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: $\mu \pm \frac{2\sigma}{\sqrt{N}}$.

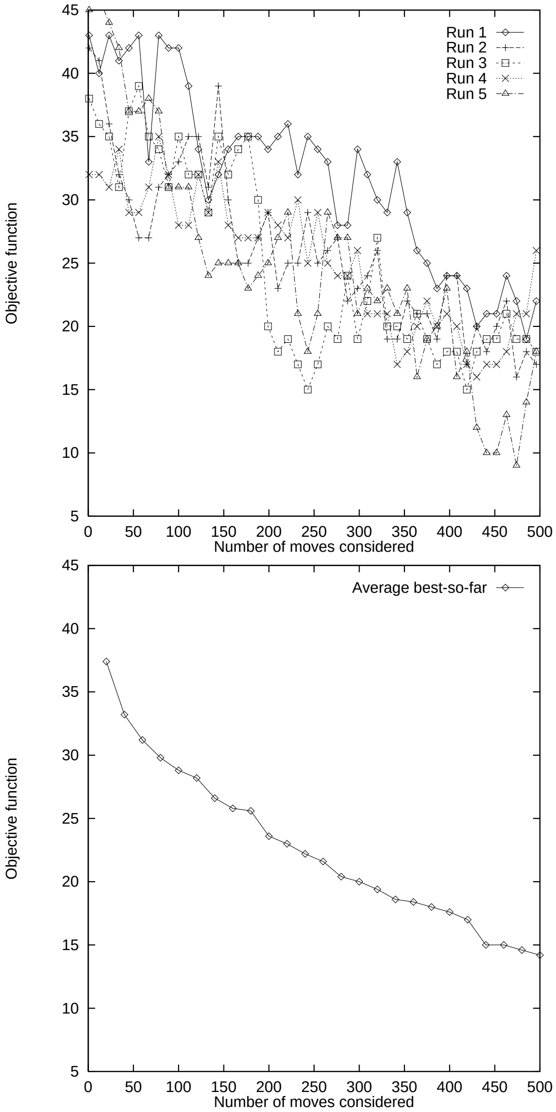

FIGURE 4.1. From a number of independent runs of an algorithm (upper left) and their best-so-far curves (upper right), a single performance curve (lower left) and box plot (lower right) are generated to summarize the algorithm's performance. Please refer to the text for a detailed explanation.

[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 $\mathrm{Obj}(x) = m$, 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. 54):

- Feature 1: $\mathrm{Obj}(x)$, the number of bins used by solution x
- Feature 2: $\mathrm{Var}(x)$, 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.)

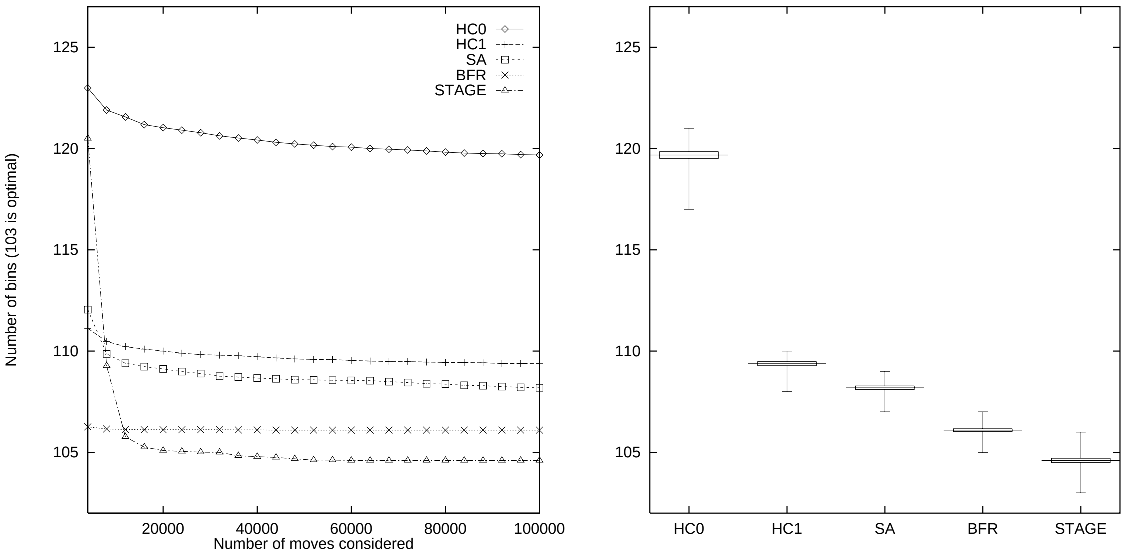

FIGURE 4.2. Bin-packing performance

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.

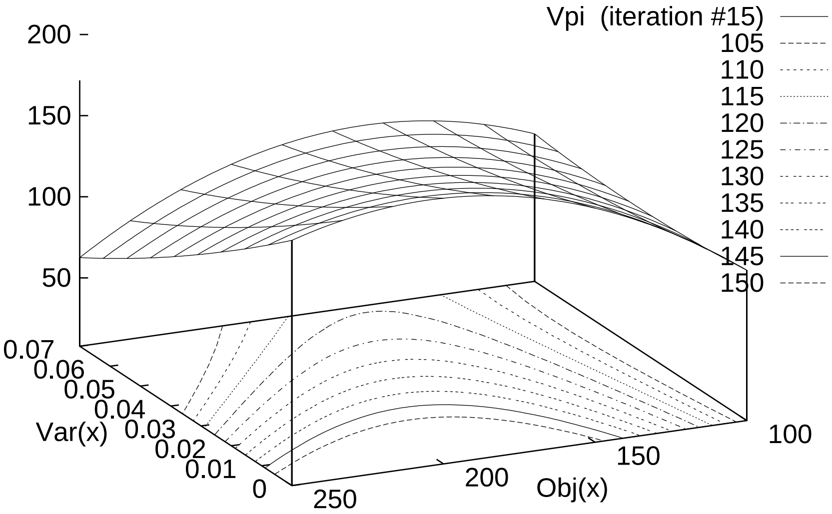

FIGURE 4.3. An evaluation function learned by STAGE on bin-packing instance u250_13. STAGE learns that the states with higher variance (wider arcs of the contour plot) are promising start states for hillclimbing.

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

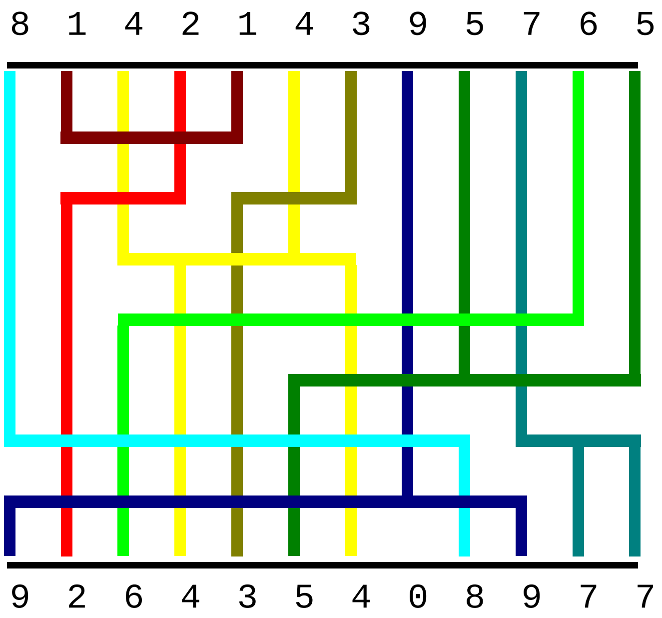

FIGURE 4.4. A small channel routing instance, shown with a solution occupying 7 horizontal tracks.

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:
>
> $$C(x) = w(x)^2 + \lambda_p \cdot p(x)^2 + \lambda_U \cdot U(x) \qquad (4.1)$$
>
> 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 $\text{Obj}(x) = w(x)$.
- 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. $\text{Obj}(x^*) = 10$. A 12-track solution is depicted in Figure 4.6.

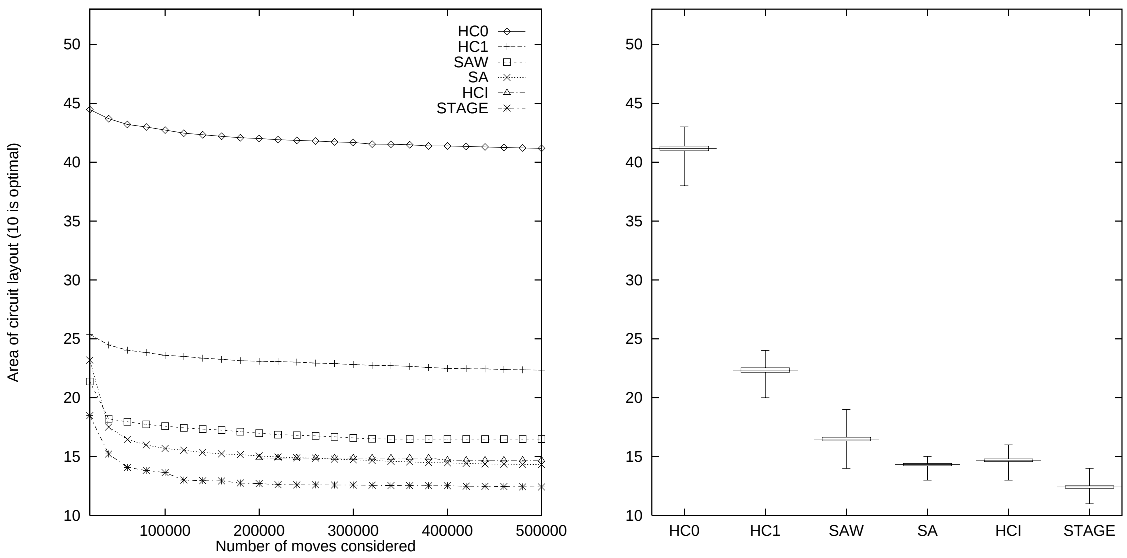

FIGURE 4.5. Channel routing performance on instance YK4

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 $\text{Obj}(x) = w(x)$ 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) mean	best	worst	≈ time	moves 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

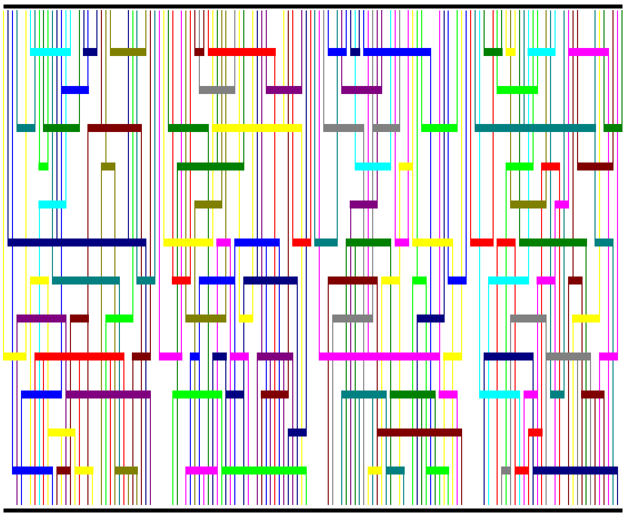

FIGURE 4.6. A 12-track solution found by STAGE on instance YK4

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

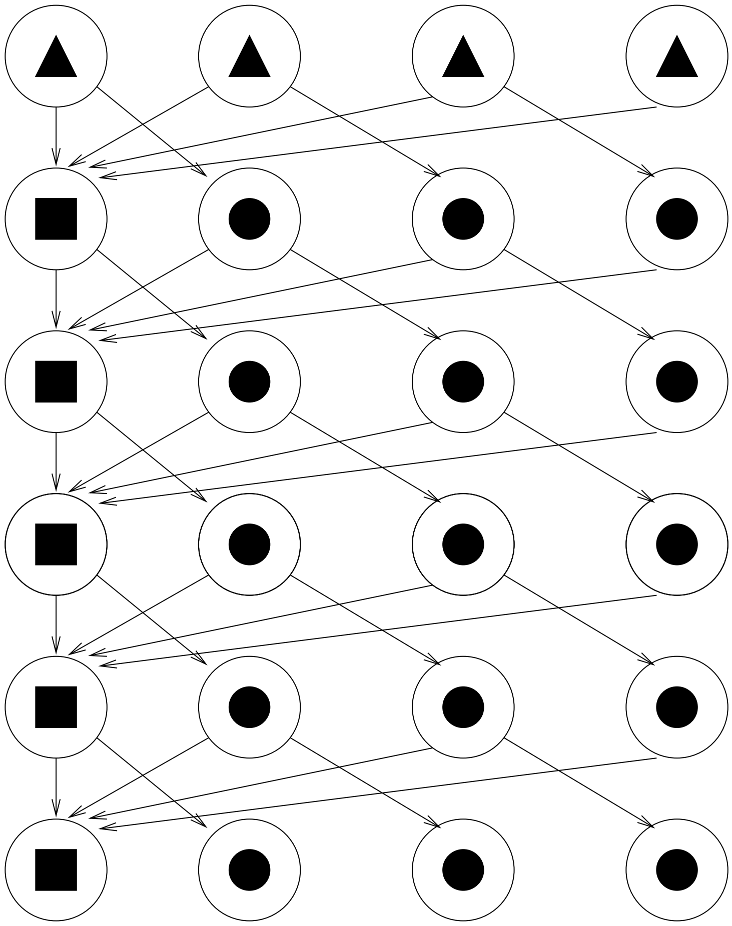

FIGURE 4.7. The SYNTH125K dataset was generated by this Bayes net (from [Moore and Lee 98]). All 24 attributes are binary. There are three kinds of nodes. The nodes marked with triangles are generated with $P(a_i = 0) = 0.8$, $P(a_i = 1) = 0.2$. The square nodes are deterministic. A square node takes 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.

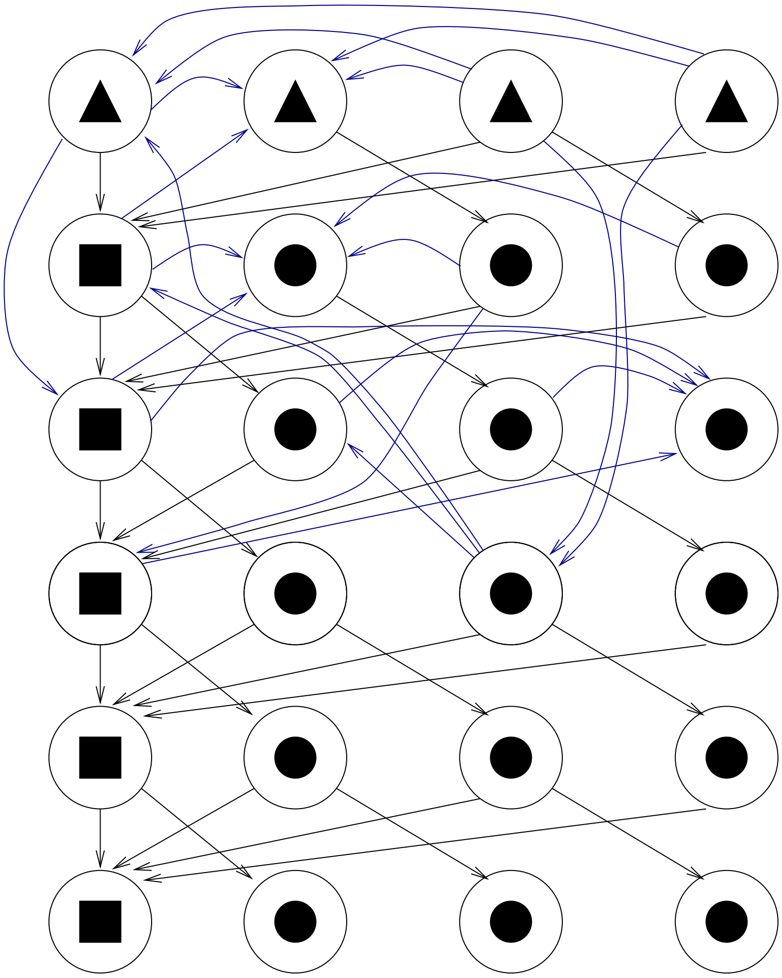

FIGURE 4.8. A network structure learned by a sample run of STAGE from the SYNTH125K dataset. Its Obj score is 719074. By comparison, the actual network that was used to generate the data (shown earlier in Figure 4.7) scores 718641. Only two edges from the generator net are missing from the learned net. 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%

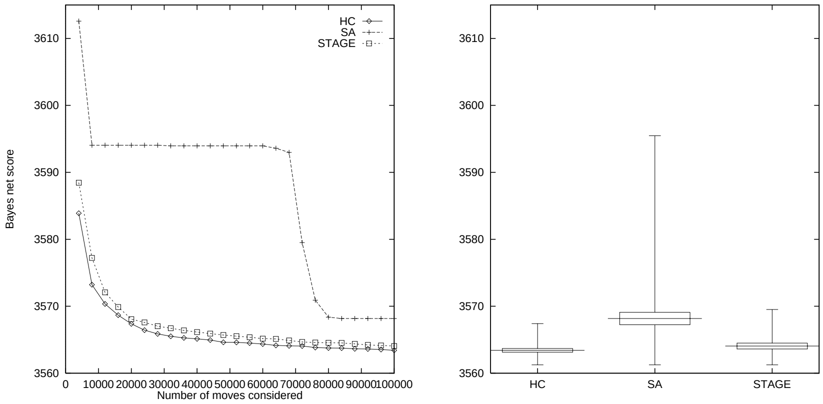

FIGURE 4.9. Bayes net performance on instance MPG

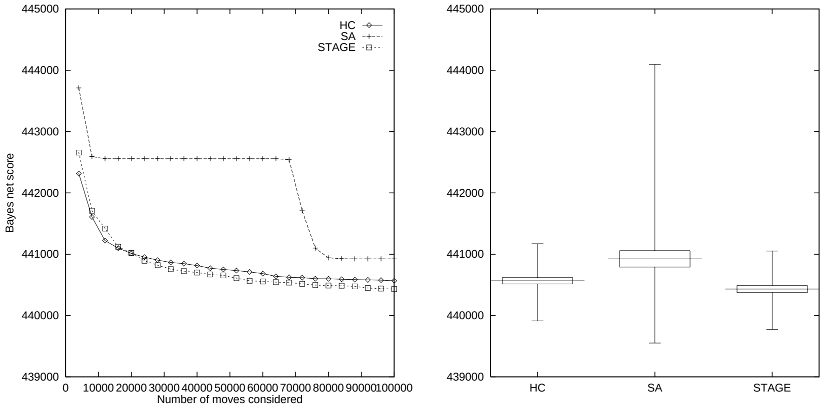

FIGURE 4.10. Bayes net performance on instance ADULT2

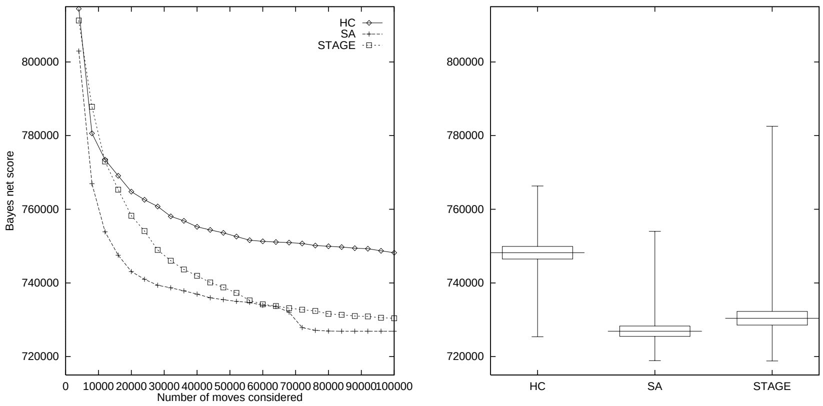

FIGURE 4.11. Bayes net performance on instance SYNTH125K

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-

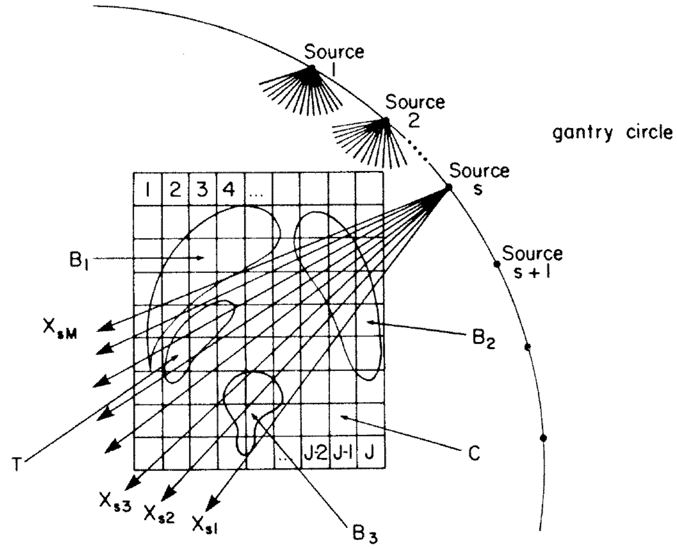

FIGURE 4.12. Radiotherapy treatment planning (from [Censor *et al.* 88])

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.

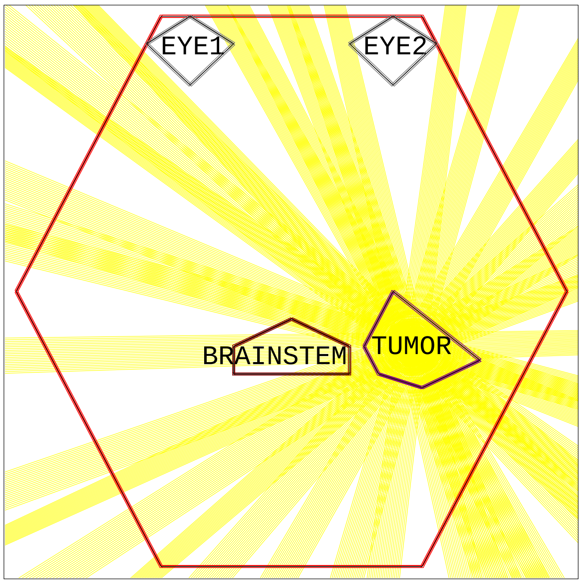

FIGURE 4.13. Radiotherapy instance 5E

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)			≈	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

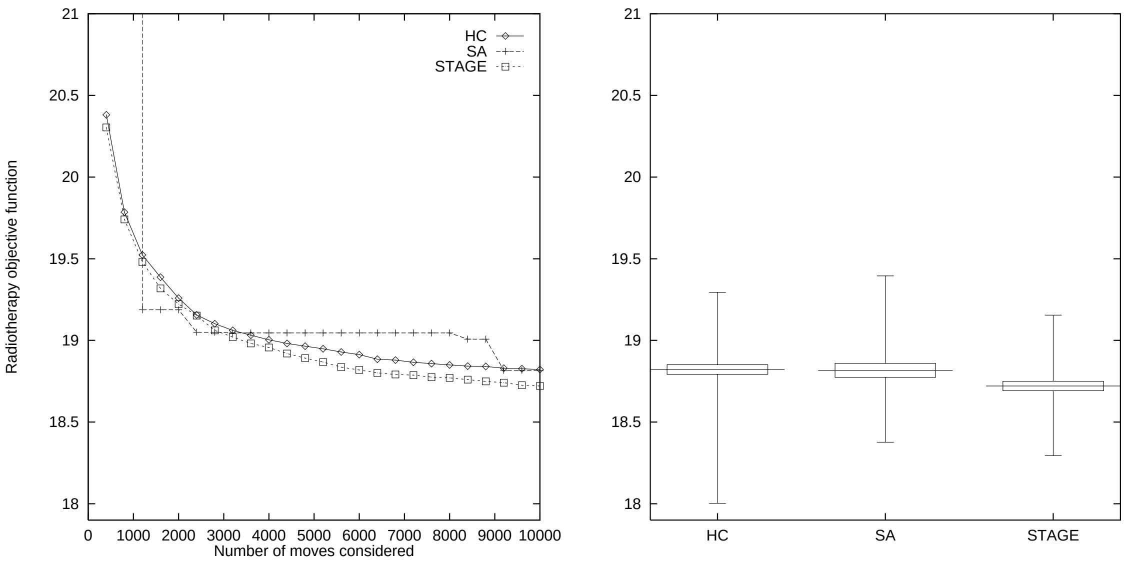

FIGURE 4.14. Radiotherapy performance on instance 5E

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

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

where $\Delta\text{Area}(x)$ 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

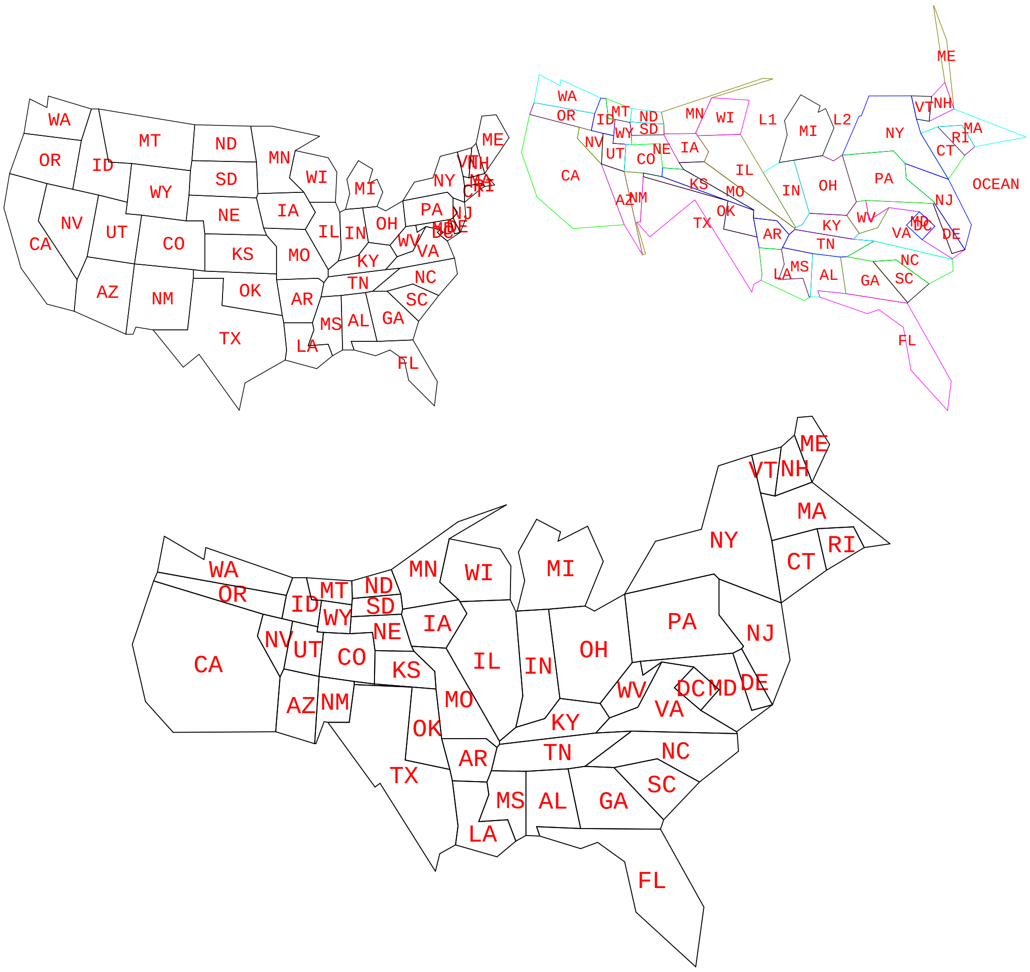

FIGURE 4.15. Cartograms of the continental U.S. Each state's target area is proportional to its electoral vote for U.S. President. The undistorted U.S. map (top left) has zero penalty for state shapes and orientations but a large penalty for state areas, so $\mathrm{Obj}(x) = 525.7$. Hillclimbing produces 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)			≈	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

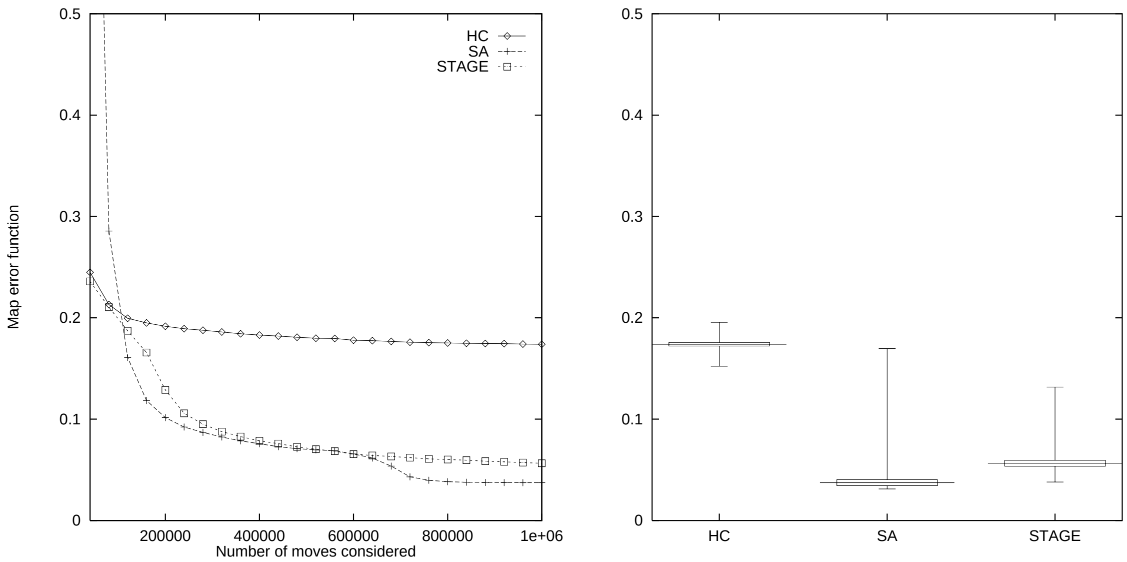

FIGURE 4.16. Cartogram performance on instance US49

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 62, 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

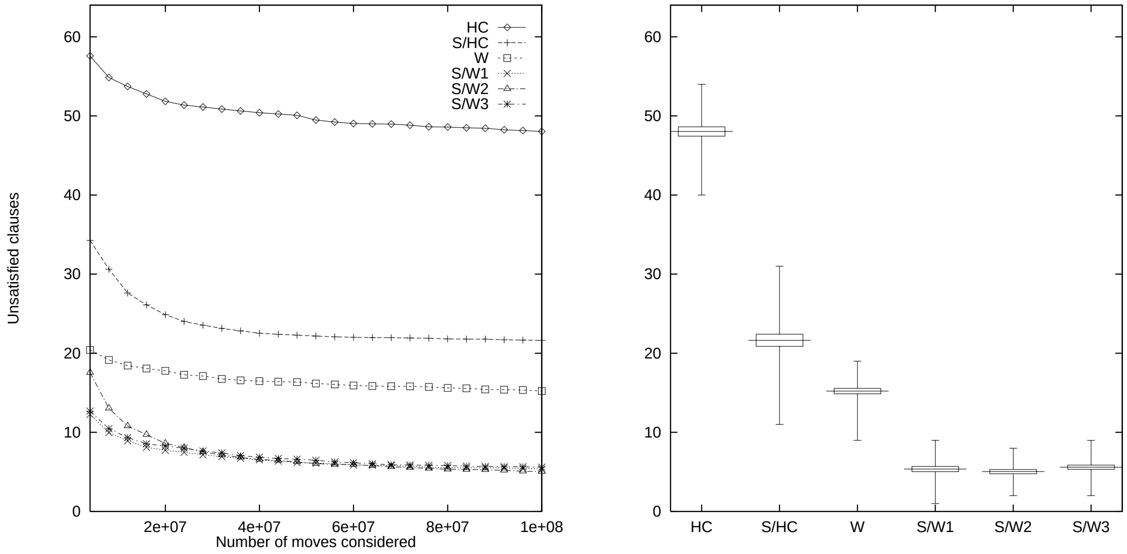

FIGURE 4.17. Satisfiability: main results on instance par32-1.cnf

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 = 2×10^9 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 $\mathrm{Obj}(x) = -\mathrm{Score}(x)/1000$. (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.

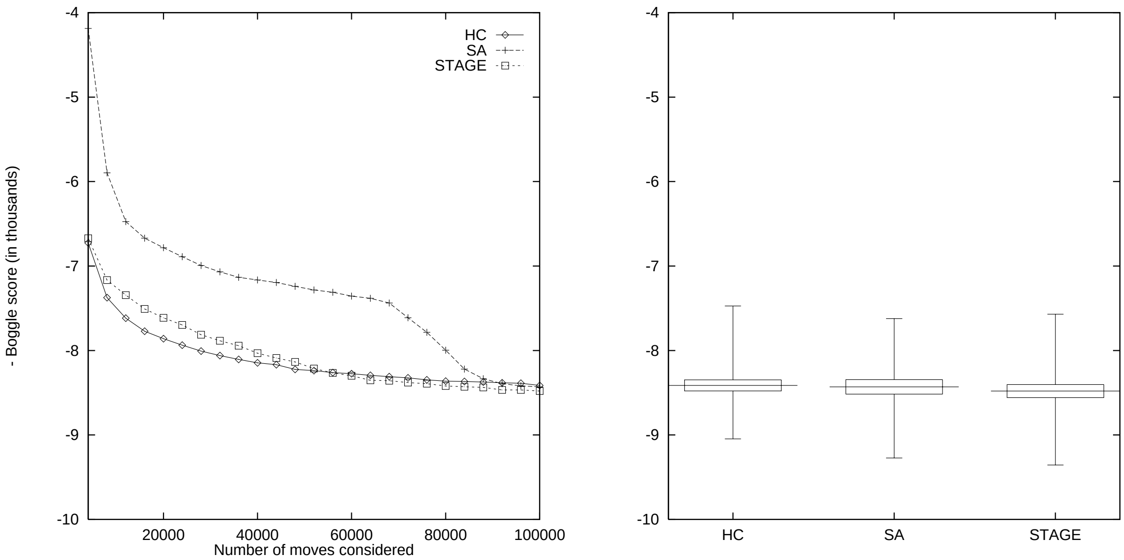

FIGURE 4.19. Performance on the Boggle domain

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.