically adjusting the search neighborhood, and tuning the cost function coefficients. However, empirically, the simulated annealing algorithm defined by Table B.2 does seem to perform very well on a wide variety of problems without requiring further tuning.

The following experiments illustrate the effectiveness of my implementation. I consider four of the optimization domains of Chapter 4: bin-packing (§4.2), VLSI channel routing (§4.3), Bayes net learning (§4.4), and cartogram design (§4.6). For each of these domains, I compare the performance of the modified Lam schedule with that of 12 different geometric cooling schedules, defined by $t_i = t_0 \cdot \alpha^i$ for all combinations of

$$\begin{aligned} \text{initial temperature } & t_0 \in \{10, 1, 0.1, 0.01\} \\ \text{cooling rate } & \alpha \in \{0.9999, 0.99999, 0.999999\}. \end{aligned}$$

The total length of the schedules is set to TotEvals, the same setting used in the comparative experiments of Chapter 4: 10^5 for bin-packing and Bayes net learning, $5 \cdot 10^5$ for channel routing, and 10^6 for cartogram design.

Results are shown in Figures B.2–B.5. Each figure plots thirteen boxes: the left-most box corresponds to the modified Lam schedule results (copied from Chapter 4), and the other 12 boxes correspond to the performance of the various geometric cooling schedules (averaged over 10 runs each). In all cases, the modified Lam schedule is nearly as effective as, if not more effective than, the best of the geometric schedules.

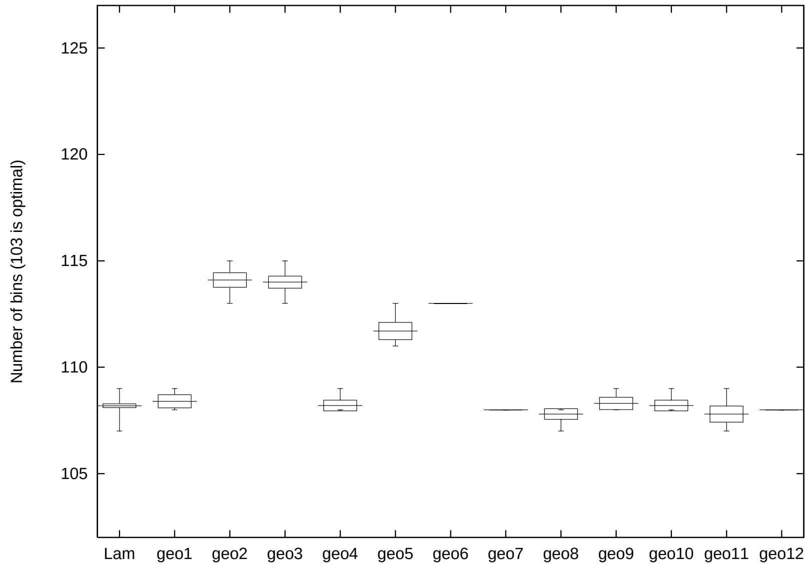


FIGURE B.2. Simulated annealing schedules for bin-packing instance u250_13: Modified Lam schedule (leftmost) versus 12 geometric cooling schedules

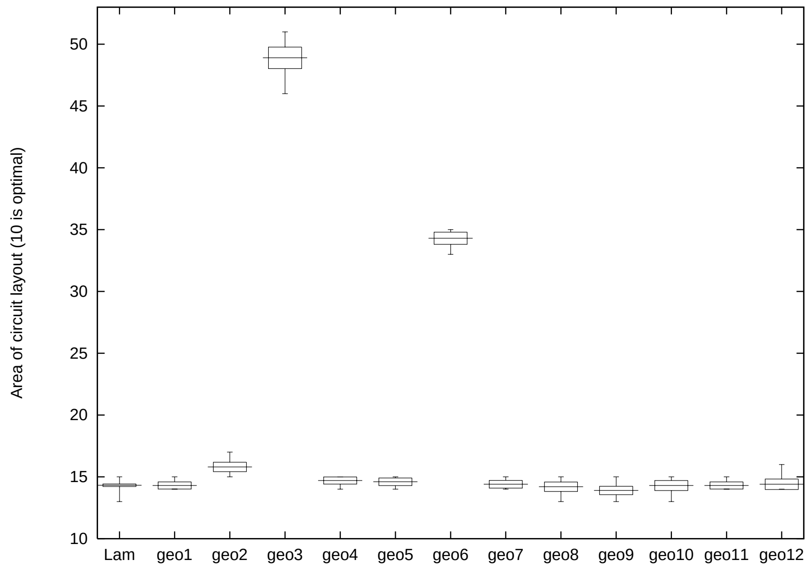


FIGURE B.3. Simulated annealing schedules for channel routing instance YK4: Modified Lam schedule (leftmost) versus 12 geometric cooling schedules

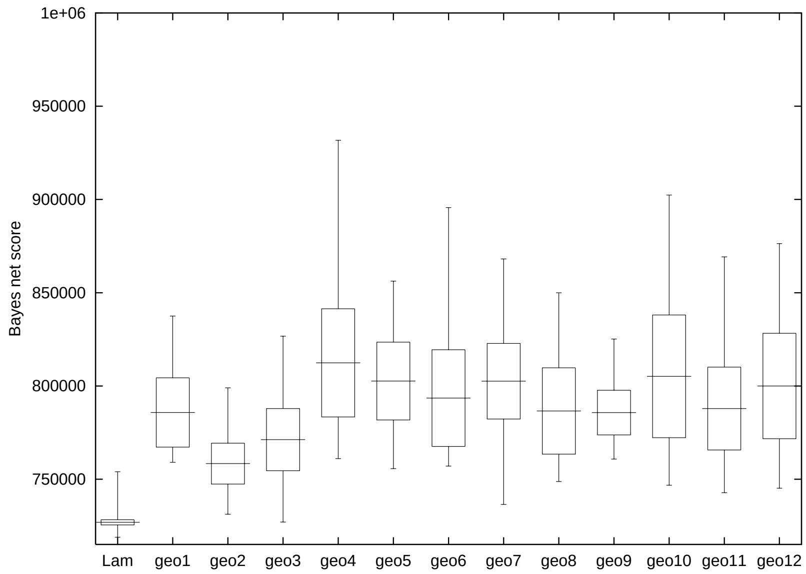


FIGURE B.4. Simulated annealing schedules for Bayes net structure-finding instance SYNTH125K: Modified Lam schedule (leftmost) versus 12 geometric cooling schedules

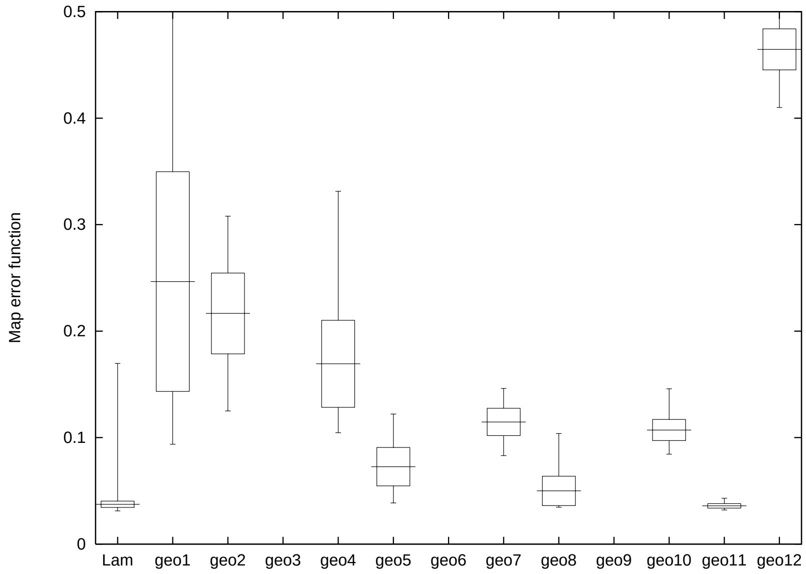


FIGURE B.5. Simulated annealing schedules for cartogram design instance US49: Modified Lam schedule (leftmost) versus 12 geometric cooling schedules

Appendix C

Implementation Details of Problem Instances

This appendix gives implementation details for the optimization problems used in the experiments of Chapter 4. For further information, including the complete datasets used in the bin-packing, Bayes net structure-finding, and cartogram domains, please access the following web page:

$$\texttt{http://www.cs.cmu.edu/\~{}AUTON/stage/} \tag{C.1}$$

C.1 Bin-packing

The bin-packing problem was introduced in Sections 3.3.2 and 4.2. We are given a *bin capacity* C and a list $L = (a_1, a_2, ...a_n)$ of *items*, each having a *size* $s(a_i) > 0$. The goal is to pack the items into as few bins as possible, i.e., partition them into a minimum number m of subsets $B_1, B_2, ..., B_m$ such that for each B_j, $\sum_{a_i \in B_j} s(a_i) \leq C$.

In the STAGE experiments, value function approximation was done with respect to two state features: the objective function itself, and the variance in bin fullness levels. In terms of the above notation, given a packing $x = \{B_1, B_2, \ldots, B_{M_x}\}$ (with all bins B_j assumed non-empty), we have

$$\mathrm{Obj}(x) \stackrel{\mathrm{def}}{=} M_x$$

$$\mathrm{Var}(x) \stackrel{\mathrm{def}}{=} \left(\frac{1}{M_x} \sum_{j=1}^{M_x} \mathrm{fullness}(B_j)^2 \right) - \left(\frac{1}{M_x \cdot C} \sum_{i=1}^{n} s(a_i) \right)^2$$

where

$$\mathrm{fullness}(B_j) \stackrel{\mathrm{def}}{=} \frac{1}{C} \sum_{a_i \in B_j} s(a_i) \; .$$

The illustrative example of Section 3.3.2 consisted of the following 30 items, to be packed into bins of capacity 100:

(27, 23, 23, 23, 27, 26, 26, 51, 26, 23,
51, 23, 51, 23, 23, 51, 23, 23, 27, 23,
51, 27, 51, 26, 27, 23, 26, 27, 26, 23)

These particular item sizes were motivated by Figure 2.5 of [Coffman *et al.* 96], which depicts a template of a worst-case example for the "First Fit Decreasing" offline bin-packing heuristic. The optimal packing fills 9 bins exactly to capacity.

The twenty instances of the u250 class were contributed to the Operations Research Library by Falkenauer [96]. They may be downloaded from the web page referenced at the start of this section, or directly from the OR-Library web site at

`http://www.ms.ic.ac.uk/info.html`.

C.2 VLSI Channel Routing

My implementation of channel routing follows that of the SACR system [Wong *et al.* 88]. This system allows a restricted form of doglegging, whereby a net may be split horizontally only at columns containing a pin belonging to that net.

Most of the experiments in this dissertation were conducted on the instance YK4. That instance is specified by the following pin columns:

Upper: 17 9 23 33 0 17 34 33 32 31 32 20 9 10 21 34 0 31 22 10 0 22 1 3 16 0 0 0 9 19 7 0 16 14 7 51 43 57 67 0 51 68 67 66 65 66 54 43 44 55 68 0 65 56 44 0 56 35 37 50 0 0 0 43 53 41 0 50 48 41 85 77 91 101 0 85 102 101 100 99 100 88 77 78 89 102 0 99 90 78 0 90 69 71 84 0 0 0 77 87 75 0 84 82 75 119 111 125 135 0 119 136 135 134 133 134 122 111 112 123 136 0 133 124 112 0 124 103 105 118 0 0 0 111 121 109 0 118 116 109

Lower: 0 0 0 24 10 0 4 2 21 2 4 23 1 4 24 1 4 2 0 1 4 0 3 19 2 3 20 2 3 14 3 1 9 24 0 0 0 0 58 44 0 38 36 55 36 38 57 35 38 58 35 38 36 0 35 38 0 37 53 36 37 54 36 37 48 37 35 43 58 0 0 0 0 92 78 0 72 70 89 70 72 91 69 72 92 69 72 70 0 69 72 0 71 87 70 71 88 70 71 82 71 69 77 92 0 0 0 0 126 112 0 106 104 123 104 106 125 103 106 126 103 106 104 0 103 106 0 105 121 104 105 122 104 105 116 105 103 111 126 0

These pin columns correspond to Example 1 (Figure 25) of [Yoshimura and Kuh 82], but multiplied four times and placed side by side. By "cloning" the problem in this manner, we maintain the known global optimum (in this case, 10 tracks when restricted doglegging is allowed), but make the problem much more difficult for local search.

In Section 6.2, I applied STAGE to eight additional channel routing instances, HYC1 through HYC8. The pin columns for these problems may be found in [Chao and Harper 96] and are also available on the STAGE web page.

C.3 Bayes Network Learning

For the experiments with the Bayes-net structure-finding domain (Sections 4.4 and 6.1.5), three datasets were used: MPG, ADULT2, and SYNTH125K.

- The MPG dataset contains information on the horsepower, weight, gas mileage, and other such data (10 total attributes) for 392 automobiles. It is derived from the "Auto-Mpg" dataset available from the UCI Machine Learning repository [Merz and Murphy 98], but modified by coarsely discretizing all continuous variables.

- The ADULT2 dataset was also obtained from the UCI repository. It consists of census data related to the job, wealth, nationality, etc. (15 total attributes) on 30,162 individuals.

- The SYNTH125K dataset was generated synthetically from the probability distribution given by the Bayes net in Figure 4.7 (p. 88), designed by Moore and Lee [98].

All three datasets are available for downloading from the STAGE web page.

C.4 Radiotherapy Treatment Planning

The radiotherapy treatment planning domain of Section 4.5 is too complex to explain in full detail here. Please refer to the web page for more information. Here, I describe the domain in enough detail to illustrate its complexity—in particular, to show why we must resort to using local search rather than, say, linear programming to solve it. The form of my objective function was based on discussions with domain experts; however, I did not have access to a medically accurate implementation of the dose calculations and penalty functions. Thus, I claim only that the optimization problem solved here retains most of the overall structure and geometry of tradeoffs found in the true medical domain.

I formulated the problem as follows. The treatment area is discretized into an 80×80 rectangular grid. The radiation dosage $\mathrm{dose}_p(x)$ at each pixel p can then be calculated from the current plan x according to a known forward model. Pixels within the area of the tumor t have a target dose targ_t and incur a penalty based on the ratio $r_p = \frac{\mathrm{dose}_p(x)}{\mathrm{targ}_t}$:

$$\mathrm{Penalty}(p) = \begin{cases} \exp(1/\max(r_p, 0.1)) & \text{if } r_p < 1 \text{ (underdose)} \\ r_p - 1 & \text{if } r_p \geq 1 \text{ (overdose)} \end{cases}$$

Pixels within a sensitive structure s have a maximum acceptable dose accep_s and incur a penalty based on the ratio $r_p = \frac{\text{dose}_p(x)}{\text{accep}_s}$:

$$\text{Penalty}(p) = \begin{cases} r_p & \text{if } r_p \leq 1 \text{ (safe dose)} \\ \exp(r_p) & \text{if } r_p > 1 \text{ (overdose)} \end{cases}$$

These two penalty functions are plotted in Figure C.1. Finally, the overall objective function is calculated as a weighted sum of all the per-pixel penalties. The weights are fixed and reflect the relative importance of the various structures being targeted or protected.

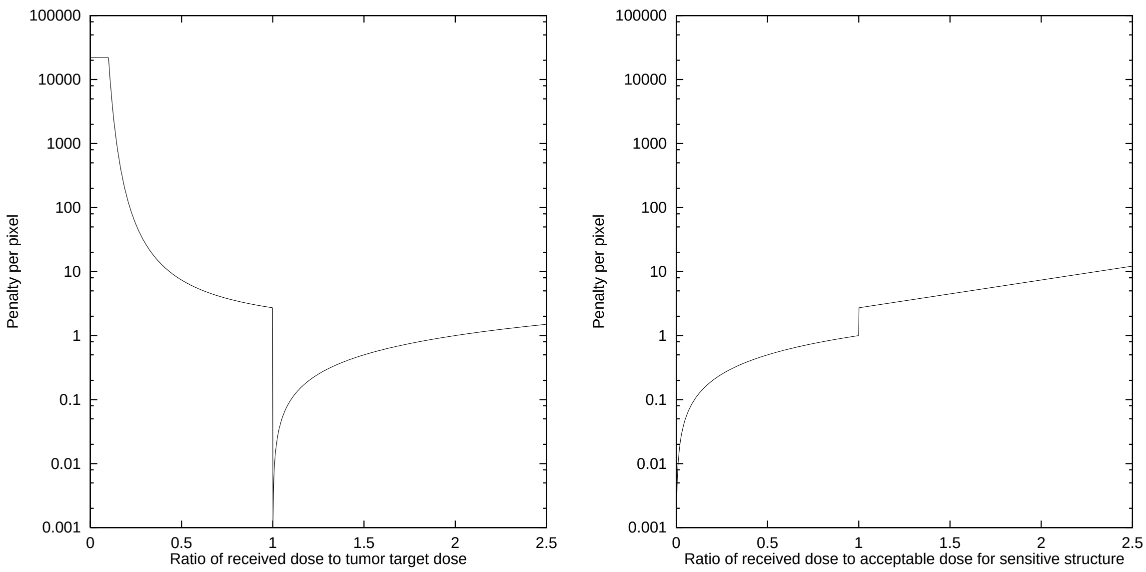


FIGURE C.1. The per-pixel penalty for pixels within a tumor (left) or within a sensitive structure (right). Note the logarithmic scale on the y-axis.

This problem formulation is nearly suitable for a linear programming solution: the dose incurred at each pixel is a linear combination of the beam intensities, and the objective function is a linearly weighted sum of penalty terms. If the penalty functions of Figure C.1 had been defined to be piecewise linear and convex, and the beam intensities were allowed to vary continuously over $[0, 1]$, then the optimal treatment plan x^* would be obtainable by linear programming. However, with 0/1 discrete beam intensities and non-convex penalty functions, as assumed in the experiments of this thesis, we must resort to heuristic search.

C.5 Cartogram Design

The cartogram design problem was introduced in Section 4.6. This thesis investigated a single instance, `US49`. Instance `US49` is defined by the 49 polygons of the continental U.S. map (48 states plus the District of Columbia) and their respective target areas, which are based on the 1990–2000 electoral vote for U.S. President. This dataset is available from the STAGE web page.

Local search moves in this domain consist of choosing one of the 162 points and perturbing it slightly. These points are chosen randomly, but with a bias toward those points that contribute most to the current map error function. To be precise: first a state is chosen with probability proportional to its contribution to $\mathrm{Obj}(x)$, and then one point on that state is chosen uniformly at random. The perturbation is then generated by adding uniformly random numbers in $[-1.5, 1.5]$ to each coordinate.

The objective function is defined as the sum of four penalty terms:

$$\mathrm{Obj}(x) = \Delta\mathrm{Area}(x) + \Delta\mathrm{Gape}(x) + \Delta\mathrm{Orient}(x) + \Delta\mathrm{Segfrac}(x)$$

The penalty terms are defined as follows:

$$\begin{aligned}
\Delta\mathrm{Area}(x) &= \frac{10}{\#\mathrm{States}} \sum_{s\in\mathrm{States}} \left(\max\left(\frac{\mathrm{Area}_x(s)}{\mathrm{Area}_{\mathrm{targ}}(s)}, \frac{\mathrm{Area}_{\mathrm{targ}}(s)}{\max(\mathrm{Area}_x(s), 0.001)}\right) - 1\right)^2 \\
\Delta\mathrm{Gape}(x) &= \frac{1}{\#\mathrm{Bends}} \sum_{\angle ABC\in\mathrm{Bends}} \left(\mathrm{measure}_x(\angle ABC) - \mathrm{measure}_{\mathrm{orig}}(\angle ABC)\right)^2 \\
\Delta\mathrm{Orient}(x) &= \frac{1}{\#\mathrm{Bends}} \sum_{\angle ABC\in\mathrm{Bends}} \left(\mathrm{measure}_x(\angle AB__) - \mathrm{measure}_{\mathrm{orig}}(\angle AB__)\right)^2 \\
\Delta\mathrm{Segfrac}(x) &= \frac{10}{\#\mathrm{Bends}} \sum_{\angle ABC\in\mathrm{Bends}} \left(\frac{\mathrm{length}_x(\overline{AB})}{\mathrm{perim}_x(\mathrm{State}(\angle ABC))} - \frac{\mathrm{length}_{\mathrm{orig}}(\overline{AB})}{\mathrm{perim}_{\mathrm{orig}}(\mathrm{State}(\angle ABC))}\right)^2
\end{aligned}$$

The "Bends" above index each angle of each polygon in the map. The notation "$\angle AB\_\_$" refers to the angle made between line $\overleftrightarrow{AB}$ and the x-axis (a fixed horizontal line). Finally, the constants 1 and 10 in these penalty terms were chosen by trial and error to create aesthetically appealing cartograms.

C.6 Boolean Satisfiability

The difficult "32-bit parity function learning" instances tested in Section 4.7 are described in [Crawford 93]. The complete formulas are available for downloading from the STAGE web page, and also from the DIMACS satisfiability archive:

`ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/` .

References

E. H. L. Aarts and P. J. M. van Laarhoven. A new polynomial-time cooling schedule. In Proc. IEEE Intl. Conf. on Computer-Aided Design, pages 206–208, 1985. (p. 190)

B. Abramson. Expected-outcome: A general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(2):182–193, February 1990. (pp. 169, 170)

D. H. Ackley and M. L. Littman. A case for distributed Lamarckian evolution. In C. Langton, C. Taylor, J. D. Farmer, and S. Ramussen, editors, *Artificial Life III: Santa Fe Institute Studies in the Sciences of Complexity*, volume 10, pages 487–509. Addison-Wesley, Redwood City, CA, 1993. (p. 172)

J. S. Albus. *Brains, Behavior, and Robotics.* Byte Books, Peterborough, NH, 1981. (p. 67)

D. Aldous and U. Vazirani. "Go with the winners" algorithms. In Proceedings of the 35th Symposium on Foundations of Computer Science, pages 492–501, 1994. (p. 164)

D. Applegate, R. Bixby, V. Chvatal, and B. Cook. Finding cuts in the TSP (A preliminary report). Technical Report 95-05, DIMACS, April 10, 1995. (p. 44)

C. G. Atkeson and J. C. Santamaria. A comparison of direct and model-based reinforcement learning. In International Conference on Robotics and Automation, 1997. (p. 147)

C. G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, 1994. (p. 27)

L. Baird. Residual algorithms: Reinforcement learning with function approximation. In Machine Learning: Proceedings of the Twelfth International Conference, 1995. (pp. 15, 25)

S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In Proc. 12th International Conference on Machine Learning, pages 38–46. Morgan Kaufmann, 1995. (p. 172)

S. Baluja and S. Davies. Combining multiple optimization runs with optimal dependency trees. Technical Report CMU-CS-97-157, Carnegie Mellon University School of Computer Science, 1997. (pp. 76, 172)

R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. Journal of Heuristics, 1(1):9–32, 1995. (pp. 72, 74)

A. Barto, R. Sutton, and C. Watkins. Learning and sequential decision making. Technical Report COINS 89-95, University of Massachusetts, 1989. (p. 22)

A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time learning and control using asynchronous dynamic programming. Artificial Intelligence, 1995. (pp. 22, 23)

J. Baxter, A. Tridgell, and L. Weaver. KnightCap: A chess program that learns by combining TD(λ) with minimax search. Technical report, Department of Systems Engineering, Australian National University, Canberra, Australia, November 1997. (p. 170)

R. Bellman, R. Kalaba, and B. Kotkin. Polynomial approximation—a new computational technique in dynamic programming: Allocation processes. Mathematics Of Computation, 17, 1963. (p. 22)

R. Bellman. *Dynamic Programming.* Princeton University Press, 1957. (pp. 15, 21)

R. Bellman. *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company, 1978. (p. 22)

D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments.* Chapman and Hall, 1985. (p. 34)

D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996. (pp. 15, 21, 22, 23, 25, 26, 66, 68, 139, 140, 141, 143, 144, 145, 152)

D. Bertsekas, J. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. Technical Report LIDS-P 2386, MIT Laboratory for Information and Decision Systems, 1997. (p. 169)

D. Bertsekas. A counterexample to temporal differences learning. Neural Computation, 7:270–9, 1995. (p. 15)

J.R. Beveridge, C. Graves, and C. E. Lesher. Local search as a tool for horizon line matching. Technical Report CS-96-109, Colorado State University, 1996. (pp. 43, 72, 163)

K. D. Boese, A. B. Kahng, and S. Muddu. A new adaptive multi-start technique for combinatorial global optimizations. Operations Research Letters, 16:101–113, 1994. (pp. 164, 165, 172)

K. D. Boese. Cost versus distance in the traveling salesman problem. Technical Report CSD-950018, UCLA Computer Science Department, May 1995. (pp. 164, 165)

K. D. Boese. *Models for Iterative Global Optimization*. PhD thesis, UCLA Computer Science Department, 1996. (pp. 164, 190)

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. (pp. 15, 16, 25, 27, 29)

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. (pp. 16, 27)

J. A. Boyan and A. W. Moore. Using prediction to improve combinatorial optimization search. In Proceedings of the Sixth International Workshop on Artificial Intelligence and Statistics (AISTATS), January 1997. (p. 16)

J. A. Boyan and A. W. Moore. Learning evaluation functions for global optimization and Boolean satisfiability. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI), 1998. (Outstanding Paper Award). (p. 16)

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. Internet resource available at `http://www.cs.cmu.edu/~reinf/ml95/`. (pp. 15, 21)

J. A. Boyan, D. Freitag, and T. Joachims. A machine learning architecture for optimizing web search engines. Technical Report AAAI Technical Report WS-96-06, Proceedings of the AAAI workshop on Internet-Based Information Systems, 1996. (p. 43)

J. A. Boyan. Modular neural networks for learning context-dependent game strategies. Master's thesis, University of Cambridge, Department of Engineering and Computer Laboratory, 1992. (pp. 15, 23, 170)

S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. Machine Learning, 22(1/2/3):33–57, 1996. (pp. 17, 139, 145, 177)

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984. (p. 67)

W. L. Buntine, L. Su, R. Newton, and A. Mayer. Adaptive methods for netlist partitioning. In Proceedings of ICCAD-97, San Jose, CA, November 1997. (p. 169)

M. S. Campbell. Deep Blue: The IBM chess machine. Talk given at Carnegie Mellon University, March 1998. (p. 14)

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. (pp. 92, 93)

H-Y. Chao and M. P. Harper. An efficient lower bound algorithm for channel routing. Integration: The VLSI Journal, 1996. (pp. 81, 156, 198)

D. M. Chickering, D. Geiger, and D. Heckerman. Learning bayesian networks is NP-hard. Technical Report MSR-TR-94-17, Microsoft Research, November 1994. (pp. 86, 87)

J. Christensen and R. Korf. A unified theory of heuristic evaluation functions and its application to learning. In Proceedings of the 4th National Conference on Artificial Intelligence, pages 148–152, 1986. (p. 22)

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. (pp. 22, 170)

W. S. Cleveland and S. J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. Journal of the American Statistical Association, 83(403):596–610, September 1988. (pp. 40, 67)

E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, 1996. (pp. 44, 45, 78, 198)

J. M. Cohn. *Automatic Device Placement for Analog Cells in KOAN*. PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, February 1992. (pp. 49, 180)

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. (p. 26)

J. Crawford. Propositional versions of parity learning problems. Internet resource available by anonymous FTP, at `ftp://dimacs.rutgers.edu/pub/challenge/-sat/contributed/crawford/README`, July 1993. (pp. 98, 201)

R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, 1996. (pp. 15, 23)

S. Davies and S. Baluja. Personal communication, April 1998. (p. 173)

J. S. de Bonet, C. L. Isbell Jr., and P. Viola. MIMIC: Finding optima by estimating probability densities. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 424. The MIT Press, 1997. (p. 172)

R. Dearden, N. Friedman, and S. Russell. Bayesian Q-Learning. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI), 1998. (p. 132)

F. D'Epenoux. A probabilistic production and inventory problem. Management Science, 10:98–108, 1963. (p. 21)

D. N. Deutsch. A 'dogleg' channel router. In Proceedings of the 13th ACM/IEEE Design Automation Conference, pages 425–433, 1976. (p. 81)

J. Doran and D. Michie. Experiments with the graph traverser program. Proceedings of the Royal Society of London, 294, Series A:235–259, 1966. (p. 171)

M. Dorigo and L. Gambardella. Ant-Q: A reinforcement learning approach to combinatorial optimization. Technical Report 95-01, Irdia, Université Libre de Bruxelles, 1995. (p. 169)

D. Dorling. Cartograms for visualizing human geography. In H. M. Hearnshaw and D. J. Unwin, editors, *Visualization in Geographical Information Systems*, pages 85–102. Wiley, 1994. (p. 95)

D. Dorling. *Area cartograms: their use and creation.* Number 59 in Concepts and Techniques in Modern Geography. University of East Anglia: Environmental Publications, 1996. (p. 95)

Q. Duan, S. Sorooshian, and V. Gupta. Effective and efficient global optimization of conceptual rainfall-runoff models. Water Resources Research, 28(4):1015–1031, 1992. (p. 43)

S. Fahlman and C. Lebiere. The Cascade-Correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990. (p. 67)

E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. In Proc. of the IEEE 1992 International Conference on Robotics and Automation, pages 1186–1192, Nice, France, May 1992. (pp. 49, 76)

E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. Journal of Heuristics, 2(1):5–30, 1996. (pp. 76, 78, 79, 198)

N. Friedman and Z. Yakhini. On the sample complexity of learning Bayesian networks. In Proc. 12th Conference on Uncertainty in Artificial Intelligence, 1996. (p. 86)

J. H. Friedman. Multivariate adaptive regression splines. Annals of Statistics, 19(1):1–67, 1991. (p. 67)

N. Friedman. Learning belief networks in the presence of missing values and hidden variables. In Proc. 14th International Conference on Machine Learning, pages 125–133. Morgan Kaufmann, 1997. (pp. 86, 87)

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. (pp. 44, 74, 97)

F. Glover and M. Laguna. Tabu search. In Modern Heuristic Techniques for Combinatorial Problems. Scientific Publications, Oxford, 1993. (p. 166)

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989. (p. 171)

G. Gordon. Stable function approximation in dynamic programming. In Proceedings of the 12th International Conference on Machine Learning. Morgan Kaufmann, 1995. (pp. 15, 25, 29)

S. M. Gusein-Zade and V. S. Tikunov. A new technique for constructing continuous cartograms. Cartography and Geographic Information Systems, 20(3):167–173, 1993. (p. 95)

L. W. Hagen and A. B. Kahng. Combining problem reduction and adaptive multi-start: A new technique for superior iterative partitioning. IEEE Transactions on CAD, 16(7):709–717, 1997. (p. 166)

B. Hajek. Cooling schedules for optimal annealing. Math. Oper. Res., 13(2):311–329, 1988. (p. 190)

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. (p. 20)

D. Heckerman, D. Geiger, and M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. Technical Report MSR-TR-94-09, Microsoft Research, 1994. (pp. 86, 87)

G. E. Hinton and S. J. Nowlan. How learning can guide evolution. Complex Systems, 1(1):495–502, June 1987. (p. 172)

R. Howard. *Dynamic Programming and Markov Processes.* MIT Press and John Wiley & Sons, 1960. (pp. 21, 169)

F. Hsu, T. S. Anantharaman, M. S. Campbell, and A. Nowatzyk. A grandmaster chess machine. Scientific American, 263(4):44–50, October 1990. (p. 14)

M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In Proceedings of the International Conference on Computer-Aided Design, pages 381–384, November 1986. (p. 190)

A. Jagota, L. Sanchis, and R. Ganesan. Approximating maximum clique using neural network and related heuristics. In D. S. Johnson and M. A. Trick, editors, *DIMACS Series: Second DIMACS Challenge.* American Mathematical Society, 1996. (p. 166)

A. Jagota. An adaptive, multiple restarts neural network algorithm for graph coloring. European J. of Oper. Res., 93:257–270, 1996. (p. 166)

Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. In First International Joint Workshop on Artificial Intelligence and Operations Research, Timberline, Oregon, 1995. (p. 104)

D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization.* Wiley and Sons, 1995. (to appear). (pp. 163, 165, 166, 172)

D. S. Johnson, C. R. Aragon, L. A. Mcgeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. Oper. Res., 37:865–892, 1989. (p. 190)

D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph colouring and number partitioning. Operations Research, 39:378–406, 1991. (p. 190)

S. A. Johnson, J. R. Stedinger, C. A. Shoemaker, Y. Li, and J. A. Tejada-Guibert. Numerical solution of continuous-state dynamic programs using linear and spline interpolation. Operations Research, 41(3):484–500, 1993. (p. 22)

D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. Unpublished manuscript, April 1996. (p. 72)

A. Juels and M. Wattenberg. Stochastic hillclimbing as a baseline mathod for evaluating genetic algorithms. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 430–436. The MIT Press, 1996. (p. 163)

L. P. Kaelbling. *Learning in Embedded Systems.* The MIT Press, Cambridge, MA, 1993. (p. 179)

R. Karp. Algorithms as a tool for molecular biology. Talk given at Carnegie Mellon University, archived on Internet at `http://ulserver.speech.cs.cmu.edu/-v/karp/`, September 1997. (p. 43)

H. Kautz. Personal communication, April 1998. (p. 101)

C. Kenyon. Best-fit bin-packing with random order. In Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pages 359–364, 1996. (p. 78)

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimisation by simulated annealing. Science, 220:671–680, 1983. (pp. 46, 190)

L. Kuvayev and R. Sutton. Approximation in model-based learning. In C. G. Atkeson and G. J. Gordon, editors, *ICML-97 Workshop on Modelling in Reinforcement Learning*, 1997. Internet resource available at `http://www.cs.cmu.edu/~ggordon/-ml97ws/`. (p. 147)

W. Lam and F. Bacchus. Learning Bayesian belief networks: An approach based on the MDL principle. Computational Intelligence, 10(4), 1994. (p. 86)

J. Lam and J.-M. Delosme. Performance of a new annealing schedule. In ACM/IEEE, editor, *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 306–311, Anaheim, CA, June 1988. IEEE Computer Society Press. (pp. 73, 190)

J. Lam. *An Efficient Simulated Annealing Schedule.* PhD thesis, Computer Science Department, Yale University, 1988. (p. 190)

K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. Artificial Intelligence, 36, 1988. (pp. 22, 170)

S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. Operations Research, 21:498–516, 1973. (pp. 14, 45)

Z.-M. Lin. An efficient optimum channel routing algorithm. In IEEE Proceedings of the SOUTHEASTCON, volume 2, pages 1179–1183, 1991. (p. 81)

L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993. (pp. 144, 168)

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. (pp. 15, 20, 34)

M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In Proc. 11th International Conference on Machine Learning, pages 157–163. Morgan Kaufmann, 1994. (p. 20)

M. L. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Department of Computer Science, Brown University, Providence, RI, 1996. Also Technical Report CS-96-09. (p. 20)

S. Mahadevan, N. Marchalleck, T. K. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In Proc. 14th International Conference on Machine Learning, pages 202–210. Morgan Kaufmann, 1997. (p. 20)

O. C. Martin and S. W. Otto. Combining simulated annealing with local search heuristics. Technical Report CS/E 94-016, Oregon Graduate Institute Department of Computer Science and Engineering, June 1994. (pp. 164, 165, 172)

D. McAllester, H. Kautz, and B. Selman. Evidence for invariants in local search. In Proceedings of AAAI-97, 1997. (p. 98)

C.J. Merz and P.M. Murphy. UCI repository of machine learning databases, 1998. Internet resource available at `http://www.ics.uci.edu/~mlearn/-MLRepository.html`. (p. 199)

T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997. (p. 86)

D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. Adv. in Applied Probability, 18:747–771, 1986. (p. 190)

R. Moll, A. Barto, T. Perkins, and R. Sutton. Reinforcement and local search: A case study. Technical Report UM-CS-1997-044, University of Massachusetts, Amherst, Computer Science, October, 1997. (pp. 66, 108)

J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. Neural Computation, 1(2):281–294, 1989. (p. 67)

A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. Machine Learning, 13:103–130, 1993. (pp. 15, 21, 132, 147)

A. W. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. Journal of Artificial Intelligence Research, 8:67–91, 1998. (pp. 87, 88, 199)

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. (pp. 179, 182)

A. W. Moore, J. G. Schneider, J. A. Boyan, and M. S. Lee. Q2: Memory-based active learning for optimizing noisy continuous functions. In Proceedings of the Fifteenth Interational Conference on Machine Learning (ICML), 1998. (p. 182)

D. Moriarty, A. Schultz, and J. Grefenstette. Reinforcement learning through evolutionary computation. Technical Report NCARAI Report AIC-97-015, Naval Research Laboratory, 1997. (p. 181)

Y. Nakakuki and N. M. Sadeh. Increasing the efficiency of simulated annealing search by learning to recognize (un)promising runs. Technical Report CMU-RI-TR-94-30, CMU Robotics Institute, 1994. (p. 179)

A. Neumaier. Molecular modeling of proteins and mathematical prediction of protein structure. SIAM Rev., 39:407–460, 1997. (p. 43)

N.J. Nilsson. *Principles of Artificial Intelligence*. McGraw-Hill, 1980. (p. 14)

E. Ochotta. *Synthesis of High-Performance Analog Cells in ASTRX/OBLX*. PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, April 1994. (pp. 181, 190)

T. Perkins, R. Moll, and S. Zilberstein. Filtering to improve the performance of multi-trial optimization algorithms. Unpublished manuscript, November 1997. (p. 179)

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. (p. 181)

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. (pp. 69, 144, 178, 181, 190)

M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision processes. Management Science, 24:1127–1137, 1978. (p. 21)

M. L. Puterman. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994. (pp. 14, 15)

L. A. Rendell. A new basis for state-space learning systems and a successful implementation. Artificial Intelligence, 20:369–392, July 1983. (p. 170)

D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8. MIT Press, 1986. (p. 67)

G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR166, Cambridge University Engineering Department, 1994. (p. 21)

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995. (p. 46)

J. Rust. Numerical dynamic programming in economics. In Handbook of Computational Economics. Elsevier, North Holland, 1996. (p. 22)

A. L. Samuel. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3:211–229, 1959. (pp. 22, 170)

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. (pp. 22, 170)

J. G. Schneider, J. A. Boyan, and A. W. Moore. Value function based production scheduling. In Proceedings of the Fifteenth International Conference on Machine Learning (ICML), 1998. (p. 37)

N. Schraudolph, P. Dayan, and T. Sejnowski. Using TD(λ) to learn an evaluation function for the game of Go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *NIPS-6*. Morgan Kaufmann, 1994. (p. 170)

C. Sechen and A. Sangiovanni-Vincentelli. TimberWolf 3.2: A new standard cell placement and global routing package. In Proceedings of the IEEE/ACM Design Automation Conference, pages 432–439, 1986. (p. 190)

B. Selman and H. A. Kautz. An empirical study of greedy local search for satisfiability testing. In Proceedings of the 11th National Conference on Artificial Intelligence, pages 46–53, Menlo Park, CA, USA, July 1993. AAAI Press. (p. 45)

B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. (pp. 46, 97, 98)

B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI), 1997. (p. 98)

S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 974. The MIT Press, 1997. (p. 152)

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. Machine Learning, 22:123–158, 1996. (pp. 21, 141)

S. P. Singh and R. Yee. An upper bound on the loss from approximate optimal-value functions (Technical Note). Machine Learning, 1994. (p. 25)

S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement learning algorithms. Machine Learning, 1998. To appear. (pp. 21, 23, 24)

L. Su, W. L. Buntine, R. Newton, and B. Peters. Learning as applied to stochastic optimization for standard cell placement. International Conference on Computer Design (Submitted), 1998. (p. 180)

R. Subramanian, R. P. Scheff Jr., J. D. Quillinan, D. S. Wiper, and R. E. Marsten. Coldstart: Fleet assignment at delta air lines. Interfaces, 24(1):104–120, Jan.-Feb. 1994. (pp. 43, 44)

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. (p. 140)

R. S. Sutton and S. D. Whitehead. Online Learning with Random Representations. In Machine Learning: Proceedings of the Tenth International Conference, pages 314–321, San Mateo, CA, 1993. Morgan Kaufmann. (pp. 67, 68)

R. S. Sutton. Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation. Technical Note TN87-509.1, GTE Laboratories, May 1987. (p. 24)

R. S. Sutton. Learning to predict by the methods of temporal differences. Machine Learning, 3, 1988. (pp. 15, 21, 23, 139, 140, 141)

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. (pp. 21, 22, 147)

R. S. Sutton. Gain adaptation beats least squares. In Proceedings of the 7^{th} Yale Workshop on Adaptive and Learning Systems, pages 161–166, 1992. (p. 152)

R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996. (pp. 25, 29, 141)

W. Swartz and C. Sechen. New algorithms for the placement and routing of macro cells. In Satoshi Sangiovanni-Vincentelli, Alberto; Goto, editor, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 336–339, Santa Clara, CA, November 1990. IEEE Computer Society Press. (pp. 73, 190, 191)

S. Szykman and J. Cagan. A simulated annealing-based approach to three-dimensional component packing. ASME Journal of Mechanical Design, 117, June 1995. (pp. 43, 49)

T. G. Szymanski. Dogleg channel routing is NP-complete. IEEE Transactions on Computer-Aided Design, CAD-4:31–40, 1985. (p. 81)

G. Tesauro and G. R. Galperin. On-line policy improvement using Monte-Carlo search. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1997. (pp. 169, 170)

G. Tesauro. Practical issues in temporal difference learning. Machine Learning, 8(3/4), May 1992. (pp. 15, 22, 23, 170)

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. Neural Computation, 6(2):215–219, 1994. (p. 22)

S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, March 1992. (pp. 23, 24, 132)

S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, Cambridge, MA, 1995. (p. 170)

J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, MIT, 1996. (pp. 15, 23, 25, 139, 141)

W. Tunstall-Pedoe. Genetic algorithms optimizing evaluation functions. International Computer Chess Association Journal, 14(3):119–128, 1991. (p. 181)

P. Utgoff and J. Clouse. Two kinds of training information for evaluation function learning. In Proceedings of the National Conference on Artificial Intelligence (AAAI), 1991. (p. 181)

P. E. Utgoff. Feature function learning for value function approximation. Technical Report UM-CS-1996-009, University of Massachusetts, Amherst, Computer Science, January, 1996. (p. 178)

C. Watkins. *Learning From Delayed Rewards*. PhD thesis, Cambridge University, 1989. (pp. 15, 21, 22, 180)

S. Webb. Optimization by simulated annealing of three-dimensional conformal treatment planning for radiation fields defined by a multileaf collimator. Phys. Med. Biol., 36:1201–1226, 1991. (pp. 43, 93)

S. Webb. Optimising the planning of intensity-modulated radiotherapy. Phys. Med. Biol., 39:2229–2246, 1994. (p. 93)

A. Wilk. Constraint satisfaction channel routing. Technical Report CS96-04, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, 1996. (pp. 81, 82)

D. F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design*. Kluwer, 1988. (pp. 14, 43, 48, 81, 82, 83, 84, 113, 116, 121, 172, 198)

K. Woolsey. Rollouts. Inside Backgammon, 1(5):4–7, September-October 1991. (p. 169)

T. Yoshimura and E. S. Kuh. Efficient algorithms for channel routing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-1(1):21–35, January 1982. (pp. 81, 198)

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pages 1114–1120, 1995. (pp. 15, 23, 163, 166)

W. Zhang and T. G. Dietterich. Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. Submitted for publication, 1998. (pp. 157, 167, 168)

W. Zhang. *Reinforcement Learning for Job-Shop Scheduling*. PhD thesis, Oregon State University, 1996. (p. 166)

M. Zweben, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, chapter 8, pages 241–255. Morgan Kaufmann, 1994. (pp. 167, 168)