Jacob Meyers
Assignment 2 - Randomized Optimization
CS 4641
3 March 2019

Introduction

The purpose of this assignment is to highlight the differences, strengths, and weakness of four randomized optimization algorithms - randomized hill climbing, simulated annealing, a genetic algorithm, and MIMIC. In order to test these algorithms, I used four optimization problems - finding the weights of a neural network, count ones, traveling salesman, and four peaks. The problems and how they relate to the strengths of each algorithm is summarized below.

The Problems

The first problem selected was to try to find good weights for a neural network. I selected a neural network and trained it on the red wine dataset I used in Assignment 1, and instead of using backpropagation to learn the weights, I did trials with randomized hill climbing, simulated annealing, and a simple genetic algorithm with a 30% mutation probability. Given the random nature of each of these algorithms, especially when compared with backpropagation, I did not expect them to outperform the network using backpropagation.

The second problem was the count ones problem. This problem seeks to maximize the number of ones present in a bitstring. The fitness function used simply sums each of the values in the bitstring and returns this value. The problem is set up in such a way as to maximize this fitness function. I selected this problem to highlight the strength of simulated annealing, though randomized hill climbing should also perform fairly well on this problem. There is a single global maximum, which is the bitstring of all ones, and no real local optima. There is also a steadily increasing slope towards this global maximum. Because of these properties of the problem, randomized hill climbing and simulated annealing - which is more or less a variant of hill climbing - should perform fairly well.

The third problem selected was the four peaks problem. Like count ones, this problem operates on bitstrings. The difference is in the fitness function, which is defined as such:
Given a bitstring S of length N, we define $f(S, T)$ as $max[tail(0, S), head(1, S)] + R(S, T)$, where $tail(b, S)$ is the number of trailing b's in S, $head(b, S)$ is the number of leading b's in S, and $R(S, T) = N \; if \; tail(0, S) > T \; and \; head(1, S) > T, \; 0 \; otherwise$. There are two global maxima for this function, which are when there are $T + 1$ leading ones followed by all zeros and when there are $T + 1$ trailing zeroes preceded by all ones. There are also two suboptimal local maxima - strings of all ones and strings of all zeroes. This problem is designed to do two things. The first is to trap hill climbing and simulated annealing. The local maxima will be difficult to overcome for these algorithms. The second is to highlight the strengths of genetic algorithms.

Unlike the hill climbing algorithms, genetic algorithms generate an entire population of potential solutions at any given time step. This, combined with the notion of crossover and mutation will result in a larger portion of the search space being searched. Not only that, but since the crossover only happens for the fittest members of the population, genetic algorithms can potentially very quickly arrive at a decent solution by taking the best parts of the fittest members of the population.

The last problem I selected was the travelling salesman problem. In this problem, we're given a graph with some distance associated between two vertices. We would like to minimize the distance traveled by our salesman while still visiting all of the vertices and ending the trip at the starting vertex. Keeping with the theme of this assignment of maximizing a fitness function rather than minimizing a cost function, I chose to simply maximize -1 * distance traveled, which is equivalent to minimizing the distance itself. It should be noted that this problem is considered NP-hard. Given the complexity and difficulty of this problem, I considered it a good problem for MIMIC, since it is the most aggressive of the four algorithms in using information from previous iterations to find a solution.

Experiment Methodology

I'll now describe my methodology for my experiments of the three discrete optimization problems. For each experiment, I would vary the size of the input. I would compare each algorithm with a fixed set of max iterations on the entire range of input size I selected for a given problem. I would then repeat this procedure for different numbers of iterations for each of the algorithms, using 100, 1000, and 10000. This allowed me to compare the algorithms across both input size and number of iterations ran. I also kept track of the runtime, in seconds, of each of the algorithms for each of the input sizes, and plotted this as a function of the input size. For the count ones problem, I ranged input from 40 to 200 bits in steps of 20 bits. For the four peaks problem, I ranged input from 10 to 100 bits in steps of 100 bits. For the salesman problem, I varied the size of the graph from 5 vertices to 25 vertices, with increments of 1 vertex. For each of the algorithms, I stuck with a max_attempts parameter (maximum number of attempts to find a better state at each step) of 10. For the genetic algorithm, I used a population size of 200 and a mutation probability of 0.1.

For the problem of finding the weights of a neural network, I used a neural network trained on the red wine dataset from Assignment 1. I used a simple architecture of 1 hidden layer of 100 nodes. For each of randomized hill climbing, simulated annealing, and the genetic algorithm, I made a simple test/train split of the data, and kept track of both training and testing error for each iteration of each algorithm. 1000 iterations were used for each algorithm. Randomized hill climbing was allowed 100 random restarts, and the genetic algorithm used had a population size of 100 and a mutation probability of 0.7. I also used a classifier trained on the data with the same hidden layer architecture with backpropagation to use as a reference. At the end, I found at which iteration the best performing set of weights was found at, in terms of training accuracy.

Results

The first problem I tested was the count ones problem. There is one caveat with my number of iterations for this experiment - I held the number of iterations constant for MIMIC. I did this for a couple of reasons. The first was the runtime. As we will be able to see, the runtime for MIMIC as the inputs grow becomes very large, even for only 100 iterations. I felt comfortable doing this because MIMIC performed almost perfectly immediately, even with only 100 iterations. Not much more improvement could be gained by increasing the number of iterations for MIMIC. So, as I increased the number of iterations for the other three algorithms, I held MIMIC's number of iterations to 100. One other caveat for the plots below is that the legend is missing from the first two plots, but the associated curves are consistent between plots, so the legends on one of the others is valid to use when referring to the plot for n_iterations = 100.

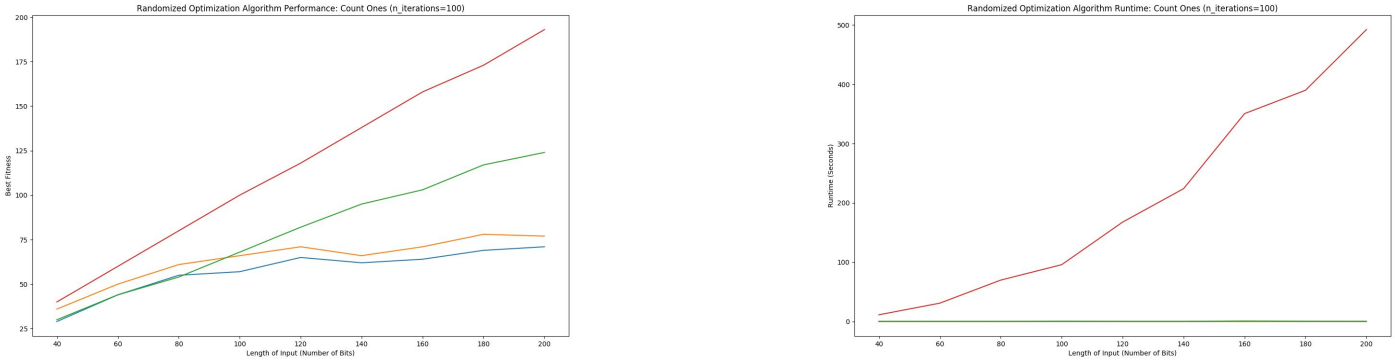


Figure 1. Plots showing both the performance and runtime of all four algorithms on the count ones problem with 100 iterations. MIMIC is red, genetic algorithm is green, simulated annealing is blue, randomized hill climbing is yellow.

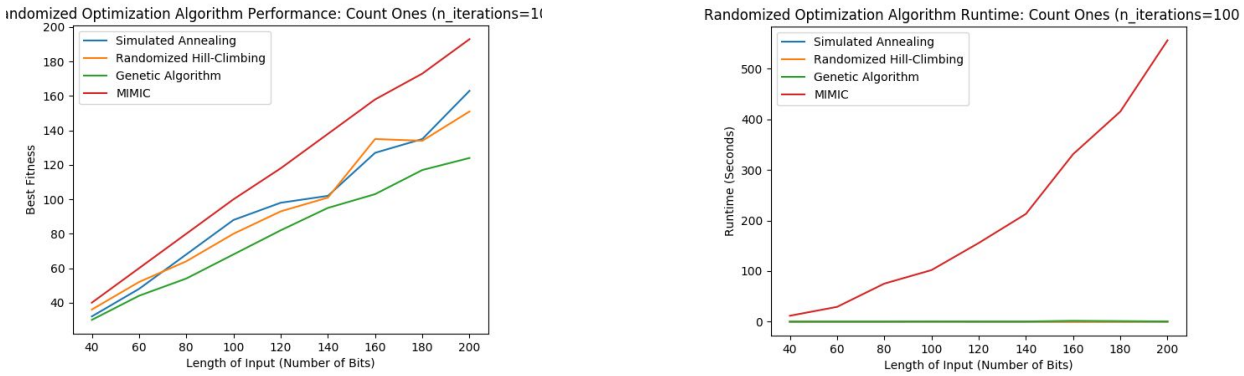


Figure 2. Plots showing both the performance and runtime of all four algorithms on the count ones problem with 1000 iterations (except MIMIC, which was held to 100 iterations).

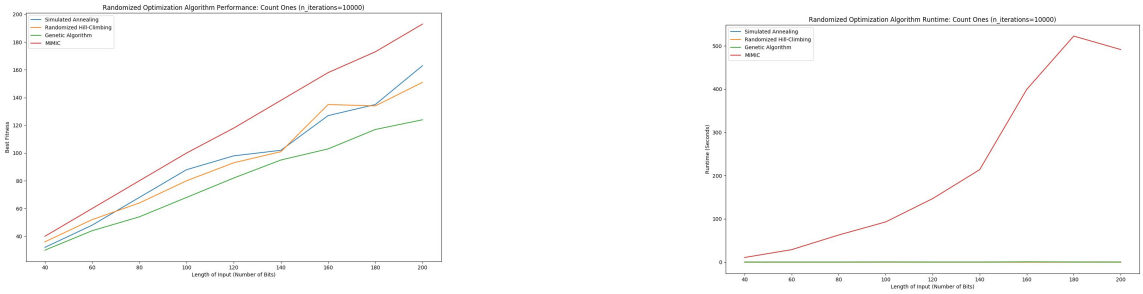


Figure 3. Plots showing both the performance and runtime of all four algorithms on the count ones problem with 10000 iterations (except MIMIC, which was held to 100 iterations).

We can see from Figures 1-3 that MIMIC performed much better than each of the algorithms, even though it was limited to 100 iterations. Even the genetic algorithm outperformed the hill climbing algorithms at 100 iterations. However, as the number of iterations increased, simulated annealing and randomized hill climbing overtook genetic algorithms. Though this problem was selected for simulated annealing, I expected randomized hill climbing to perform similarly, which ended up being the case. This is because of the relative simplicity of the search space, with no local minima. MIMIC outperformed both algorithms, but at the cost of a dramatically increased runtime. MIMIC performed on the magnitude of minutes, while hill climbing and simulated annealing never broke the one second mark. That would be the tradeoff in this case - choose MIMIC and get near-perfect performance at the cost of a severe runtime, or choose simulated annealing or randomized hill-climbing and get still good but not completely optimal performance with a much better runtime.

The next problem I tested was the four peaks problem. The results are listed below.

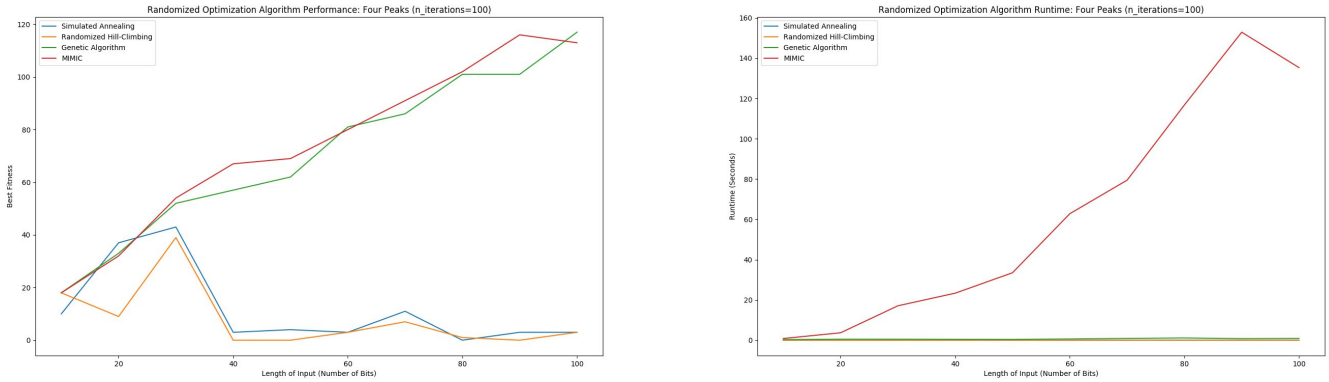


Figure 4. Plots showing both the performance and runtime of all four algorithms on the four peaks problem with 100 iterations.

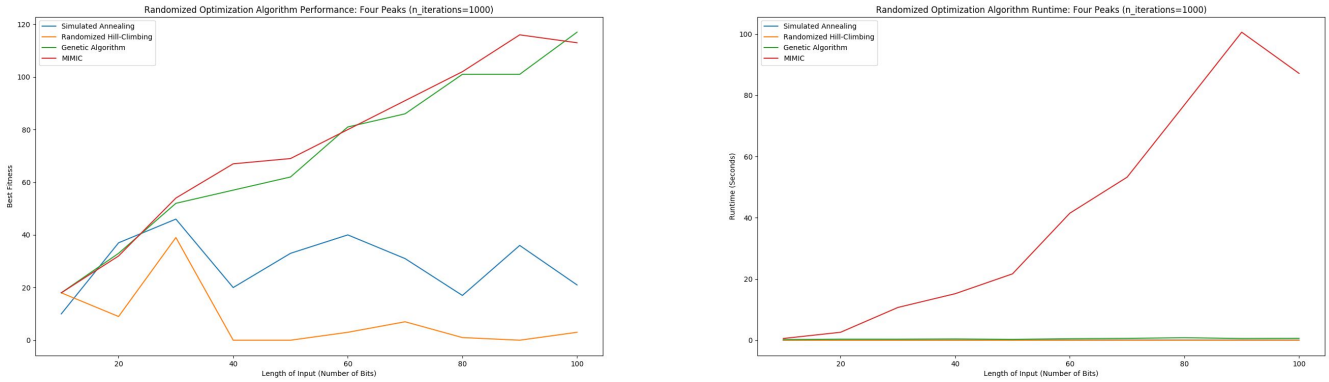
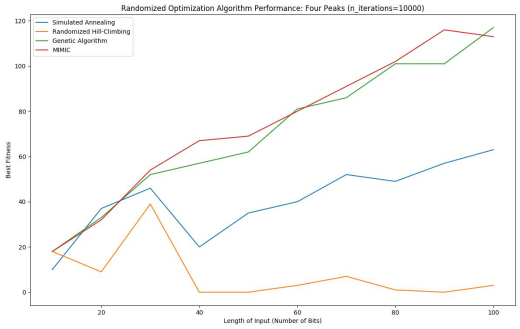


Figure 5. Plots showing both the performance and runtime of all four algorithms on the four peaks problem with 1000 iterations.



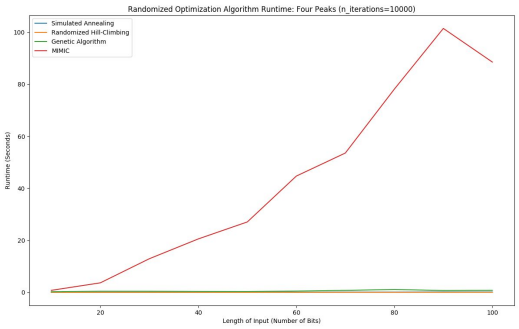


Figure 6. Plots showing both the performance and runtime of all four algorithms on the count ones problem with 10000 iterations.

The first thing that jumps out to me from these results is that the number of iterations seemed to have no effect on the performance of these algorithms. This is interesting to me, as this tells me that each of the algorithms arrived at their optimal solution within the first 100 iterations. I would not have predicted this before the experiments, especially with the knowledge that the hill climbing algorithms improved dramatically as iterations increased. Another interesting trend is the sharp decline in performance of simulated annealing and randomized hill climbing as the size of the input increases. At first I thought that this could be explained by the previously mentioned two local minima of the four peaks problem. However, after looking at the best state found by these algorithms at these larger inputs, none of them were the local minima of all ones or all zeros. Rather, they were seemingly random bitstrings. This tells me that for these larger inputs, hill climbing and simulated annealing were unable to even find the local optima, instead thrashing around the search space until the maximum number of iterations is reached. Perhaps the issue with regards to number of iterations is that even 10,000 may not be enough - perhaps 100,000 or even 1,000,000 iterations is needed. For smaller inputs, simulated annealing performed on par with the genetic algorithm and MIMIC before it fell off. As expected, the genetic algorithm performed very well with this problem. Though it performed below MIMIC for most of the inputs, we can see that it overtakes MIMIC and the upper end of the input size. The trajectories look as though genetic algorithms will perform better than MIMIC for inputs greater than size 100, but it's hard to say without definitive data on these input sizes, as the genetic algorithm curve is somewhat jagged and could conceivably decrease to below MIMIC's performance at some point past 100. As with the previous experiment, MIMIC had a significantly longer runtime than the other algorithms, but due to the smaller input sizes, performed in the neighborhood of 20 to 100 seconds rather than 100 to 500 seconds like with the count ones problem.

The last discrete optimization problem was the travelling salesman problem. The results are listed below. One note to make with the plots is the negative y-axis - this is because the travelling salesman problem is really a minimization problem, but I made it a maximization problem by maximizing the negative of the fitness score (which is the total distance traveled).

Another is the downward trend of the plots, compared to the earlier upward trends. This is again a consequence of the negative y-axis. The best performing algorithm will still have a curve that is above the others on the plot. With all this being said, let's look at the data.

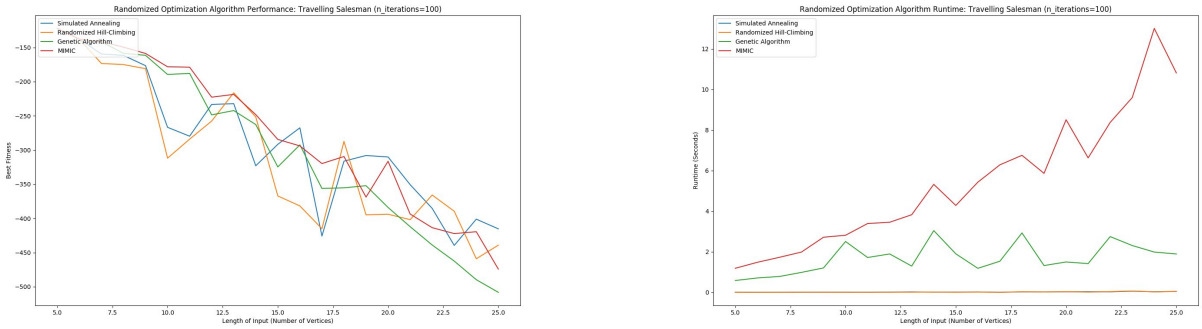


Figure 7. Plots showing both the performance and runtime of all four algorithms on the travelling salesman problem with 100 iterations.

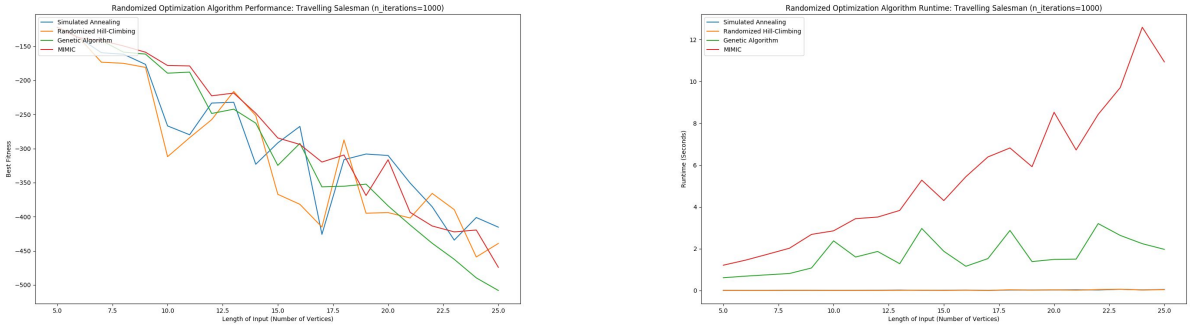


Figure 8. Plots showing both the performance and runtime of all four algorithms on the travelling salesman problem with 1000 iterations.

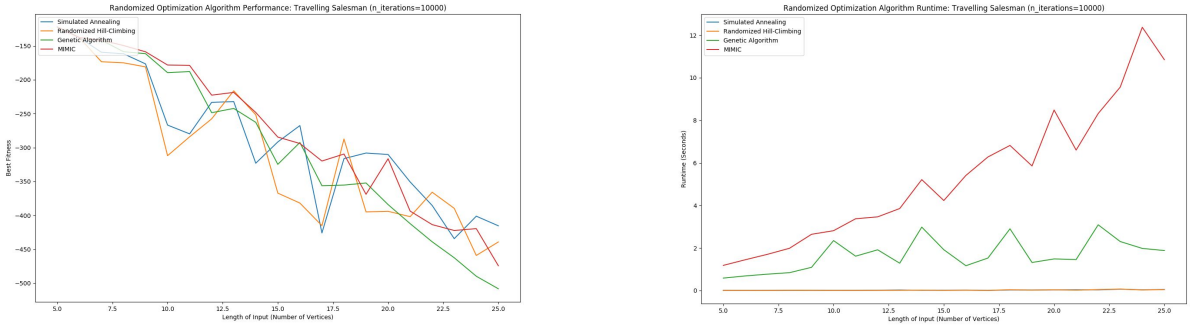


Figure 9. Plots showing both the performance and runtime of all four algorithms on the travelling salesman problem with 10000 iterations.

Once again, number of iterations had no real effect on the curves. I was surprised by this, as I thought that the complexity of this problem and the nature of these algorithms would lend themselves to better performance with more iterations. On the other hand, after seeing these results, I can imagine just the opposite being true - this problem's complexity and difficulty could be such that increasing the number of iterations for random search algorithms doesn't help much. The performance results of this experiment were the most surprising to me of the three. There really isn't much of a difference between any one of the algorithms. MIMIC doesn't even manage to hold on to the top spot for all sizes of inputs explored. This could partially be due to the relatively small size of the inputs - only graphs with up to 25 vertices were considered. However, in the previous experiments, if MIMIC was going to outperform the other algorithms, it usually did so from the very beginning, regardless of number of iterations or size of the inputs. This tells me I may have put too much stock into the prior knowledge MIMIC used in its iterations for the purposes of solving this problem. Generally, however, MIMIC performed well, especially on the smaller inputs.

The last experiment was that of finding weights for neural nets. In order to have a benchmark to compare these results to, I trained a backpropagation network on the same data with the same hidden layer size. It performed at around 58% to 61% test accuracy, with training scores around 85% and validation scores around 53%. The results of the 3 randomized optimization algorithms are listed below.

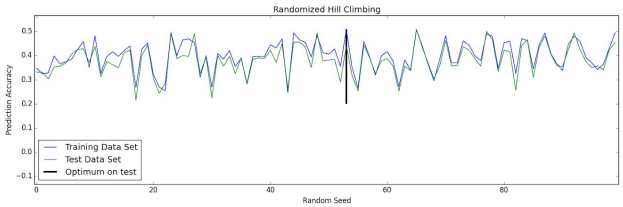


Figure 10. Plot showing the performance of a neural network on the wine dataset using randomized hill climbing to choose the weights. The optimal test performance is denoted by the

black bar. The test and train scores are plotted against the random restarts - for each random restarts, 1000 iterations of hill climbing were performed.

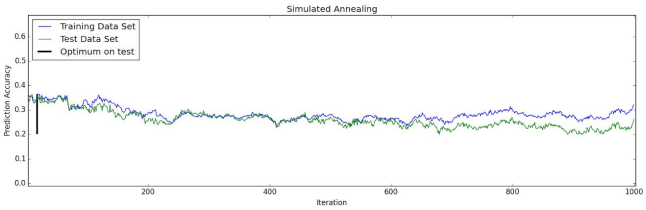


Figure 11. Plot showing the performance of a neural network using simulated annealing to choose weights on the wine dataset as a function of iteration number. The black bar denotes the iteration in which the optimal test performance was achieved.

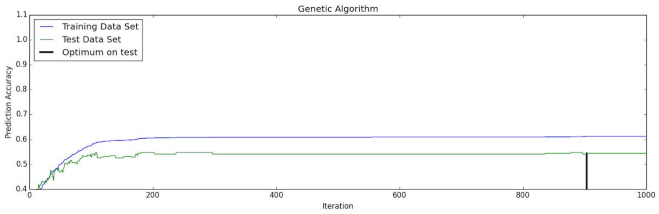


Figure 12. Plot showing the performance of a neural network using a genetic algorithm to choose weights on the wine dataset as a function of iteration number. The black bar denotes the iteration in which the optimal test performance was achieved.

The first thing to notice is that the genetic algorithm vastly outperforms the other two algorithms. Not only that, but the optimal test accuracy achieved by genetic algorithms, 0.5984, is within range of the test scores achieved by the benchmark backpropagation network. It is also over 15% better than the best accuracy achieved by randomized hill climbing, and almost 25% better than simulated annealing. I'm not particularly surprised by the fact the genetic algorithm outperformed the others, but I am surprised by the magnitude of the gap between the genetic

algorithm and the others. I think the genetic algorithm's use of an entire population and notion of crossover is useful for this problem. It allows for some information from the past to be carried over to the current iteration, much like backpropagation. The hill climbing algorithms fared much worse, with randomized hill climbing reaching an optimal test accuracy of 0.4406 and simulated annealing achieving an optimal test accuracy of an abysmal 0.3593. I imagine this could be because searching for neural network weights is a very large search space riddled with local optima for simulated annealing and randomized hill climbing to get stuck in. I attribute randomized hill climbing's success over simulated annealing to its random restarts, which give it a better chance to explore more of the search space and get "unstuck" from any local optima it may have found itself in. Simulated annealing does not have as drastic a measure, and thus performed the worst of all of the algorithms.