Randomized Local Search as Successive Estimation of Probability Densities

Charles L. Isbell, Jr.

Abstract

In many optimization problems, the set of optimal solutions exhibit complex relationships between different input parameters. For example, experience may establish that a subset of parameters should take on particular values. Similarly, experience may tell us that certain parameters are closely related and should not be explored independently. Any search of the optimization landscape should take advantage of these relationships. We present MIMIC, a framework in which we analyze the global structure of the optimization landscape. A novel and efficient algorithm for the estimation of the structure is derived. We use knowledge of the structure to guide a randomized search through the solution space and, in turn, to refine our estimate of the structure.

In this paper, we derive the MIMIC framework and describe the resulting algorithm in detail. We draw connections between this framework and other techniques for probability estimation. Finally, we present several results for cost functions of varying difficulty. We show that this technique obtains significant speed gains over other randomized optimization procedures.

This handout is based on (DeBonet, Isbell and Viola, 1996).

1 Introduction

Given a cost function $C(x)$, we may search for the minimal x in many ways. Techniques that use variations of gradient descent are perhaps the most popular; however, these techniques fail when there are many local minima that are far from optimal. In this case, the search must either include a brute-force component or incorporate randomization. Classical random searches include Simulated Annealing (SA) and Genetic Algorithms (GAs).

During the optimization process, many thousands or perhaps millions of samples of $C(x)$ are evaluated. Most optimization algorithms take these millions of pieces of information and compress them into a single point x—the current estimate of the solution (one notable exception are GAs; we will return to them shortly). We can imagine splitting the search process into two parts, both taking $t/2$ time steps (see Figure 1). Both parts are structurally identical: taking a description of $C()$, they start their search from some initial point. The sole benefit enjoyed by the second part of the search over the first is that the initial point is perhaps closer to an optimum. Intuitively, there must be some additional information that could be communicated from the first half of the search, if only to warn the second half about avoidable mistakes and pitfalls.

We present an optimization algorithm called Mutual-Information-Maximizing Input Clustering (MIMIC). It explicitly communicates information about the cost function obtained from one iteration of the search to later iterations of the search. It does this in an efficient and principled way. There are two main components of MIMIC: first, a randomized optimization algorithm that samples from those regions of the input space most likely to contain the optima for $C()$; second, an effective density estimator that can be used to capture a wide variety of structure on the input space, yet is computable from simple second order statistics of the data. MIMIC's results on several cost functions indicate an order of magnitude improvement in performance over related approaches.

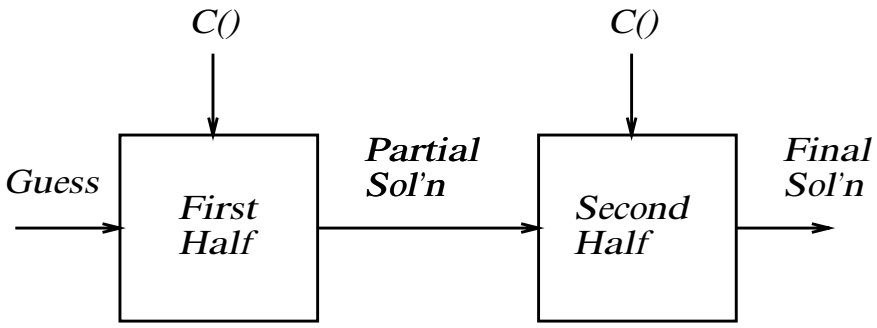

Figure 1: The problem with most optimization algorithms is that much of the work done in the first half of the search does not directly benefit the second half of the search.

2 Related Work

2.1 Genetic Algorithms

Genetic algorithms (GAs) are a family of adaptive systems motivated by theories of evolutionary biology. While many well known optimization procedures do not represent the structure of the optimization landscape, GAs attempt to capture this structure implictly by using an ad hoc embedding of input parameters onto a line (known as the chromosome). Candidate optima are realized as a set of vectors (usually of bits). An initial "population" of these vectors is chosen at random and allowed to "evolve" over several generations according to the fitness value (i.e. $C()$) derived from the environment. Methods for evolving new generations are varied, but in general fall into three categories: selection, mutation and crossover.

Selection uses the relative fitness of a candidate to select how likely it is to return in the subsequent population. Mutations are small random changes in the components of a candidate and are used to preserve diversity and facilitate exploration of the space. Crossover operators combine partial solutions from pairs of candidates. New candidates are created by merging subsets of the pairs.

The crossover operator is the operator that most distinguishes GAs from other optimization methods. The intent of the crossover operation is to preserve and propagate groups of parameters that *might* be partially responsible for generating favorable evaluations. Even when such groups exist, many of the offspring that are generated do not preserve the structure of these groups because the choice of crossover point is random.

In problems where the benefit of a parameter is completely independent of the value of all other parameters, the population simply encodes information about the probability distribution over each parameter. In this case, the crossover operation is equivalent to sampling from this distribution; the more crossovers the better the sample. Even in problems where fitness is obtained through the combined effects of clusters of inputs, the GA crossover operation is beneficial only when its randomly chosen clusters happen to closely match the underlying structure of the problem or can be easily determined beforehand by the programmer. Because of the rarity of such a fortuitous occurrence, the benefit of the crossover operation is greatly diminished. As as result, GAs have a checkered history in function optimization [3, 5]. One of our goals is to incorporate insights from GAs in a principled optimization framework.

2.2 Population Based Incremental Learning

There have been other attempts to better realize the advantages of GAs. Population Based Incremental Learning (PBIL) attempts to recast the notion of a candidate population by replacing it with a single probability distribution. In the original work on PBIL [1], this distribution was realized as a probability vector. Each element of the vector is the probability that a particular bit in a solution is on. During the learning process, the probability vector can be thought of as a simple model of the optimization landscape. Bits whose values are firmly established have probabilities that are close to 1 or 0. Those that are still unknown have probabilities close to 0.5.

PBIL begins with a uniform distribution for each bit (i.e., each element in the probability vector is 0.5) and generates several candidate solutions according to that distribution. The fitness of each solution is then used to update the probability vector. In the simplest case, the best solution is used to update the vector, p:

$$p = (1 - \alpha)p + \alpha y^*, \tag{1}$$

where y^* is the best candidate and α is the learning rate. Variations include movement towards the top few best solutions and away from the worst few solutions, also according to this learning rule. In any case, we can rewrite the update rule as:

$$p = p + \alpha(y^* - p), \tag{2}$$

which is, in fact, the generalized LMS rule:

$$p = p + \alpha(y^*(x) - y_p(x))\nabla_p y_p(x). \tag{3}$$

The generalized LMS rule is used in supervised learning and describes how to update a set of parameters, p, in order to minimize the mean squared error between an estimate of a target function, $y_p(x)$ and the actual target function $y^*(x)$. In this case, $y_p(x)$ is determined probablistically, so we can replace both it and $\nabla_p y_p(x)$ with their expected values. $E[y_p(x)] = p$ by construction. And because each component of p, p_i, is treated independently, $E[\nabla_{p_i} y_{p_i}(x)] = 1$ for each p_i.

When it is the *structure* of the components of a candidate rather than the particular values of the components that determines how it fares, it can be difficult to move this representation towards a viable solution. Nevertheless, even in these sorts of problems PBIL often out-performs GAs because those algorithms are hindered by the fact that random crossovers are infrequently beneficial. Recently, PBIL has been extended to include a more complex probability model [1] similar to the one that we will describe in section 3.

2.3 Sabes and Jordan

A very distinct, but related technique was proposed by Sabes and Jordan for a reinforcement learning task [8]. In their framework, the learner must generate actions so that a reinforcement function can be completely explored. Simultaneously, the learner must exploit what it has learned so as to optimize the long-term reward. Sabes and Jordan chose to construct a Boltzmann distribution from the reinforcement function: $p(x) = \frac{\exp(\frac{R(x)}{T})}{Z_T}$ where $R(x)$ is the reinforcement function for action X, T is the temperature, and Z_T is a normalization factor. They use this distribution to generate actions. At high temperatures this distribution approaches the uniform distribution, and results in random exploration of $R()$. At low temperatures only those actions which garner large reinforcement are generated. By reducing T, the learner progresses from an initially randomized search to a more directed search about the true optimal action. Interestingly, their estimate for $p(x)$ is to some extent a model of the optimization landscape which is constructed during the learning process. To our knowledge, Sabes and Jordan have neither attempted optimization over high dimensional spaces, nor attempted to fit $p(x)$ with a complex model.

3 The General MIMIC algorithm

Knowing nothing about $C(x)$ or its structure, it might not be unreasonable to search for, say, its minimum by generating points from a uniform distribution over the inputs x. Unfortunately, such a search allows none of the information generated by previous samples to effect the generation of subsequent samples and, so, will likely take a very long time. While this state of affairs seems unacceptable, it is not too far removed from many optimization algorithms, which only remember the best point seen.

We might be much better off if we could generate samples from a distribution, $p^\theta(x)$, that is uniformly distributed over those x's where $C(x) \leq \theta$ and has a probability of 0 elsewhere. For example, if we had access to $p^{\theta_M}(x)$ for $\theta_M = \min_x C(x)$ a single sample would be sufficient to find an optimum.

This insight suggests a process of successive approximation: given a collection of points for which $C(x) \leq \theta_0$, construct a density estimator for $p^{\theta_0}()$. From this density estimator generate additional samples, establish a new threshold, $\theta_1 = \theta_0 - \epsilon$, and construct a new density estimator. Repeat this process until the values of $C(x)$ cease to improve.

This is precisely the approach that MIMIC takes. MIMIC begins by generating a random population of candidates choosen uniformly from the input space. From this population the median fitness is extracted and is denoted θ_0. The algorithm then proceeds:

1. Update the parameters of the density estimator of $p^{\theta_i}(x)$ from a sample.
2. Generate more samples from the distribution $p^{\theta_i}(x)$.
3. Set θ_{i+1} equal to the Nth percentile of the data. Retain only the points less than θ_{i+1}. Repeat.

The validity of this approach is dependent on two critical assumptions: $p^{\theta}(x)$ can be successfully approximated with a finite amount of data; and $p^{\theta-\epsilon}(x)$ and $p^{\theta}(x))$ are similar enough that samples from $p^{\theta}(x)$ are also likely to be samples from $p^{\theta-\epsilon}(x)$ (we will formally define "similar" below). Bounds on these conditions can be used to prove convergence in a finite number of successive approximation steps.

The performance of this approach is dependent on the nature of the density approximator used. We have chosen to estimate the conditional distributions for every pair of parameters as well as the unconditional distributions for each parameter, a total of $\mathcal{O}(n^2)$ numbers. In the next section we will show how we use these conditionals to construct a distribution which is most similar to the true joint distribution. Such an approximator is capable of representing clusters of highly related parameters. This is similar to the intuitive behavior of crossover, but this representation is strictly more powerful. More importantly, our clusters are *learned* from the data, and are not pre-defined by the programmer.

4 Estimating Probability Distributions with Conditional Probabilities

The joint probability distribution over a set of random variables, $X = \{X_i\}$, is:

$$p(X) = p(X_1|X_2 \ldots X_n)p(X_2|X_3 \ldots X_n) \ldots p(X_{n-1}|X_n)p(X_n). \tag{4}$$

Given only pairwise conditional probabilities, $p(X_i|X_j)$ and unconditional probabilities, $p(X_i)$, we are faced with the task of generating samples that match as closely as possible the true joint distribution, $p(X)$. It is clearly not possible to capture all possible joint distributions of n variables using only the unconditional and pairwise conditional probabilities; however, we would like to describe any arbitrary joint distribution as closely as possible. Below, we derive an algorithm for choosing such a description.

4.1 Dependency Trees

We must first decide upon a more tractable representation of our probability distribution. We define a class of probability distributions, $\hat{p}_\pi(X)$:

$$\hat{p}_\pi(X) = \prod_i p(X_i|X_{\pi_i}) \tag{5}$$

where π is an n-dimensional vector of the numbers between 1 and n, $\pi = i_1 i_2 \ldots i_n$. The vector π maps every parameter to a single "parent" parameter but may map any parent to many "children". By necessity, $\hat{p}_\pi(X)$ cannot allow any cycles (here $p(X_i|X_i) \equiv p(X_i)$), so π defines a directed acyclic graph. Graphs used in this manner are known as directed dependency trees (in previous work we have used directed chains). These are useful structures because they are among the richest that depend soley on conditional probabilities while admiting tractable construction and sampling algorithms.

Our goal is to choose the π that maximizes the agreement between $\hat{p}_\pi(X)$ and the true distribution $p(X)$. The agreement between two distributions can be measured by the Kullback-Liebler divergence:

$$\begin{aligned} D(p||\hat{p}_\pi) &= \int_{\mathcal{X}} p[\log p - \log \hat{p}_\pi] dX \\ &= E_p[\log p] - E_p[\log \hat{p}_\pi] \\ &= -h(p) - E_p[\log \prod_i p(X_i|X_{\pi_i})] \\ &= -h(p) + \sum_i h(X_i|X_{\pi_i}). \end{aligned}$$

This divergence is always non-negative, with equality only in the case where $\hat{p}(\pi)$ and $p(X)$ are identical distributions. The optimal π is defined as the one that minimizes this divergence. For a distribution that can be completely described by pairwise conditional probabilities, the optimal π will generate a distribution that will be identical to the true distribution. Insofar as the true distribution cannot be captured this way, the optimal $\hat{p}_\pi(X)$ will diverge from that distribution.

The first term in the divergence does not depend on π. Therefore, the cost function, $J_\pi(X)$, we wish to minimize is:

$$J_\pi(X) = \sum_i h(X_i|X_{\pi_i}). \tag{6}$$

where the optimal π is the one that produces the lowest pairwise entropy with respect to the true distribution.

There is a simple algorithm for finding the optimal π. Imagine that each of the parameters, X_i, is a node in a graph. For each pair of nodes, X_i and X_j, there is an edge (i, j) with weight $h(i|j)$. Clearly, finding the minimum spanning tree (MST) of that graph is eqiuvalent to finding the optimal π.

Normally, we can use Prim's algorithm to find the MST of a graph in time $\mathcal{O}(n^2)$; however, Prim's algorithm only works for undirected graphs. We could instead apply Tarjan's algorithm [9] for finding the MST of a directed graph. Tarjan showed that if the graph is dense, as in our case, this algorithm can be made to run in time $\mathcal{O}(n^2)$. Unfortunately, there are technical difficulties in implementing this algorithm.

Instead, we will simplify our problem by finding an equivalent undirected tree. This is done by simply noting that we can change our cost function, $J_\pi(X)$, by adding another term:

$$\begin{aligned} J'_\pi(X) &= -\sum_i h(X_i) + \sum_i h(X_i|X_{\pi_i}) \\ &= -\sum_i I(X_i; X_{\pi_i}) \end{aligned}$$

where $I(a; b)$ is the mutual information between the two random variables a and b. Because $-\sum_i h(X_i)$ does not depend on π, minimizing $J'_\pi(X)$ is equivalent to minimizing $J_\pi(X)$. Further, mutual information is a symmetric measure, so this transforms our previously directed graph into an undirected graph where each edge (i, j) now has weight $-I(X_i; X_j)$.

A straightforward application of Prim's algorithm finds an optimal undirected dependency tree in time $\mathcal{O}(n^2)$:

1. Construct a graph with nodes X_i.
2. For each unordered pair, i and j add an edge (i, j) with weight $-\hat{I}(X_i; X_j)$, where $\hat{I}()$ is the empirical mutual information.
3. Use Prim's MST algorithm to find π.

A naive implementation of this approach results in the use of $\mathcal{O}(n^2)$ space as well. Fortunately, this is not a requirement, as we shall see in section 5.

4.2 Sampling from Dependency Trees

Given the optimal dependency tree, π, we can sample from it trivially:

1. Begin at any node, X_i, in the tree. This is the root. Pick its value according to it's empirical probability $\hat{p}(X_i)$.
2. Perform a depth-first traversal of the tree from the root. For each node X_i, choose its value according to the empirical conditional probability $\hat{p}(X_i, X_{\pi_i})$.

This method generates a single value X. Because the tree is undirected, it does not matter where we begin our traversal. All traversals produce equivalent distributions.

4.3 Refining Samples Generated by MIMIC

In practice, we cannot know how well we have estimated and sampled from $p^{\theta_i}(x)$. Poor estimation might occur for several reasons. For example, it may be the case the true distribution we care about is not easily described by a dependency tree. If this is the case, we may generate many samples that are not from $p^{\theta_i}(x)$. Similarly, we may generate samples that cover only a small subset of the samples that would be generated by $p^{\theta_i}(x)$, limiting our ability to search the optimization landscape fully.

We can at least address these problems by adding two simple extensions to our sampling algorithm. First, we can simply ignore any sample generated by the tree that clearly does not belong to p^{θ_i}. That is to say, we will ignore any samples, x, where $C(x) > \theta_i$. Second, we can add a small amount of noise, ϵ, to our conditional and unconditional probability estimates, ensuring that we will generate all points with some non-zero probability. As a practical matter, ϵ cannot be too large; otherwise, we will gain nothing from our search. Despite the addition of noise, we will eventually cease generating points that belong to $p^{\theta_i}(x)$. Even if such points exist, the probability of generating such a point may be so low that it is very unlikely that we will generate it in a reasonable amount of time. When this occurs we can simply assume that we have converged. Alternatively, we could use this final distribution as a new initial distribution and reset θ in the hopes that we might explore another part of the space.

While both approaches have improved performance in our experiments, they may seem a little *ad-hoc*. In point of fact, they are very strongly related to *rejection sampling* and *importance sampling* (see [7] for a detailed discussion). In rejection sampling, one assumes that there is a true distribution, $p(x)$, that can be evaluated but is difficult to sample from and an estimate, $\hat{p}(x)$, that is easy to sample from (and hopefully approximates $p(x)$). Rejection sampling assumes that there exists some constant c such that $p(x) \leq c\hat{p}(x)$ for all x. To generate a point for $p(x)$, one repeatedly generates a point from $\hat{p}(x)$, accepting it as a true point with probability $\frac{p(x)}{c\hat{p}(x)}$.

In our case, the true distribution always yields zero or $\frac{1}{Z}$ (for some suitable normalizing constant Z) for a point x, so the rejection process is very simple. Further, by adding noise to our estimates, we guarantee that there exists some c such that $p(x) \leq c\hat{p}(x)$ for all x. It can be shown that rejection sampling will indeed sample from the true distribution; however, rejection sampling is not always a practical solution. Indeed, to use it properly, we would have set c very high ($\mathcal{O}(1/\epsilon^n)$), leading to an unacceptably high rejection rate.

Importance sampling requires only that the density estimate be non-zero wherever the true function is non-zero (again, guaranteed by adding noise). Given a true distribution $p(x)$ and an estimate of that distribution, $\hat{p}(x)$, this procedure weights functions of x by $\frac{p(x)}{\hat{p}(x)}$ (suitably normalized). Again, the construction of our true distribution means that we will ignore values above θ.

Both rejection and importance sampling are basic techniques in Monte Carlo sampling. Both can fail when the estimate of the distribution to be approximated does not have high probability where the true distribution does. On the other hand, both are known to be helpful when combined with a search for such areas of high probability. This is precisely what MIMIC does.

5 Computational and Space Issues

It is worthwhile to understand the computational and space complexities of MIMIC and similar approaches to optimization. For MIMIC, if we generate m n-dimensional samples. Several costs arise *per iteration*:

1. Generated samples must be stored. This requires space $\mathcal{O}(mn)$.

2. Samples must be sorted. This requires time $\mathcal{O}(m \log m)$.

3. A minimum space tree must be computed. This requires time $\mathcal{O}(n^2)$. The MST itself requires space $\mathcal{O}(n)$.

4. If the empirical probabilities are stored (i.e. the graph of mutual information weights is built explicitly), this also requires space $\mathcal{O}(n^2)$. On the other hand, the weights can be computed on-the-fly from the stored samples, so storing the entire graph is not necessary.

5. Samples must be generated. This requires time $\mathcal{O}(mn)$.

5.1 Time Complexity

During any given iteration, techniques like GA's and randomized gradient descent procedures do not have the same space and time requirements: they do not construct minimum spanning trees, nor make any use of conditional probabilities. On the other hand, in the course of optimization, these techniques perform many, many more iterations (as we shall see in section 6) and evaluate $C(x)$ many, many more times.

The basic trade-off, then, is between performing a few computationally-intensive iterations and many computationally-light iterations. When evaluating $C(x)$ is computationally cheap, we do not pay much of a price for performing many iterations. Indeed, under the right circumstances, this allows us to search a great deal of the space. Clearly, if evaluating $C(x)$ is free, we may as well evaluate every x possible. On the other hand, when evaluations of $C(x)$ are very costly, we will be unable to search most of the space. In this case, we are much better off extracting as much information as possible from each iteration in order to best focus our search.

MIMIC clearly chooses this side of the trade-off. Therefore, it is well suited for problems where the computation of $C(x)$ is expensive.

5.2 Space Complexity

MIMIC estimates $p(X_i|X_j)$ for all pairs X_i, X_j as well as $p(X_i)$ for each X_i. Because MIMIC restimates $p^{\theta_i}(x)$ each iteration, there is no need to actually store all $\mathcal{O}(n^2)$ numbers. By contrast, algorithms that incrementally update their estimates of the conditional probabilities must store these numbers. This is the case for the Dependency Tree version of PBIL [2], for example.

There may be advantages to incrementally estimating conditional probabilities. Slowly moving parameters may provide a more robust search, for example. Further, it may obviate the need for some of the sampling techniques outlined in section 4.3. On the other hand, when the dimensionality of X becomes very large, the space requirements become unmanageable. For $n = 10000$ for example, space requirements for a table of single precision floating point numbers will exceed 120 megabytes. Once we allow our parameters to take on more than two values, these space requirements increase even further. If each parameter can take on five discrete values, the table will exceed 9 gigabytes! Problems of this size routinely occur in several problems where we might want to apply random optimization algorithms, such as information retrieval. As such, approaches more like MIMIC will have to be employed.

6 Experiments

To measure the performance of MIMIC, we performed several benchmark experiments and compared our results with those obtained using several standard optimization algorithms.

We will use four algorithms in our comparisons:

1. MIMIC - the algorithm above with 200 samples per iteration

2. PBIL - standard population based incremental learning

3. RHC - randomized hill climbing

4. GA - a standard genetic algorithm with single crossover and 10% mutation rate

6.1 Four Peaks

The Four Peaks problem is taken from [1]. Given an N-dimensional input vector $\vec{X}$, the four peaks evaluation function is defined as:

$$f(\vec{X}, T) = \max\left[tail(0, \vec{X}), head(1, \vec{X})\right] + R(\vec{X}, T) \tag{7}$$

where

$$tail(b, \vec{X}) = \text{number of trailing } b\text{'s in } \vec{X} \tag{8}$$

$$head(b, \vec{X}) = \text{number of leading } b\text{'s in } \vec{X} \tag{9}$$

$$R(\vec{X}, T) = \begin{cases} N & \text{if } tail(0, \vec{X}) > T \text{ and } head(1, \vec{X}) > T \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

There are two global maxima for this function. They are achieved either when there are $T+1$ leading 1's followed by all 0's or when there are $T+1$ trailing 0's preceded by all 1's. There are also two suboptimal local maxima that occur with a string of all 1's or all 0's. For large values of T, this problem becomes increasingly more difficult because the basin of attraction for the inferior local maxima become larger.

Results for running the algorithms are shown in figure 2. In all trials, T was set to be 10% of N, the total number of inputs. The MIMIC algorithm consistently maximizes the function with approximately one tenth the number of evaluations required by the second best algorithm.

6.2 Six Peaks

The Six Peaks problem is a slight variation on Four Peaks where

$$R(\vec{X}, T) = \begin{cases} N & \text{if } tail(0, x) > T \text{ and } head(1, x) > T \text{ or} \\ & \quad tail(1, x) > T \text{ and } head(0, x) > T \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

This function has two additional global maxima where there are $T+1$ leading 0's followed by all 1's or when there are $T+1$ trailing 1's preceded by all 0's. In this case, it is not the values of the candidates that is important, but their structure: the first $T+1$ positions should take on the same value, the last $T+1$ positions should take on the same value, these two groups should take on different values, and the middle positions should take on all the same value.

Results for this problem are shown in figure 3. As might be expected, PBIL performed worse than on the Four Peak problem because it tends to oscillate in the middle of the space while contradictory signals pull it back and forth. The random crossover operation of the GA occasionally was able to capture some of the underlying structure, resulting in an improved relative performance of the GA. As we expected, the MIMIC algorithm was able to capture the underlying structure of the problem, and combine information from all the maxima. Thus MIMIC consistently maximizes the Six Peaks function with approximately one fiftieth the number of evaluations required by the other algorithms.

6.3 Max K-Coloring

A graph is K-Colorable if it is possible to assign one of k colors to each of the nodes of the graph such that no adjacent nodes have the same color. Determining whether a graph is K-Colorable is known to be NP-Complete. Here, we define Max K-Coloring to be the task of finding a coloring that minimizes the number of adjacent pairs colored the same.

Results for this problem are shown in figure 4. We used a subset of graphs with a single solution (up to permutations of color) so that the optimal solution is dependent *only* on the structure of the parameters. Because of this, PBIL performs poorly. GA's perform better because *any* crossover point is representative of some of the underlying structure of the graphs used. Finally, MIMIC performs best because it is able to capture all of the structural regularity within the inputs.

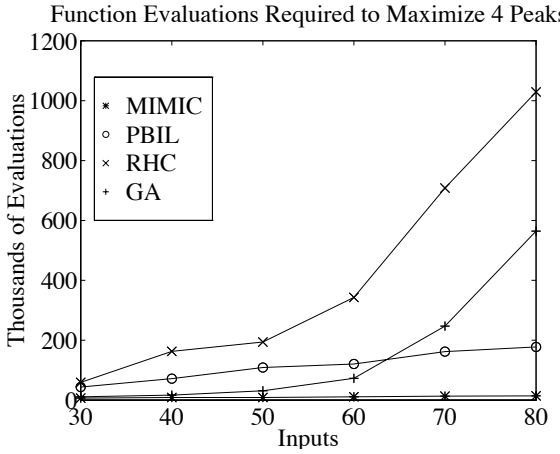

Figure 2: Number of evaluations of the Four-Peak cost function for different algorithms plotted for a variety of problems sizes.

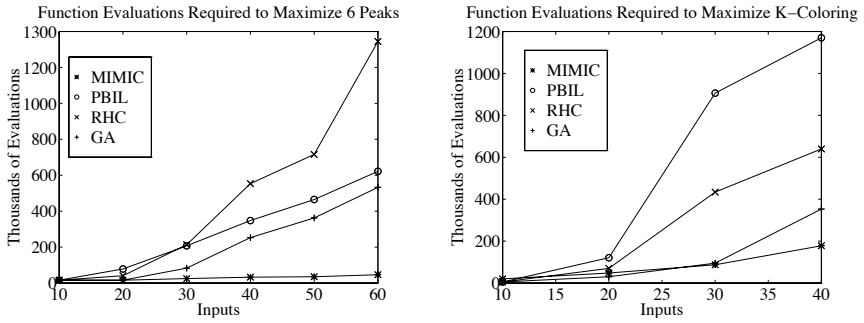

Figure 3: Number of evaluations of the Six-Peak cost function for a variety of problem sizes.

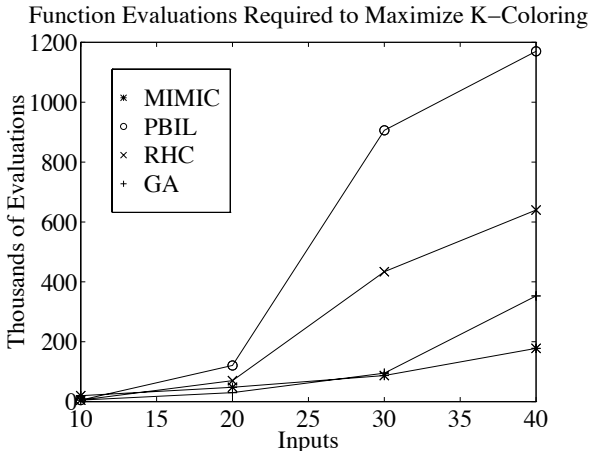

Figure 4: Number of evaluations of the K-color cost function for different algorithms plotted for a variety of problems sizes.

7 Conclusions

We have described MIMIC, a novel optimization algorithm that converges faster and more reliably than several other existing algorithms. MIMIC accomplishes this in two ways. First, it performs optimization by successively approximating the conditional distribution of the inputs given a bound on the cost function. Throughout this process, the optimum of the cost function becomes gradually more likely. As a result, MIMIC directly communicates information about the cost function from the early stages to the later stages of the search. Second, MIMIC attempts to discover common underlying structure about optima by computing second-order statistics and sampling from a distribution consistent with those statistics.

References

[1] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical report, Carnegie Mellon Univerisity, May 1995.

[2] S. Baluja and S. Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. Technical report, Carnegie Mellon Univerisity, January 1997.

[3] E. B. Baum. Toward a model of mind. Unpublished manuscript, December 1995.

[4] C. Chou and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[5] C. P. Dolan and M. G. Dyer. *Toward the Evolution of Symbols*. Hillsdale, NJ: Lawrence Erlbaum Assoc, 1987.

[6] P. V. Jeremy DeBonet, Charles Isbell. Mimic: Finding optima by estimating probability densities. In *Advances in Neural Processing Systems 1996*, 1996.

[7] R. M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical report, University of Toronto, September 1993.

[8] P. N. Sabes and M. I. Jordan. Reinforcement learning by probability matching. In M. M. David S. Touretzky and M. Perrone, editors, *Advances in Neural Information Processing*, volume 8, Denver 1995, 1995. MIT Press, Cambridge.

[9] R. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.