Rahul Zhade
CS 4641
Project 2 Write Up

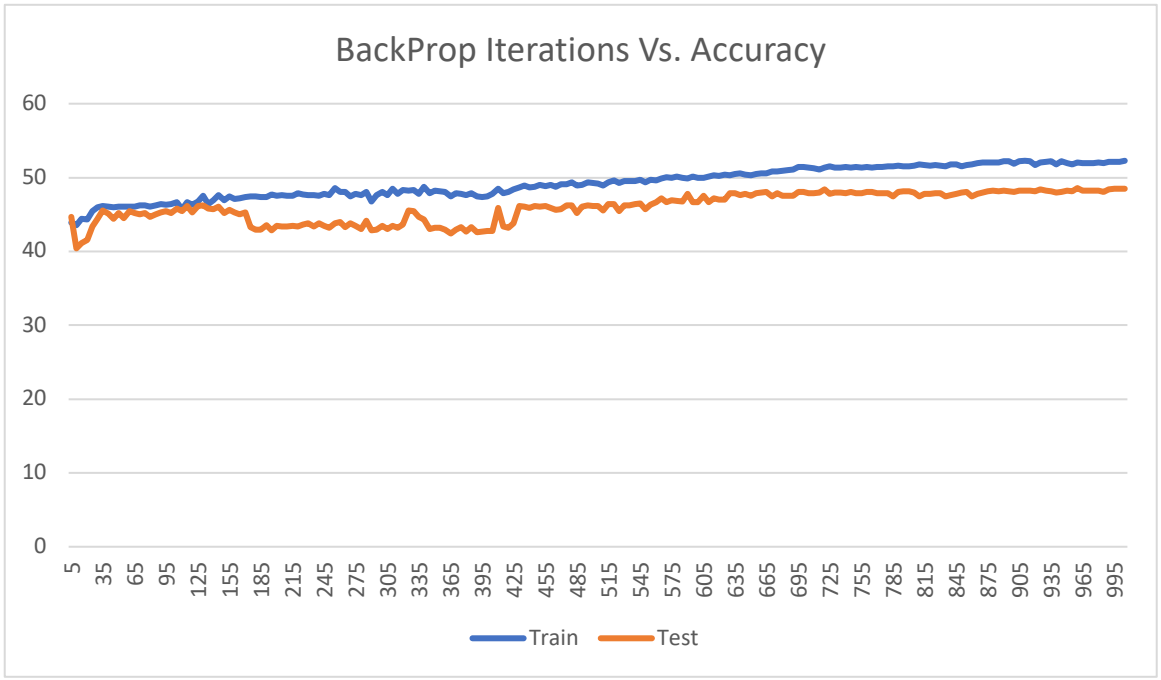
# Randomized Optimization Report

## Dataset

The Wine Quality dataset from the UCI Machine Learning Repository is a collection of different physicochemical attributes about wine, and the wine's corresponding quality, a presumably (relatively) ordinal attribute. I used the white wine subset of the data, as it has more instances than the red wine subset, at 4898 instances with 11 attributes each. The goal with this dataset is to predict the wine quality as assigned by wine experts, using the physical attributes of the wine.

## Neural Net Optimization

For phase 1 of this assignment, three randomized optimization algorithms were used to optimize the weights of a Neural Network, using the accuracy of classifying instances of the above dataset as the fitness function. These included Randomized Hill Climbing, Simulated Annealing, and Genetic Algorithm optimizers. Fitness in this case was defined to be the train accuracy, i.e. the number of train cases that were properly classified by the neural network.
As a baseline test, I first ran the neural network with just backpropagation. These results are shown below, after 1000 iterations of training:



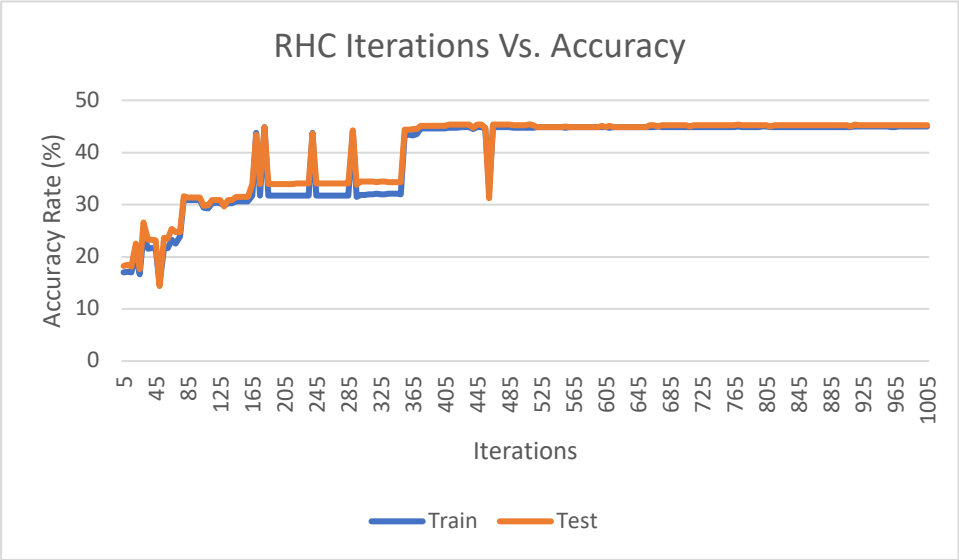

Training Time: 10.653 s

This error rate, as discussed in the previous assignment, is reasonable, considering that there are 9 possible classes. The time is also an important benchmark, as this neural network took a reasonably short amount of time to train. Furthermore, as can be shown above, the backpropagation optimizer had not converged as of 1000 iterations, so theoretically, if run for longer, this could have

had a better performance. As a note, the testing time was negligible compared to the training time, and as such, I simply added this time to the time it took to train. This observation was true of all the optimizers shown below, and as such, as a matter of simplicity, I did not include testing time in the tables.

## *Randomized Hill Climbing (RHC)*

Randomized Hill Climbing takes no parameters, so running this algorithm was fairly straightforward. Graphs are below:




| At 1000 Iterations | | |
|---|---|---|
| | Train | Train |
| Accuracy % | 44.946 | 44.796 |
| Time (s) | 29.56 | - |

The performance of the Randomized Hill Climbing optimizer as shown above shows the convergence of this optimizer towards a solution with accuracy around 45% for the train data. Even across subsequent runs, this value stayed relatively constant. While the initial accuracy varied between 0% to one instance of 40%, every time I ran the code, it always converged at an accuracy somewhere near 45%. This was expected, as the nature of RHC is to have random starting positions.

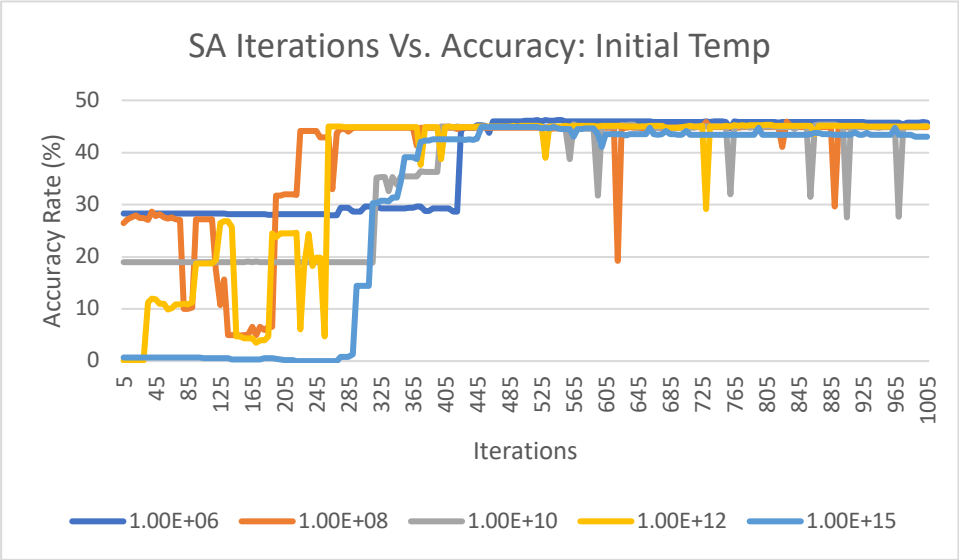
Using the backpropagation as a baseline, the randomized hill climbing optimizer performed fairly decently. Using just local search, this optimizer was able to boost the neural network's accuracy up to ~45%, almost as good as the ~48% achieved by backpropagation. RHC also converged at a suboptimal solution at around 500 iterations, while backpropagation never really converged over the interval that I trained it for. The way in which the RHC converged was also fairly interesting, as it moved up in steps, rather than increasing linearly. As a greedy local optimizer, some of the steps that this optimizer chose drove the accuracy straight up, due to minor optimizations for the weights leading to large increases in neural network performance. Since this neural network had 30 hidden layer nodes, minor adjustments to the weights could have drastic changes to performance. This was also likely the reason behind the sudden drop in performance at 450 iterations. Since the greedy optimizer optimized based on sums of the squares of errors, noise in the distribution of error could have caused the optimize to make the wrong move, which is corrected later.

The run time for this optimizer was fairly long, as compared to the backpropagation optimizer. This was likely due to the fact that this optimizer needed to compute the squares of errors at each

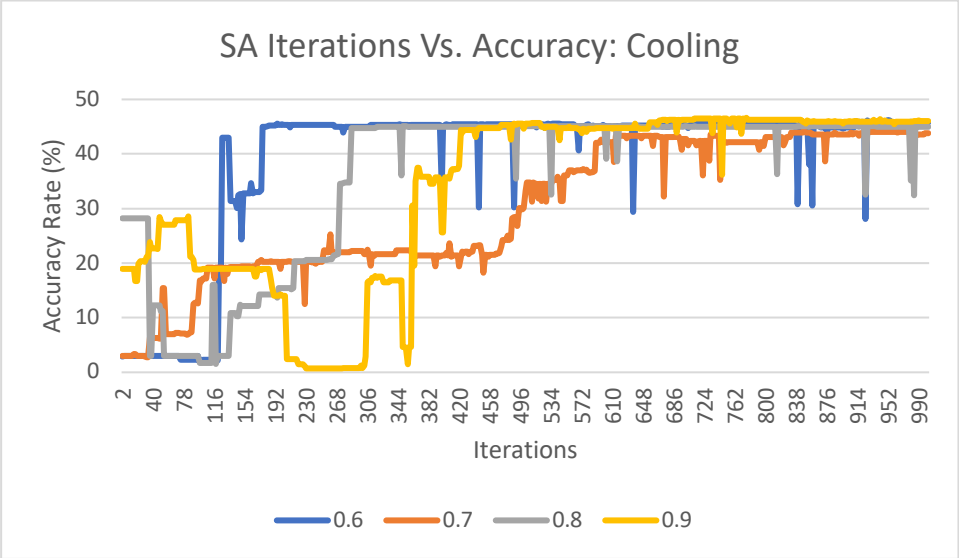
stage, and calculate local optimizations for a dataset of size ~4000, and a neural network with 30 hidden nodes.

## *Simulated Annealing (SA)*

Simulated Annealing takes two parameters: initial temperature, and cooling rate. Below, I have shown the effect of changing initial temperature, fixing a cooling rate of 0.8, or changing the cooling rate, fixing an initial temperature of 1E8. These values were determined by choosing the parameters that had the minimum average error after training for 1000 iterations.




| At 1000 Iterations | | |
|---|---|---|
| | Accuracy (%) | Time (s) |
| 1e6 | 44.13 | 29.53 |
| 1e8 | 44.898 | 30.99 |
| 1e10 | 45.021 | 31.26 |
| 1e12 | 45 | 31.13 |
| 1e15 | 43.061 | 30.62 |




| At 1000 Iterations | | |
|---|---|---|
| | Accuracy (%) | Time (s) |
| 0.6 | 45.816 | 32.88 |
| 0.7 | 43.776 | 28.40 |
| 0.8 | 45.000 | 30.89 |
| 0.9 | 45 | 31.13 |

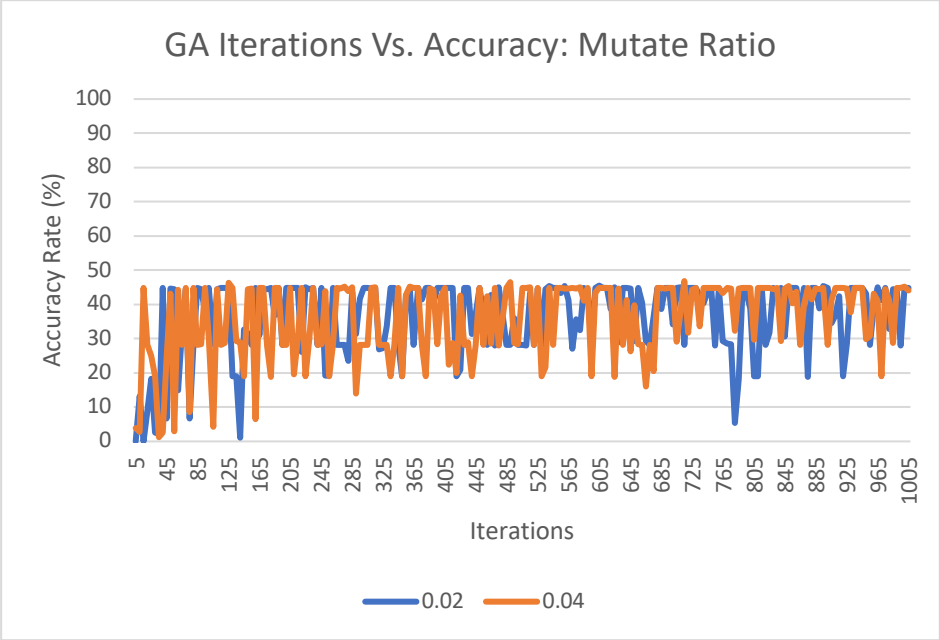
All of the simulated annealing solutions converged at nearly the same accuracy, at 45%. However, there was a major difference in the number of iterations, and by extension, the time, that it took for the optimizer to get there. From the graphs above, it is clear that, despite not starting at the greatest starting point, a cooling rate of 0.6, and an initial temperature of either 1E6 or 1E12 converged on the solution the quickest. The temperatures are not surprising, as too high an initial temperature will cause

the optimizer to keep jumping around for the first few iterations, not converging until the temperature cools down a bit. However, too low a temperature, and the SA optimizer would get stuck in a local maxima. Likewise, a higher value for the cooling rate (leading to a slower decrease in temperature) seems to increase the probability of the optimizer making a bad decision (as can be seen for C=0.8, I=40 and C=0.9, I=192 and 344). Again, this is because the optimizer is still bouncing around, leading to a lower fitness for that iteration.

The run times for all of the algorithms were within the same general region, leading me to conclude that the values for these parameters do not affect the run time of the code. This is fairly intuitive, as jumping around does not affect the length of each iteration for simulated annealing, it simply increases the amount of time required to converge to a solution. The time it took any of these optimizers to run 20000 iterations was more than 3 times longer than that of backpropagation though, even with a lower testing error.

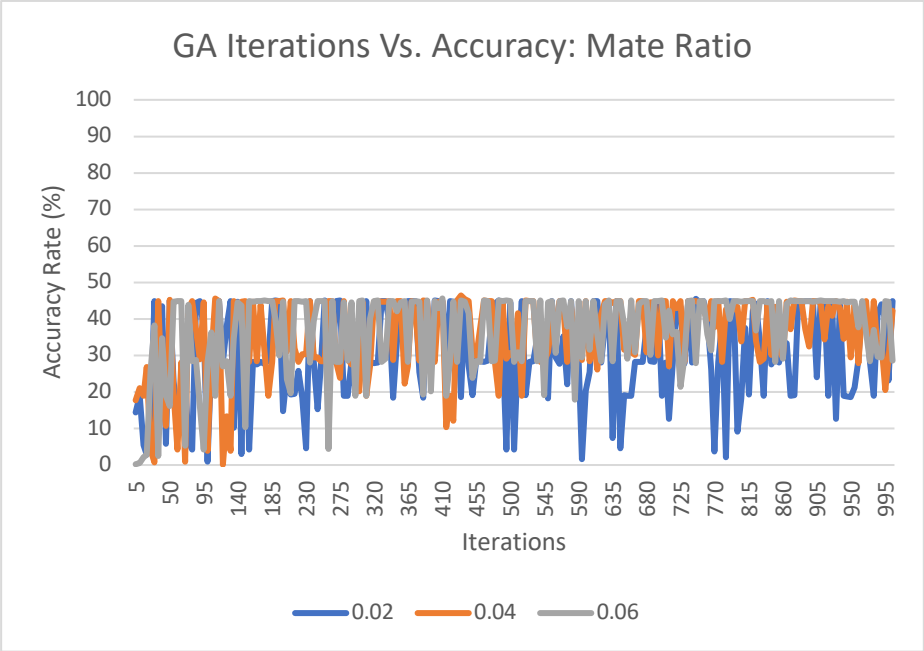
## *Genetic Algorithm (GA)*

Genetic Algorithms take three parameters: a population ratio, a mutate ratio, and a mate ratio. Similarly to the previous problem, I fixed two parameters using the method described above, and then observed the behavior of the optimizer. The fixed parameters were 0.15 for population ratio, 0.04 for the mating ratio, and 0.02 for the mutation ratio.



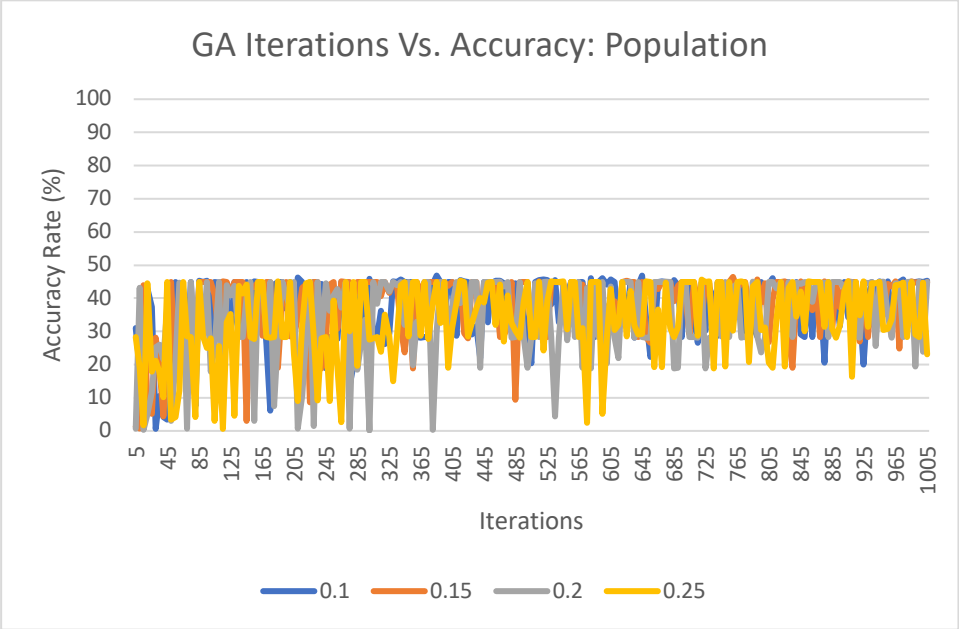

| At 1005 Iterations | | |
|---|---|---|
| | Accuracy (%) | Time (s) |
| 0.02 | 44.08 | 4352 |
| 0.04 | 44.898 | 2146 |

| | Max Accuracy |
|---|---|
| 0.02 | 45.51% |
| 0.04 | 46.837% |




| At 1005 Iterations | | |
|---|---|---|
| | Accuracy (%) | Time (s) |
| 0.02 | 44.898 | 1143 |
| 0.04 | 42.245 | 3311 |
| 0.06 | 28.571 | 6319 |

| | Max Accuracy |
|---|---|
| 0.02 | 45.51% |
| 0.04 | 46.429% |
| 0.06 | 45.612% |




| At 1005 Iterations | | |
|---|---|---|
| | Accuracy (%) | Time (s) |
| 0.1 | 45.31 | 4352 |
| 0.15 | 44.79 | 4420 |
| 0.2 | 44.90 | 6415 |
| 0.25 | 23.061 | 8452 |

| | Max Accuracy |
|---|---|
| 0.1 | 46.837% |
| 0.15 | 46.429% |
| 0.2 | 45% |
| 0.25 | 45.612% |

The first thing that can be noticed from these graphs is the inconsistency of the accuracies. While all of the optimizers, were able to find a solution with >45% accuracy, which is better than Simulated Annealing and RHC, there was so much noise in these solutions, that the final output spat out by the optimizer was almost never the optimal solution. For the mate ratio of 0.06, the final accuracy was an abysmal 28.57%. despite it being able to find a solution of maximum accuracy 45.61%. This noise is likely due to the fact that sets of weights that might be optimal on their own, might not make for ideal weights when combined. Especially considering the fact that there are several hidden nodes in the neural network, small changes, especially mutations, can lead to large deviations in the performance of the network.

In terms of the parameters, most of the accuracies seemed fairly similar, all lying between 45 – 46%, so I can't extrapolate about increases in accuracy due to these parameter changes. Increasing the population ratio seemed to increase the deviations from the optimal solution, as P=0.25 and P=0.2 had the largest deviations from their max accuracy, as can be seen from the graph above. This was fairly surprising, as I would have expected the genetic algorithm to perform more consistently with a larger population size, so it would be less likely to overfit. However, this might have been the very reason why the GA performed less well, as the larger population size might have meant that there was a greater variety of solutions, not all of which were compatible with one another. If certain individuals were overfitting, they would have had a misleadingly high fitness, propagating them to the next stage. If they are crossed over at this stage, due to the fact that both are overfitted, the fitnesses of these individuals might not carry on to the next stage.

Similarly counterintuitive, decreasing the mate ratio of the genetic algorithm made the algorithm more noisy, and deviate more from the optimal path. I am not quite sure why this could have happened, however, I believe that this might have been because a population without any crossover is more likely to contain suboptimal individuals. In a population with a lot of crossover, by virtue of numbers, there are going to be more fitter individuals, whereas in a population without crossover, there is less choice among the progeny.

In terms of run time, increasing the mutate ratio tended to decrease the run time, while increasing the mating ratio and population ratio increased the run time. This makes sense, as mating individuals take computational effort, in order to find the individuals to mate, as well as to cross them over, while the mutate function can be randomly applied. The population ratio increases the population size, which increases the run time because there are more individuals that the algorithm must worry about.

### *Conclusion*

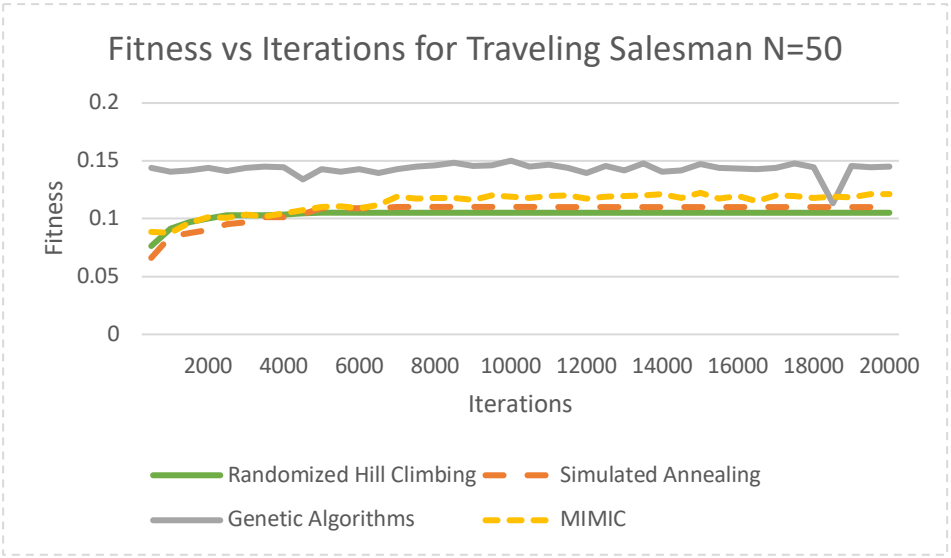
From the first part, we can gather that, while none of these optimizers was able to match the performance of backpropagation, either in run time or accuracy achieved. A neural network using backpropagation was able to consistently achieve test accuracy rates of 48%, while Genetic Algorithms stumbled their way towards 47% with the right parameters, Simulated Annealing converging at 45-46% and Randomized Hill Climbing always reaching a maximal accuracy of 44.95%. Backpropagation was also able to perform faster than all of these optimizers, even the local search optimizers SA and RHC. This might be because backpropagation corrects the weights at each instance of data passes through the neural network, rather than all at once at the end of each iteration, like the other optimizers. Just among these three optimizers, simulated annealing with the right parameters worked the best, as, although RHC had similar performance in accuracy and run time, it was less consistent in the number of iterations it took to converge. GA had a higher test accuracy, however, it was so inconsistent in converging that it would not be practical at all to use as a optimizer for the weights of a neural network.

## Optimization Problems

### *Traveling Salesman Problem: Genetic Algorithms*

This is a graph problem requires finding the shortest path between a set of vertices in a graph. This problem takes one parameter, **N**, which is the number of vertices in the problem. For simplicity, I

plotted the below graphs for an N of 50, however, I did test out other values and will discuss them below.




| Run Time for 20000 Iterations (secs) | |
|---|---|
| RHC | 0.041 s |
| SA | 0.077 s |
| GA | 32.204 s |
| MIMIC | 50985.88 s |

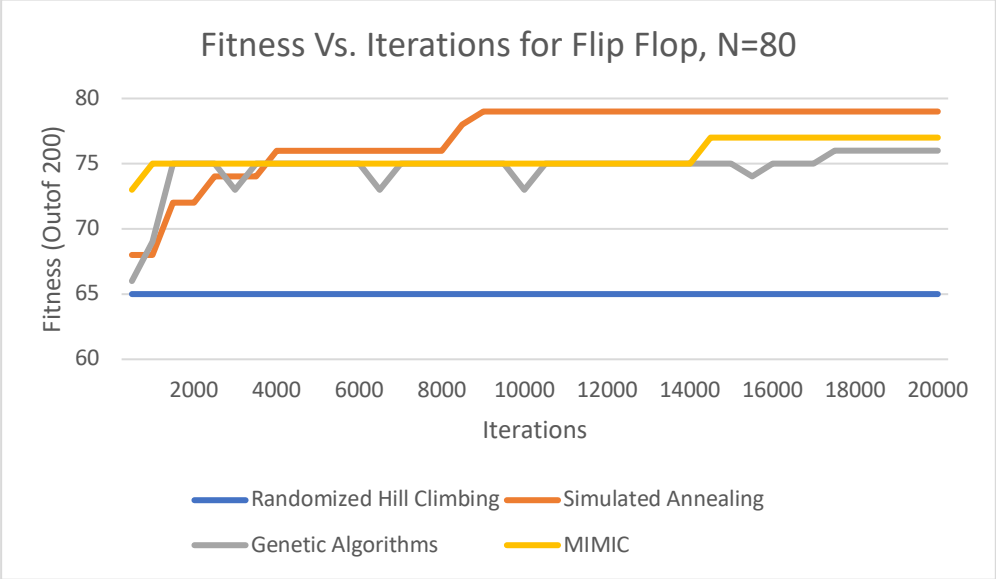
This was a pretty interesting problem, as it demonstrated the weaknesses of the MIMIC optimizer perfectly, along with the strengths of the genetic algorithm optimizer. From the graph, it is clear that the genetic algorithm peaked at the highest fitness, as well as arriving at a better solution faster than any of the other optimizers. This is because this problem favors similar solutions, so the genetic algorithm which crosses similarly fit solutions was able to generate more fit solutions quicker. However, this still had the possibility of generating unfit children, as can be seen in the one dip at ~19000 iterations. However, as a whole, genetic algorithms thoroughly outperformed all of the other optimizers for this problem. This problem has an extremely large hypothesis space, due to the fact that there are an infinite number of paths joining a set of points on a graph. This was likely the reason behind the extremely long run time for MIMIC, at nearly 15 hours. Due to the large hypothesis space, as well as the improbability of converging at a good solution quickly, MIMIC did not converge anytime during the 20000 iterations that I ran it for. It was clear that, even at the end, it was still getting better results after more iterations, however, I had already ran it for 15 hours, so I stopped before it converged. RHC converged at a suboptimal solution at around 4000 iterations, which was expected. Simulated Annealing took longer to converge, which was also an expected result, however, it was able to converge to a greater fitness than the Simulated Annealing optimizer. When run for higher values of N, the results were somewhat similar, however, MIMIC took much longer to run, which was expected due to the probabilistic sampling nature of MIMIC. RHC and SA took a bit longer to run as well, and performed worse, likely because the larger space of hypotheses leads to more local maxima to get stuck in.

There are likely a few optimizations that could have been done to make the above code run faster and more efficiently. Firstly, decreasing the number of samples that MIMIC took would have likely sped up the code significantly, as 200 samples taken over 20000 iterations leads to 4000000 samples that need to be analyzed. However, a smaller sample size means that it is less likely that the optimizer would converge upon the global maxima. The genetic algorithm optimizer performed quite well in terms of converging at a good fitness, however, the runtime was a bit long. This is natural of GA, but could have

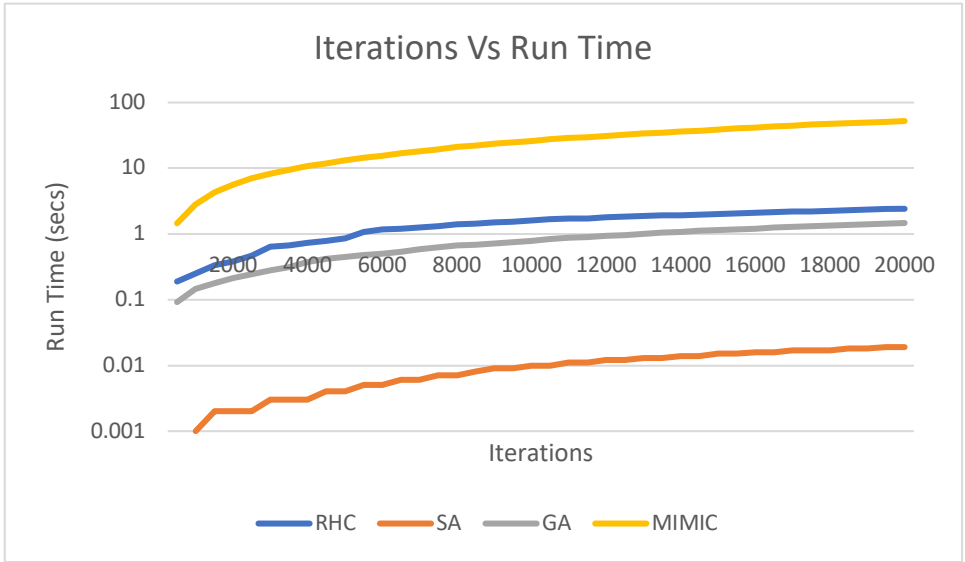
probably been reduced by increasing the mate ratio. As explained in the first phase, a higher mate ratio for the proper problem leads to faster convergence, at the expense of some bias.

## *Flip Flop: Simulated Annealing*

The Flip Flop problem is a bit string problem in which fitness is determined by the number of times consecutive bits in a string alternate. This is quite similar to the Four Peaks problem, however, there are even more suboptimal local maxima that make this problem much more useful to distinguish between the merits of the two local optimizers. This problem takes one input, **N**, which describes the length of the bit string. I ran simulated annealing at a cooling of both 0.95 and 0.8, however, the performance of the 0.8 was much better so I will discuss those results below.




| Run Time for 20000 Iterations | |
|---|---|
| RHC | 2.255 s |
| SA | 0.016 s |
| GA | 1.691 s |
| MIMIC | 52.342 s |



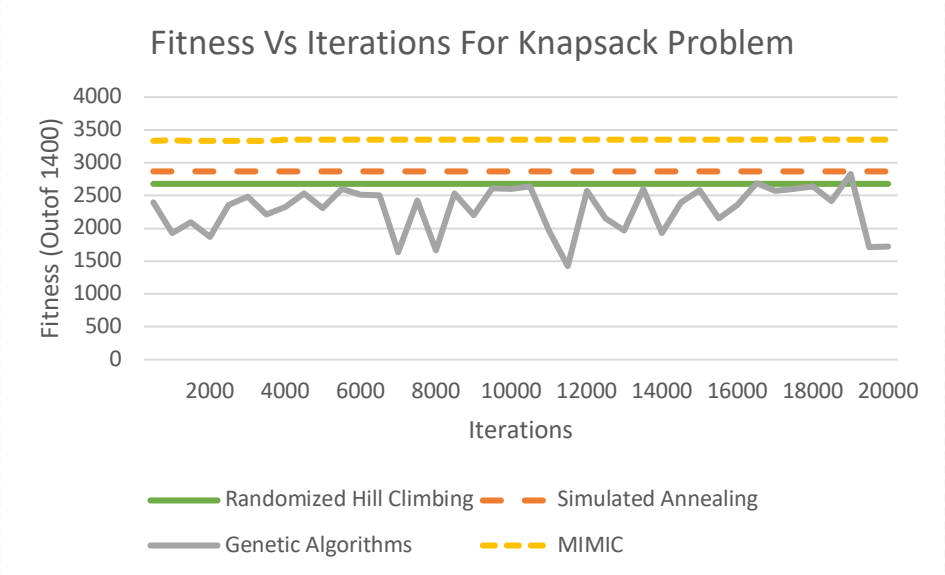

While simulated annealing initially got worse results than both MIMIC and genetic algorithms, after around 8000 iterations, it was able to converge upon a more optimal solution than either one of those

optimizers. It was unsurprising that RHC got stuck at a suboptimal solution, and was unable to break out of it even after 20000 iterations, as the nature of the problem is such that most local answers lead to their own local maxima rather than to the global maxima. Simulated Annealing is able to get unstuck from these local maxima because of the annealing portion of the algorithm, while also getting a quickness boost because it is still a local optimizer.
It was interesting however, that Simulated Annealing only has an advantage for smaller values of N, and, as N grows larger, it begins to perform more poorly. This is likely due to the fact that larger values of N mean a larger hypothesis space, which means more local optima that the local optimizers need to avoid.

As we saw above, in Phase 1 of this project, a simulated annealing optimizer, when applied to the correct problem, can converge faster if a smaller cooling factor is used (cooling down the optimizer quicker). While I ran this experiment with a value of C=0.8, running it with a value of C=0.6 might have made the convergence happen in less iterations. However, this also runs the risk of cooling down too fast, leading to the optimizer getting stuck in a suboptimal solution.

## *Knapsack Problem: MIMIC*

The Knapsack problem is a common optimization problem which requires placing as many weighted object into a knapsack as possible, without exceeding the maximum weight capacity. This given knapsack problem also has a volume constraint, so the ideal solution would stuff as many items into the bag while still meeting the weight and volume constraints.




| Run Time for 20000 Iterations | |
|---|---|
| RHC | 0.157 s |
| SA | 0.595 s |
| GA | 14.335 |
| MIMIC | 762.455 |

An important thing to note from this graph is that most of the optimizers converged fairly quickly, within the first 3000 iterations for all of RHC, SA and MIMIC. Again, run time was as expected, with RHC taking the shortest amount of time, followed by SA, with MIMIC coming dead last by a large margin. However, it was the maximum fitness achieved that shows the advantages of using MIMIC. The Knapsack problem is full of local maxima, and the global maxima is unlikely to be formed from the union of two smaller optimal problems. Genetic algorithms tend to perform poorly on problems in

which there are multiple types of solutions, as the crossover function and the mutations serve to "average" the individuals towards the one true solution. However, problems such as these have a more abstract solution, so genetic algorithm optimizers tend to just stumble between fairly high fitness solutions and abysmally bad solutions, as can be seen in the graph. RHC and SA both got stuck at local optima fairly early on, even before the first measurement at 500 iterations, and were not able to break out of them even when given 20000 iterations. This attests to the nature of the hypothesis space. MIMIC was able to converge at the global maximum due to the way that it samples the most likely region to contain the global maximum. However, due to this, it also led to MIMIC's quite long run time. I tried experimenting with different parameters to decrease the run time of MIMIC, notably by decreasing the number of samples taken and the number of samples kept, however, this did not boost the runtime of MIMIC significantly enough to justify the fitness dip that occurred because of doing so.