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.

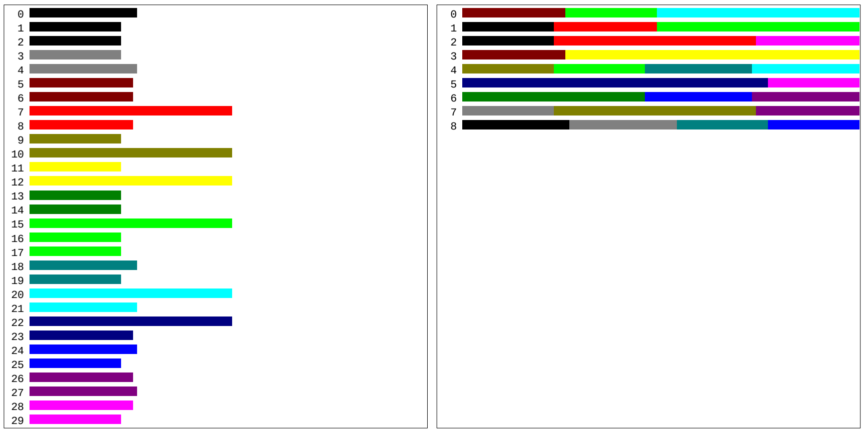


FIGURE 3.1. A small bin-packing domain. Left: initial state (30 items, each in its own bin). Right: the global optimum (nine bins packed with 0 waste).

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

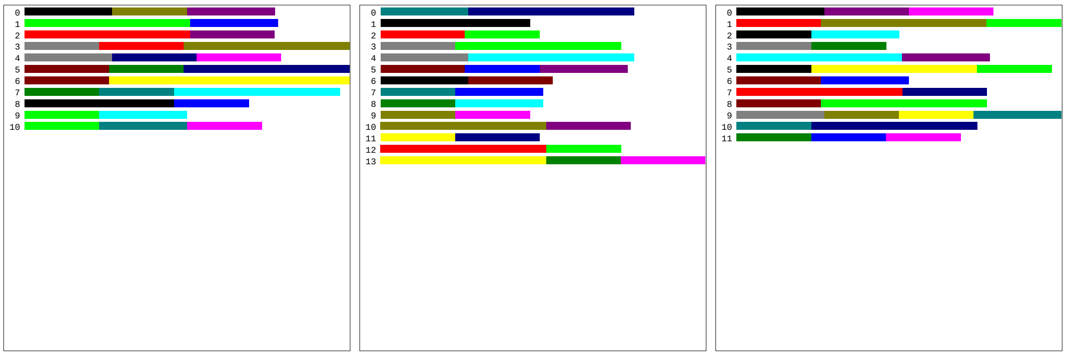


FIGURE 3.2. Three bin-packing local-optima

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, 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.

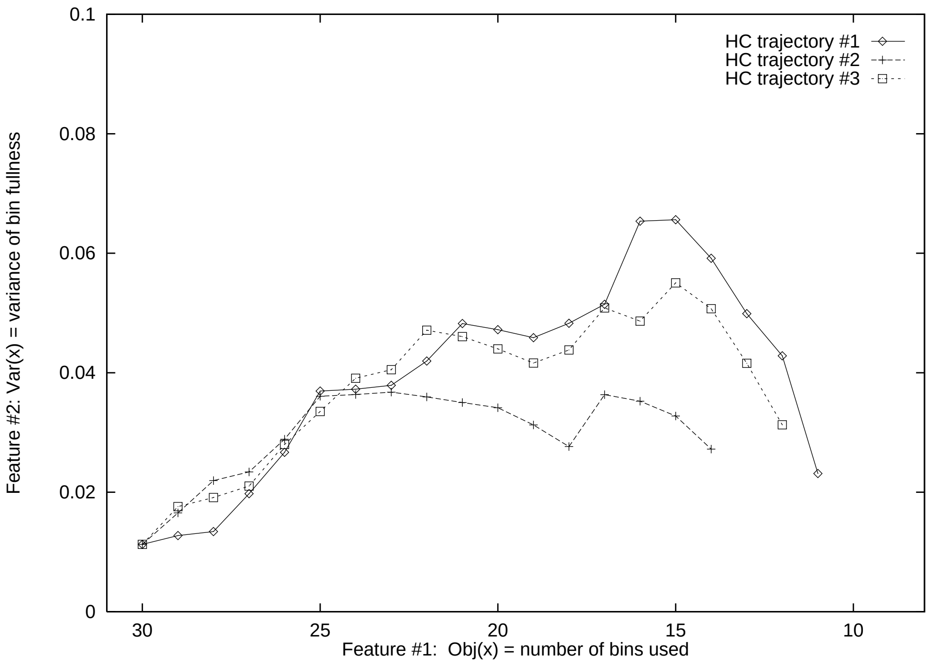


FIGURE 3.3. Three stochastic hillclimbing trajectories in bin-packing feature space

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:
>
> $$C = w^2 + \lambda_p \cdot p^2 + \lambda_U \cdot U \tag{3.1}$$
>
> where ... λ_p and λ_U are constants, ...

In this application, the authors hand-tuned the coefficients and set $\lambda_p = 0.5, \quad \lambda_U = 10$. (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.

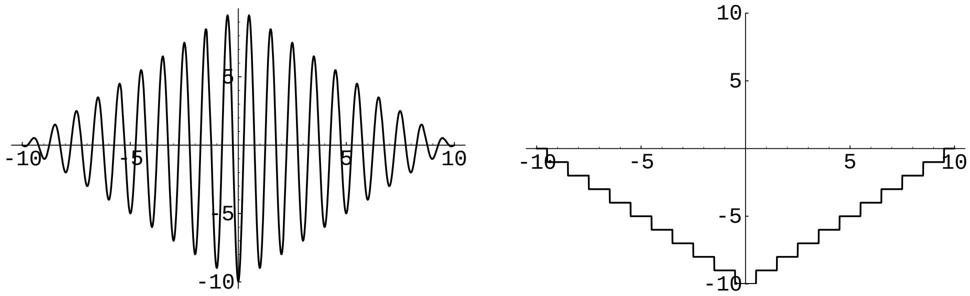


FIGURE 3.4. Left: Obj(x) for a one-dimensional function minimization domain. Right: the value function $V^\pi(x)$ which predicts hillclimbing's performance on that domain.

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.

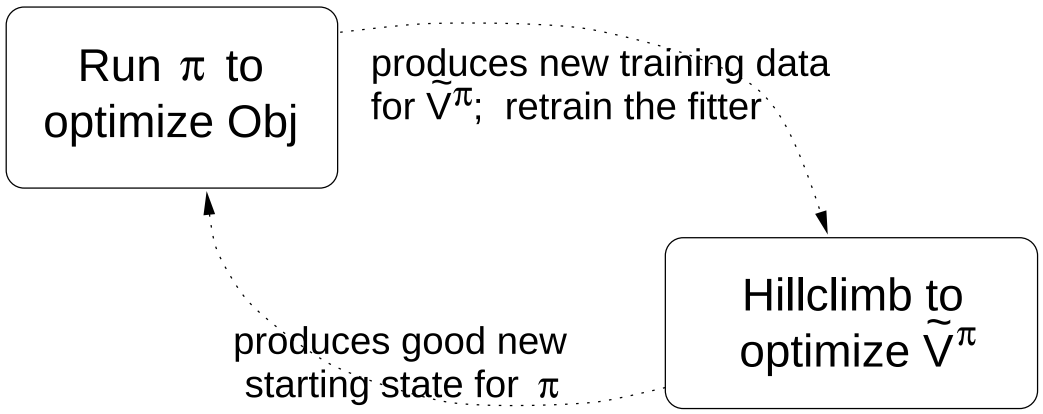


FIGURE 3.5. A diagram of the main loop of STAGE

A compact specification of the algorithm is given in Table 3.2.2 (p. 52). 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, $\text{Obj} : X \to \Re$, 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 $y_i := \min_{j=i\ldots T} \text{Obj}(x_j)$, and add the pair $\{F(x_i) \mapsto y_i\}$ to the training set for *Fit*. Retrain *Fit*; call the resulting learned evaluation function $\tilde{V}^\pi$.

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. 47). 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. 48), 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, 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. $\tilde{V}^\pi(F(z_{t+1})) <$ 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, $x_0 = 7.8$.

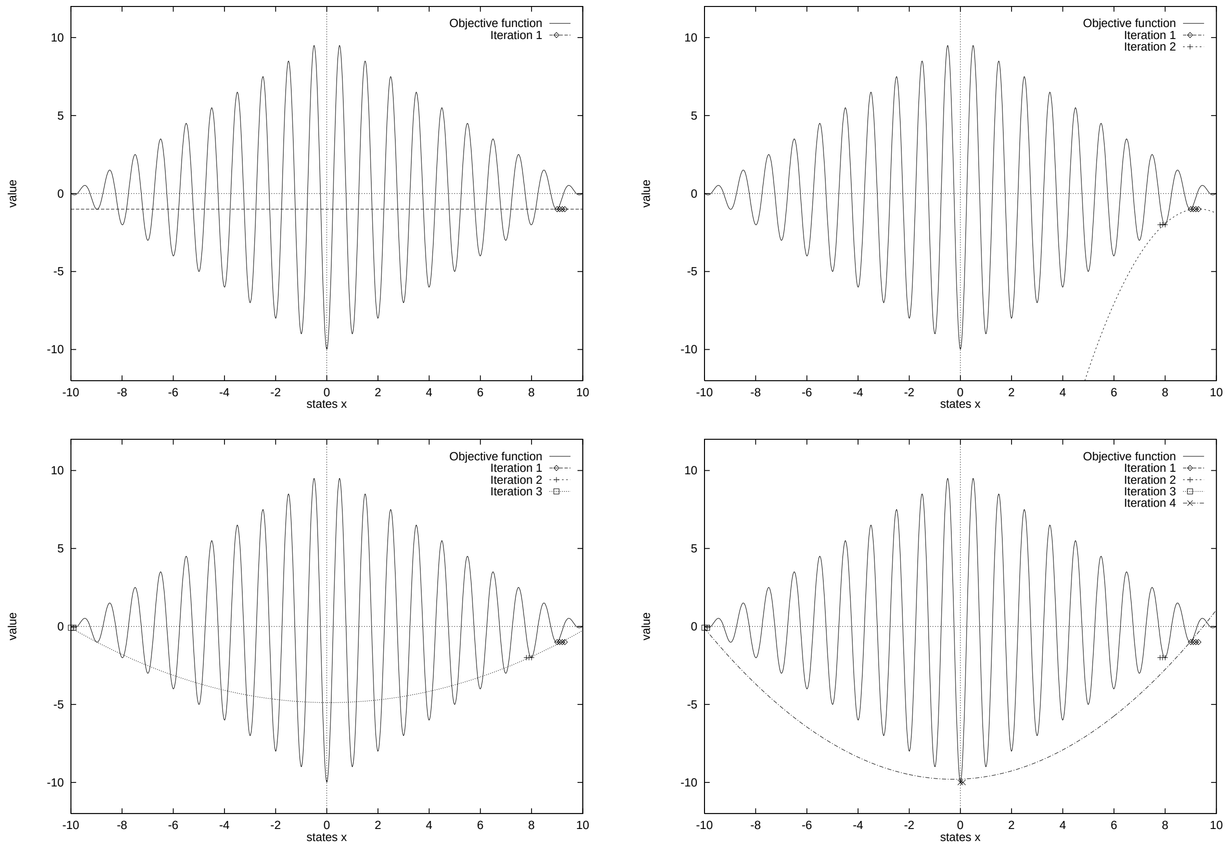


FIGURE 3.6. STAGE working on the 1-D wave example

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. 42). The baseline search from which STAGE will learn is stochastic hillclimbing over the search neighborhood described on page 43. The starting state is the packing which places each item in its own separate bin.

Recall that in Figure 3.3 (p. 46), 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 $(\text{Obj}(x) = 11, \text{Var}(x) = 0.022)$. 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 $(\text{Obj}(x) = 11, \text{Var}(x) = 0.022)$ up to $(\text{Obj}(x) = 12, \text{Var}(x) = 0.105)$. 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 $(\text{Obj}(x) = 10, \text{Var}(x) = 0.053)$.

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 $(\text{Obj}(x) = 9, \text{Var}(x) = 0)$ 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 $\text{Obj}(x) = 11$.

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.

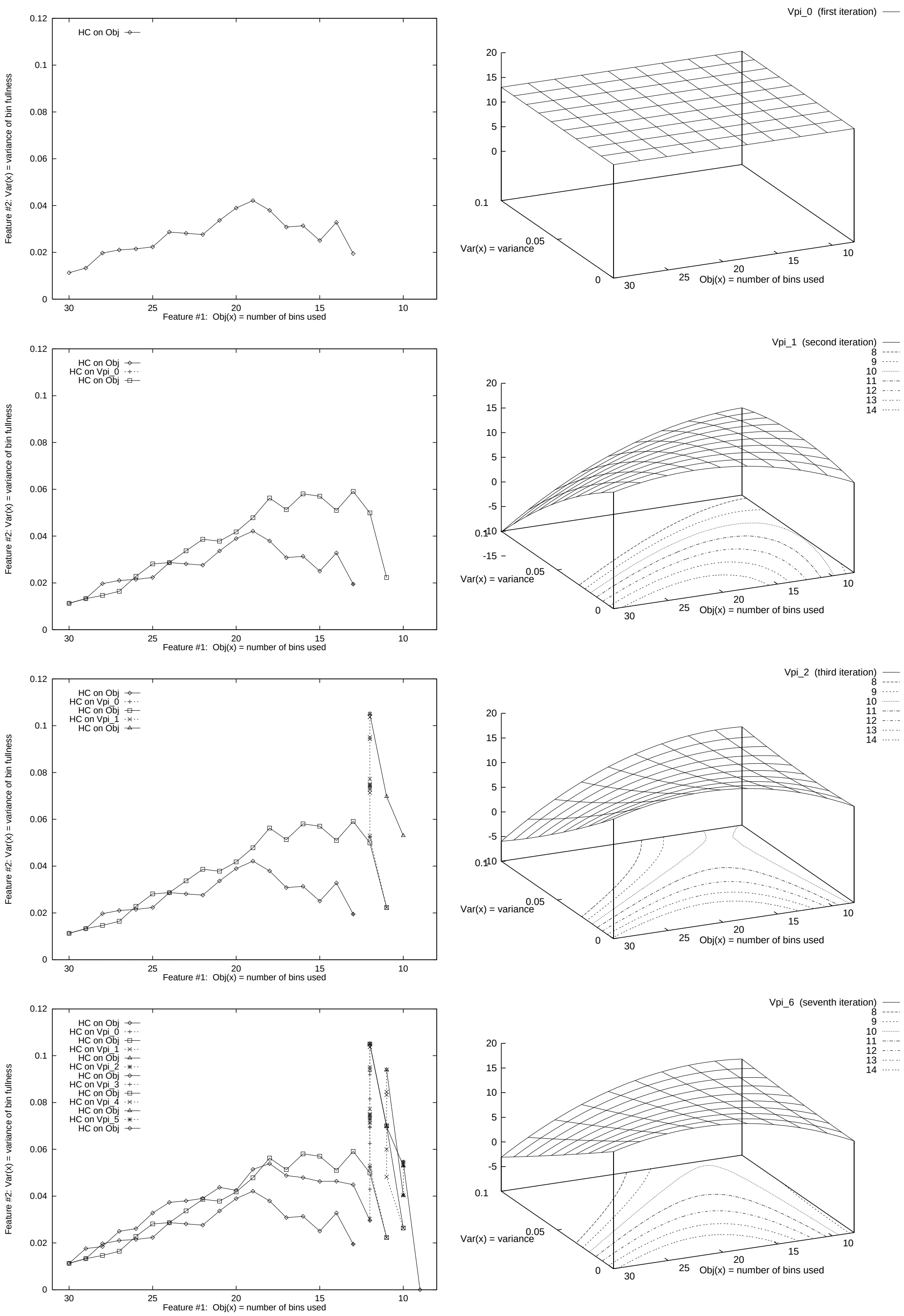


FIGURE 3.7. STAGE working on the bin-packing example

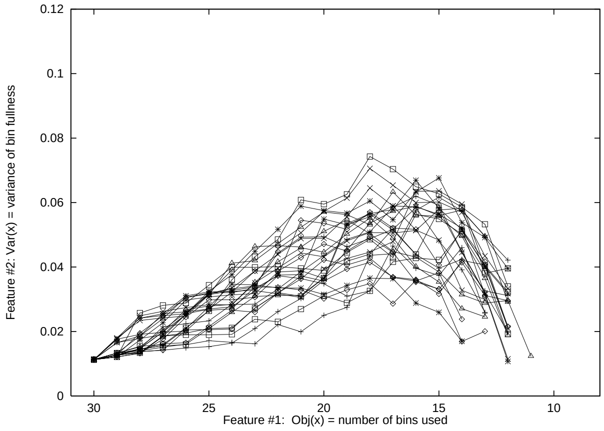


FIGURE 3.8. Trajectory of multi-restart hillclimbing in bin-packing feature space

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

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

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

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

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 52), 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 52. 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:

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

where all expectations are taken over the trajectory distribution $\mathrm{P}(\tau | x_0^\tau = x)$, and the additive cost function R is defined by

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

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:

$$p(\text{END}|x) = \left(\frac{|N(x)|-|G(x)|}{|N(x)|}\right)^{\text{PAT}}$$

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

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 $p(\text{END}|x)$ 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 $\tau = (x_0^\tau, x_1^\tau, x_2^\tau, \ldots)$ with probability $P(\tau|x_0^\tau = x_0)$, then with that same probability, $\text{BSF}(\pi)$ produces

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

where $(i_0, i_1, i_2, \ldots)$ 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 52, 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.

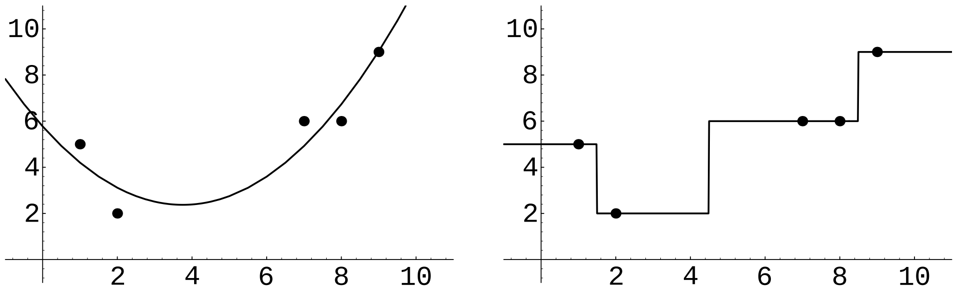


Figure 3.9. Quadratic regression and 1-nearest-neighbor

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.

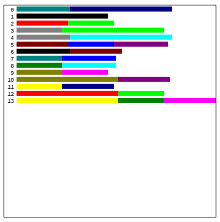
state	$\mapsto$	features	$\mapsto$	basis vector	$\mapsto$	prediction
$x \in X$	$\mapsto$	$F(x) \in \Re^D$	$\mapsto$	$\Phi(F(x)) \in \Re^K$	$\mapsto$	$\boldsymbol{\beta} \cdot \Phi(F(x)) \in \Re$
	$\mapsto$	$\{Obj = 14,$ $Var = 0.027\}$	$\mapsto$	$\{1, 14, 0.027,$ $196, 0.381,$ $0.000741\}$	$\mapsto$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

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

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 $\mathrm{O}(K^2)$ to compute the outer product $\boldsymbol{\phi}_{(i)}\boldsymbol{\phi}_{(i)}^{\mathsf{T}}$ and $\mathrm{O}(K)$ 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 $\mathrm{O}(K^3)$ 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 52). 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.