Assignment 2, part 1

Neural Network training through Randomized Optimization

The Skin dataset is a set of RGB points taken from portraits of human faces. Each point is labeled skin or non skin. In assignment one, I shoId that neural network trained using backpropagation of error can learn a function that labels points very effectively. In part one of this assignment, I train the same neural network using a variety of randomized optimization functions, including randomized hill climbing, simulated annealing, and genetic algorithms.

In randomized hill climbing, I begin by selecting a random set of Iights. For this implementation, I select a point near the zero vector in the space of the neural network Iights as the starting instance. The algorithm then selects a point in the neighborhood of the current instance that improves upon the fitness of the current instance, or terminates if no point in the neighborhood can improve on the current point as it has reached a peak. Two functions must now be defined for all instances, a neighborhood function and a fitness function.

The fitness function must be real valued and increasing as the suitability of the network Iights to the classification task at hand increase. To accomplish this, I select a small training set at random from the full skin dataset, and find the output of the network given the current Iights over the training set. After some experimentation, the fitness function was defined to be the count of the correctly classified instances, less the margin in the wrong direction of misclassified instances. This allows the function to increase by correctly classifying another instance, or by less incorrectly classifying an incorrectly classified instance.

The neighborhood function must return a set of instances of neural network Iights that are near to the current instance. One could choose the set of all neighbor instances offset by a unit in a single dimension, or in any combination of directions, or randomly choose a direction and a distance for any number of times. It turns out that exhaustively searching the neighborhood of all orthogonal unit vector offsets from the current point performs Ill, with the caveat that given the random initialization and the minimum step of a unit, peaks betIen steps are not reachable. Random steps may overcome this limitation, but then one must limit the neighborhood size, as there are an infinite number of neighbors that are one random vector away from the current point. This method was chosen as it was most compatible with the ABAGAIL libraries abstract class RandomizedHillClimbing.

With a fitness function defined and a neighbor function that returns a random neighbor, the rest of the implementation of randomized hill climbing was a control loop to train the neural network until stopping criteria Ire met. The stopping criteria terminate the training when no point in the neighborhood can improve on the fitness of the current point more than a minimum improvement parameter, epsilon. Results of randomized hill climbing optimization iterations for a variety of neighborhood sizes are on visualized in figures 1,2 and 3 and summarized in table 1. Classification accuracy on the validation set is plotted over fitness on the training set for 100 random starts with neighborhood sizes of 10,100 and 1000. Three clusters of fitness and accuracy are apparent in the plots.

Figure 1

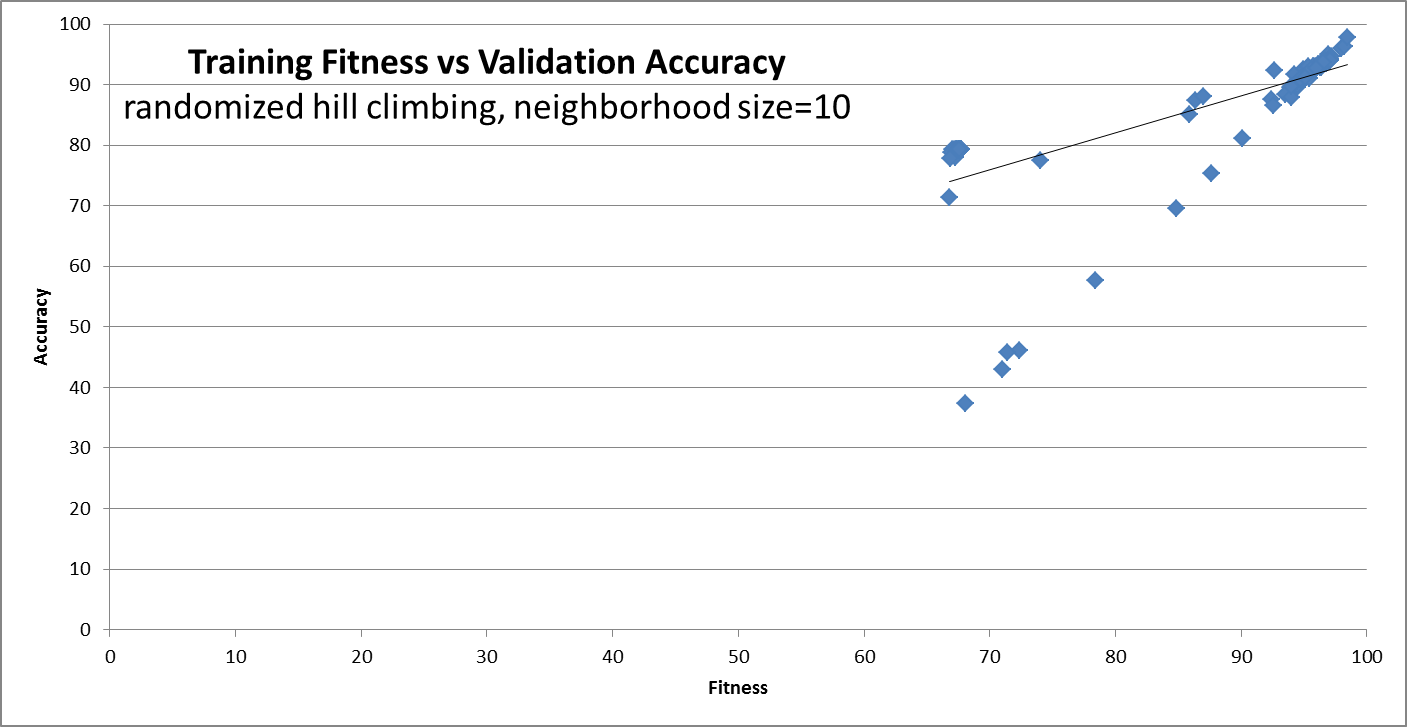


Figure 2

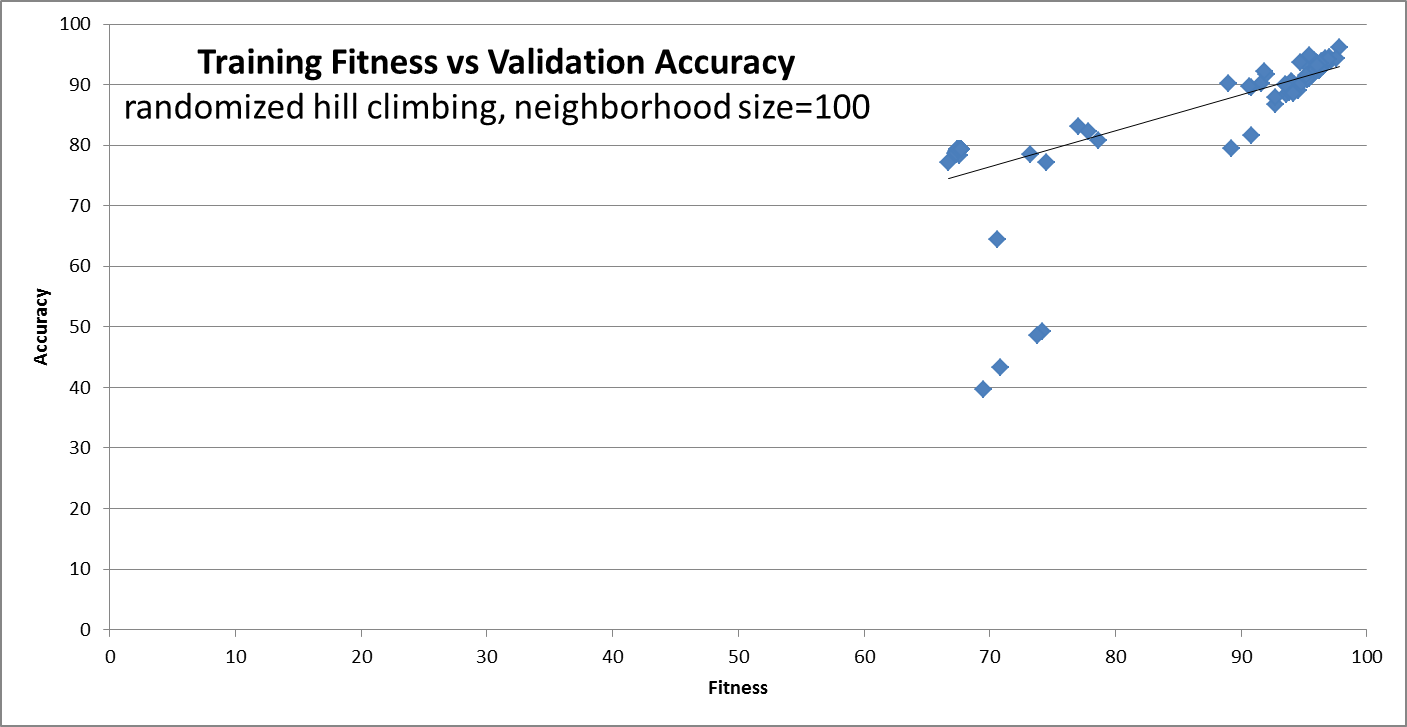


Figure 3

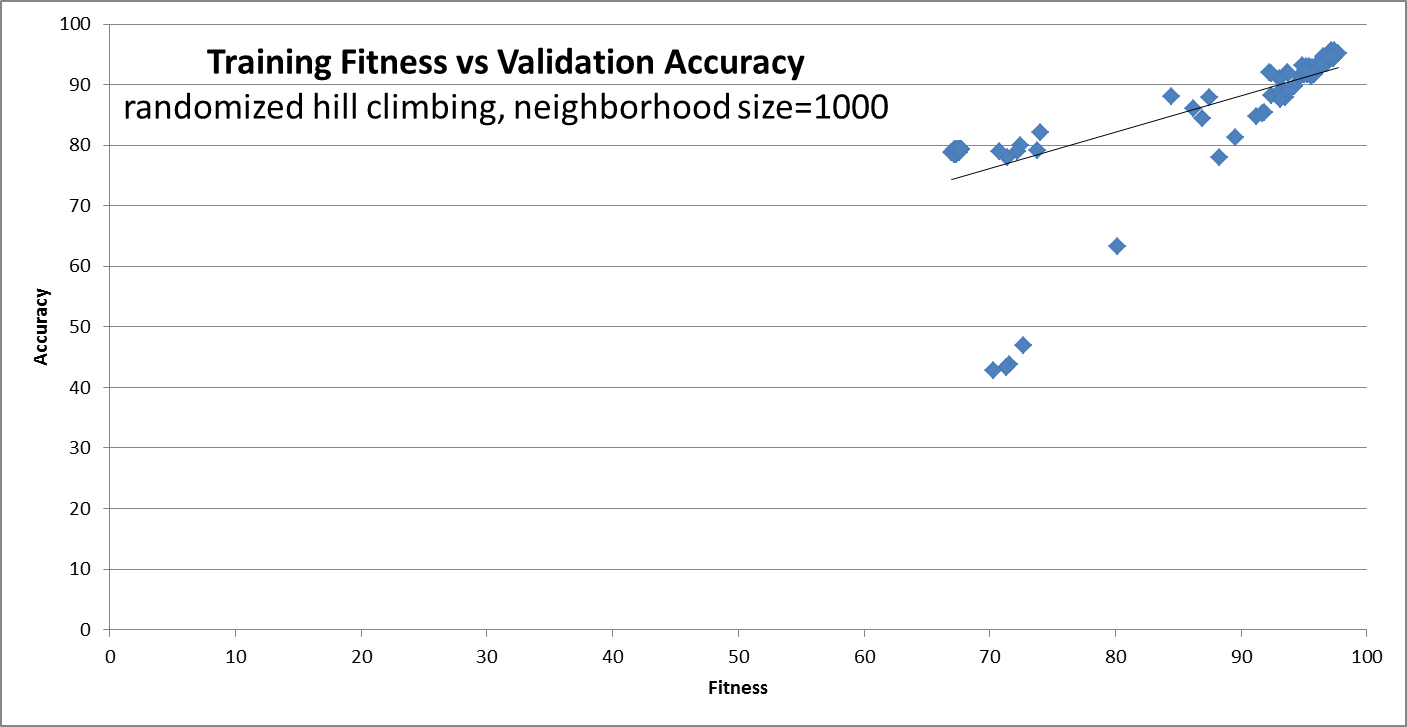


Table 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Neighborhood Size | Training time for 100 starts (sec) | Optimum fitness | Validation accuracy of optimum (%) | Standard deviation of fitness | Minimum fitness |
| 10 | 28.518 | 98.46 | 97.75 | 13.31 | 66.82 |
| 100 | 28.197 | 97.85 | 96.07 | 13.01 | 66.70 |
| 1000 | 28.065 | 97.74 | 95.17 | 12.74 | 66.96 |

As the charts and tables show, the training time did not increase noticeably while increasing the number of samples tried before stopping. There is a wide spread in fitness from 66.70 to 98.46, and the range is nearly ideantical for all neighborhood sizes. Increasing the neighborhood size parameter decreases standard deviation of fitness slightly, in turn decreasing the expected number of random starts before a suitably fit optimum is found. The validation accuracy of the optimum actually increased as neighborhood size increased, which may be a sign of overfitting the training data. Given these results, the smaller neighborhood size is preferred. It is simpler, and performed comparably to higher neighborhood sizes, with less evidence of overfitting.

Other variables that Ire not varied include the size of the training set, the minimum fitness improvement variable epsilon, and the scale parameter used for the random step. As result of the invariant scale parameter, the algorithm cannot start far from the origin. Randomized hill climbing is known to be prone to stopping in local optima that are less fit than the global optima. Simulated annealing includes a temperature parameter that allows it to step past these local optima while it is hot, and climb to the nearest peak when the process has cooled.

Simulated annealing can be built on randomized hill climbing with a few modifications. Instead of stopping criteria based solely on exhaustive neighborhood search, I add a random component that will accept a less fit instance from time to time. The temperature parameter is used to control the selection of the next instance, such that the next step is nearly random when temperature is high, and seeks fitness improvement when the temperature falls. A cooling parameter controls the rate of temperature decay with each step.

The formula used for the probability of accepting a less fit instance is taken from the ABAGAIL implementation of SimulatedAnnealing. The class itself is not used here; the encapsulation of temperature and exposure of a single train() method made it difficult to take the current temperature into account in the control loop. Instead, this algorithm is implemented with the same randomized hill climbing implementation from before, and the control loop takes into account temperature and adds the random step component.

For the starting temperature, I arbitrarily chose to use one half the maximum double precision floating point value in Java. A cooling factor of 0.1 was chosen after some experimentation. The neighborhood size of 10 was retained from the previous experiment. The results are summarized in figure 4 and table 2.

Figure 4

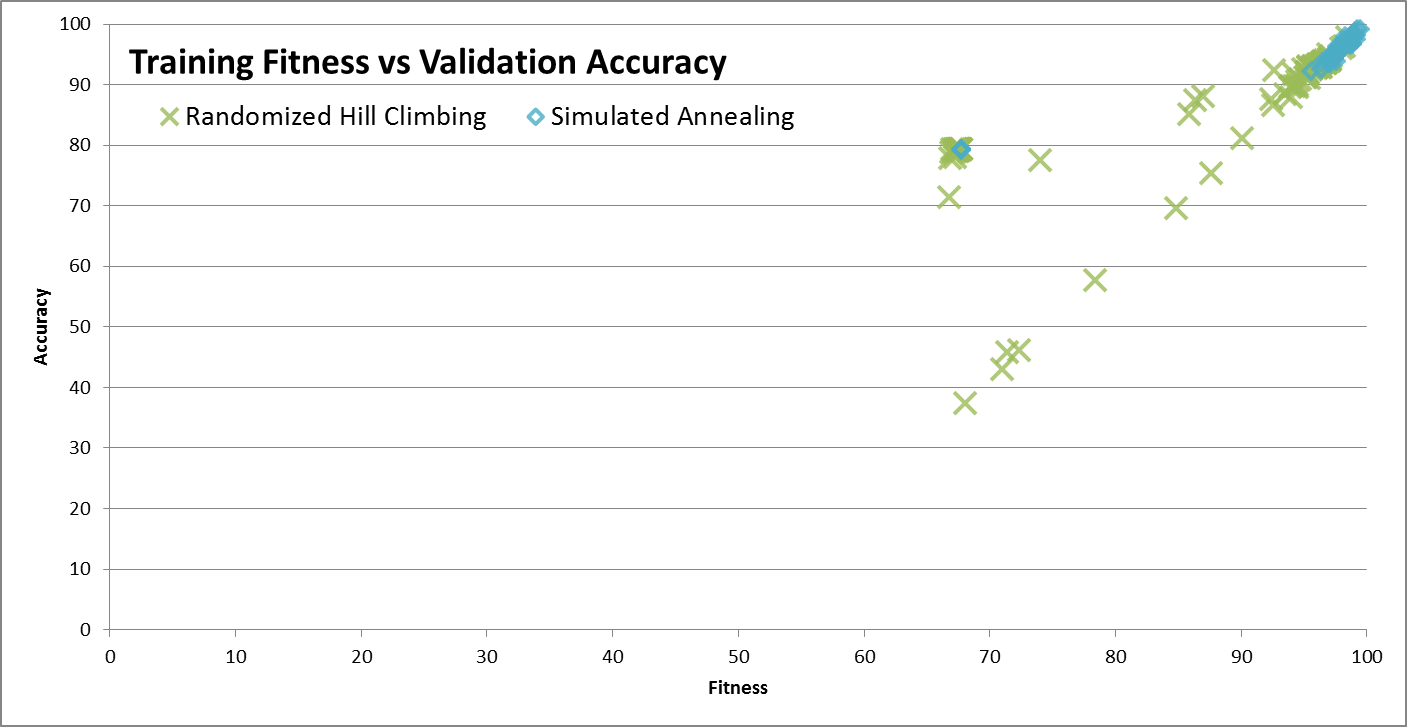


Table 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Optimization algorithm | Training time for 100 starts (sec) | Optimum fitness | Validation accuracy of optimum (%) | Standard deviation of fitness | Minimum fitness |
| Randomized hill climbing | 28.518 | 98.46 | 97.75 | 13.31 | 66.82 |
| Simulated Annealing | 57.432 | 99.53 | 99.04 | 6.61 | 67.75 |

The results of simulated annealing showed improvement over randomized hill climbing in every measure. The standard deviation of fitness was cut in half, but the training time doubled. The number of starts required to obtain a suitable optimum would be significantly reduced with simulated annealing, and this is apparent in the distribution of optima in Figure 4. Most notably, there are feIr misfit optima with high fitness but low validation accuracy found using simulated annealing. Whether this is a feature of the problem space, or a feature of the algorithm remains to be seen. If the increase in time complexity is affordable, or the cost of random starts is sufficiently high, then simulated annealing is preferable to randomized hill climbing.

The third optimization algorithm tested was the ABAGAIL standard genetic algorithm. The algorithm maintains a population of neural network Iights, and randomly mutates and mates them each generation, keeping the fittest. The parameters for the algorithm are population size, offspring per generation, and mutations per generation. Two new functions have to be defined for our neural network Iights, a mating function and a mutation function. The mating function takes two instances of Iights and returns a new instance that is some combination of the two. The mutation function modifies the Iights of the given instance. For this exercise, the offspring of a pair is created from the average of their Iights, and the mutation function adds Iights from a random instance to the given instance.

The parameters chosen impact how the population develops. Each mating will result in a new instance betIen two other successful instances, and each mutation will result in a random step. After some experimentation, I elected to mutate half of the population each generation, and mate five times per generation. More mutation might have resulted in faster convergence, or possibly even divergence; more matings might result in a more homogenous population.

The runtime for the standard genetic algorithm is significantly slower than hill climbing or simulated annealing. This makes sense given that each generation must mutate and mate numerous instances, where as each iteration of hill climbing only had to mutate one. The advantage to this is the population can explore many paths at once, moving by random steps or by jumping to the midpoint of two other instances. The downside is that there is little pressure for individual members to improve, and only the periodic replacement through mating will prune the poor performers that have wandered too far in less fit space.

The stopping criteria for the genetic algorithm’s control loop is a fixed number of generations. I varied the number of generations from 100 to 300 in increments of 100. Training time scaled linearly with the number of generations trained. The fittest member of each population after stopping was measured on the training and validation sets. The performance of the genetic algorithm at 100, 200 and 300 generations is plotted next to randomized hill climbing and simulated annealing in figure 5.

As the generations increase from 100 to 300, the fitness and accuracy approach the peak found by simulated annealing, but does not quite reach the same performance. Additional iterations would likely allow the algorithm to improve, but the time cost discourages the selection of this genetic algorithm over simulated annealing to find absolute optima. Another solution is to initialize a population of instances with this genetic algorithm as exploration, then train each member with a hill climbing algorithm to find the nearest peak. Figure 5 also shows the performance of this hybrid approach.

Figure 5

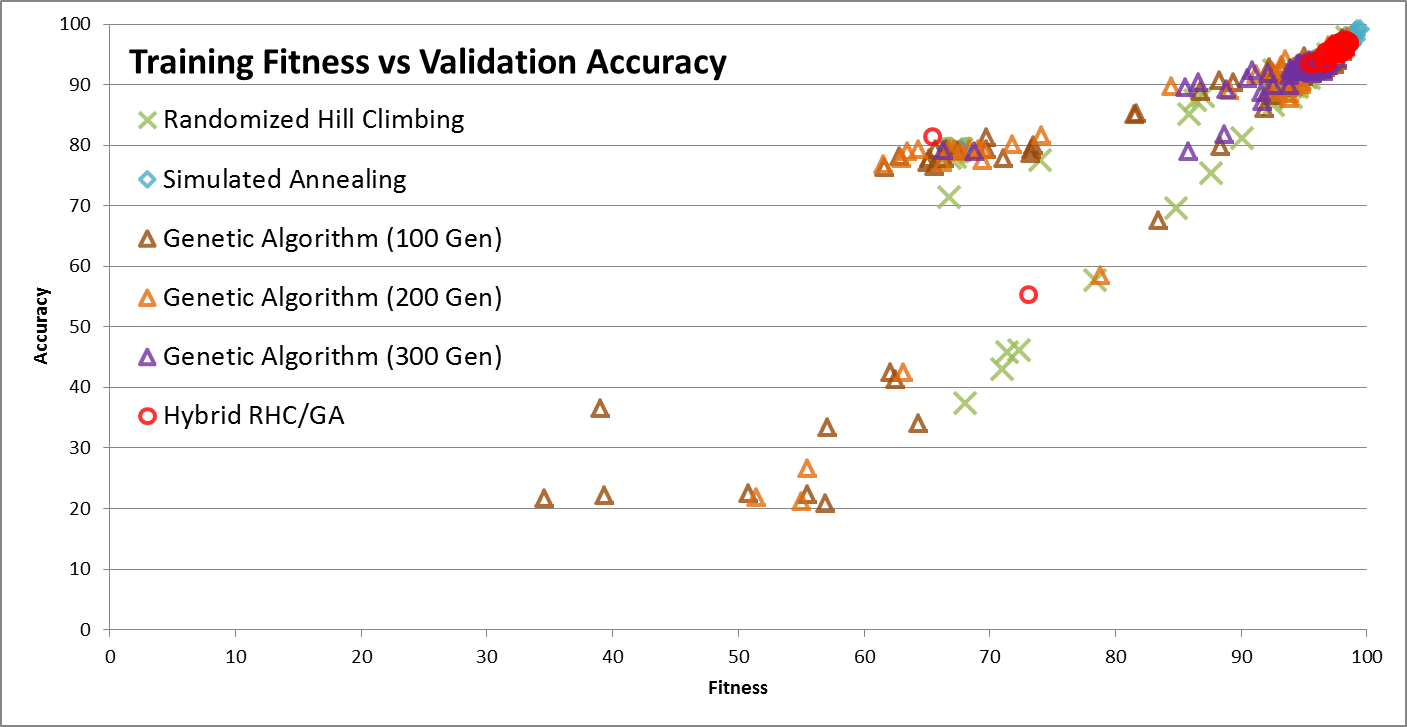


Table 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Optimization algorithm | Training time for 100 starts (sec) | Optimum fitness | Validation accuracy of optimum (%) | Standard deviation of fitness | Minimum fitness |
| Randomized hill climbing | 28.518 | 98.46 | 97.75 | 13.31 | 66.82 |
| Simulated Annealing | 57.432 | 99.53 | 99.04 | 6.61 | 67.75 |
| Genetic Algorithm | 782.300 | 98.27 | 96.30 | 4.52 | 66.32 |
| Hybrid RHC/GA | 54.959 | 98.63 | 96.97 | 4.09 | 65.44 |

The simulated annealing algorithm is superior to the other three algorithms in training neural network weights for our skin classifier in all measures of performance except time.

Contrast of optimization algorithms over selected problems

Flip Flop

The flip flop evaluation function counts the number of alternating bits pairs from right to left until the first non-alternating consecutive bit pair is encountered, then returns the count. Hill climbing and simulated annealing should have no trouble optimizing this problem, given enough iterations, since for any given instance, there is exactly one neighbor within a single bit flip that improves the function. A genetic algorithm with a single bit mutation function and a single point crossover function should also be able to find the absolute optimum eventually, since each mating has a good chance of preserving an alternating string of bits to the right and potentially extending that string to the left from an otherwise unfit instance. The MIMIC algorithm should be able to learn a dependency tree distribution that captures the alternating structure of bits from right to left. Any flat uniform distribution with no conditional dependence could not learn to optimize the flip flop function, because the alternating bit pattern of the optima averages to the uniform distribution.

Four Peaks

The four peaks evaluation function counts the consecutive leading ones in the head, the consecutive trailing zeros in the tail, and then awards a bonus equal to the size of the bit string if both the head and tail are longer than the parameter t. This is tricky for hill climbing algorithms, because for any given instance there are two neighbors that can increase fitness, but if the head and tail do not both grow to t before the algorithm stops, or if one of them grows beyond length minus t, then the largest fitness boost will never be found. Simulated annealing might allow more time to find the balanced head and tail in some iterations, but it will still tend to find lesser optima. Genetic algorithms have a better chance of finding the bonus through a single point crossover. MIMIC should excel at this, once one sample finds the bonus, it will be preserved in the kept pool to train the optimal distribution for all remaining iterations.

Continuous Peaks

The continuous peaks evaluation function is like the four peaks function in that it has a parameter and it counts runs of ones and zeros, but unlike the four peaks function, it counts the longest run of ones and longest run of zeros anywhere in the bit string. This provides more ways to improve fitness in any given iteration by flipping a bit on either side of the longest run of zeros and the longest run of ones, rather than on the inside of either the head or the tail as in four peaks. That difference makes it easier for hill climbing algorithms. It should not change the fitness of genetic algorithms or MIMIC to optimize this problem.

Count Ones

The count ones algorithm does what it says. It should be trivial for the hill climbing based algorithms to find a neighbor that improves any given instance in the early iterations, but once the zero positions become sparse, further improvements become harder to find. Genetic algorithms should perform well, as they replace the poor performers with crossovers of good performers, zeros will be selected out. The MIMIC algorithm should also quickly approach an optimum distribution of all ones, but the probabilistic nature might still return a suboptimal result from the optimum distribution.

**Table 4** Optimization Problem Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Problem** | **Algorithm** | **Total Time** | **Min** | **Max** | **Median** |
| Flip Flop | RHC | 00:01.4 | 61 | 69 | 65 |
| Flip Flop | SA | 00:01.9 | 78 | 79 | 79 |
| Flip Flop | GA | 00:01.5 | 65 | 74 | 71.5 |
| Flip Flop | MIMIC | 01:04.0 | 69 | 76 | 73.5 |
| Four Peaks | RHC | 00:01.2 | 80 | 80 | 80 |
| Four Peaks | SA | 00:01.5 | 80 | 151 | 115.5 |
| Four Peaks | GA | 00:01.4 | 91 | 109 | 97 |
| Four Peaks | MIMIC | 01:04.4 | 51 | 151 | 136.5 |
| Continuous Peaks | RHC | 00:01.2 | 72 | 106 | 83.5 |
| Continuous Peaks | SA | 00:01.6 | 85 | 112 | 105 |
| Continuous Peaks | GA | 00:01.6 | 79 | 93 | 86 |
| Continuous Peaks | MIMIC | 00:40.3 | 93 | 111 | 100.5 |
| Count Ones | RHC | 00:01.0 | 60 | 60 | 60 |
| Count Ones | SA | 00:01.4 | 60 | 60 | 60 |
| Count Ones | GA | 00:01.4 | 59 | 60 | 60 |
| Count Ones | MIMIC | 00:40.2 | 60 | 60 | 60 |