Randomized Optimization

William Tidwell
Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
trace.tidwell@gatech.edu

***Abstract*—The purpose of this assignment is to explore randomized searched techniques used in optimization. Four search algorithms – randomized hill climbing, simulated annealing, genetic algorithm, and MIMIC – are implemented and analyzed. First, the weights of a neural network are tuned using the first three algorithms. The, all four algorithms are compared on three different problems in an attempt to highlight the strengths and weaknesses of each. A description of the problem and an analysis of the results are included for each part.**

Keywords—hill climbing, simulated annealing, genetic algorithm, MIMIC

I. Introduction

I first learned the phrase, "There is no such thing as a free lunch," in tenth grade economics. Simply put, it means one cannot get something for nothing. As I began studying machine learning, I learned that the phrase was also used in reference to algorithms, with much the same meaning. It means that no one algorithm is inherently better than another for solving a given problem. Each problem must be considered on an individual basis, and prior knowledge must be applied in order to achieve optimal results. In the case of machine learning, one cannot just select "the best" algorithm to solve a problem and expect to have long-term success. An important part of knowing which algorithm to select is domain knowledge, or experience in the field in which one is working. After having spent considerable time in a certain field or solving certain problems, one begins to have an understanding of which algorithms perform well and which do not, as well as how best to implement the algorithms. In this assignment, we will begin exploring different algorithms and techniques used in supervised learning with the goal of starting to develop some domain knowledge or, at the very least, being exposed to some of the problems faced in machine learning.

II. Algorithms

A. Randomized Hill Climbing

Randomized hill climbing (RHC) is a very simple algorithm. A solution to the problem is "guessed" or initialized randomly. The neighbors of the solution are evaluated. If a neighbor provides a better solution. If all neighbors are evaluated and no improvement is found, the algorithm terminates. The major drawback of RHC is that is incredibly susceptible to local optima. Since it only ever accepts better solutions, it will only ever find the closest local optimum to its starting point. A common method for dealing with this problem is random restarts. Once the algorithm reaches a local optimum, the solution is saved, and the search is started again. The process can be repeated many times. This still does not guarantee a global optimum will be found, but it allows the algorithm to explore more of the solution space.

B. Simulated Annealing

Simulated annealing (SA) is modeled after the physical process of annealing. The idea is that, at the beginning of the search, the temperature is high, which encourages exploration of the solution space. As the search progresses, the temperature is gradually decreased, and the algorithm becomes less likely to explore and more likely to accept only those solutions that are improvements over the current one. SA is still susceptible to local optima, but because of the higher likelihood of accepting worse solutions early in the search, it does a better job of exploring the solution space than RHC. In fact, one of the properties of SA is a probability distribution that the algorithm will terminate at a given solution. This is known as the Boltzmann distribution, and is defined in Equation 1 as

$$\Pr(x) = \frac{e^{f(x)/T}}{Z_T} \tag{1}$$

where $f(x)$ is the fitness of x and T is the temperature. Z_T is a normalizing factor to ensure the probabilities sum to one.

C. Genetic Algorithms

Genetic Algorithms (GA) are modeled after the biological process of natural selection. An initial population is given or generated randomly, and the fitness of each individual is calculated. Next, crossover is performed, which is similar to the process of mating. Two parents are selected from the population, usually in relation to their fitnesses. Two children are created by combining parts of each parent. After a new population of children has been created, a mutation is applied to some percentage of the children, which changes part of their solution. The children population is added to the original population, and the best candidates are carried into the next generation. Like the other two algorithms, genetic algorithms are also prone to finding local optima. However, starting with multiple solutions in the form of the population combined with crossover and mutations help the algorithm explore the solution space. A problem specific to genetic algorithms is when a single individual dominates the rest of the others in terms of fitness. This greatly increases its likelihood of being selected for crossover and can often end up with the entire population being made up of a single solution.

D. *Mutual-Information-Maximizing Input Clustering*

Mutual-Information-Maximizing Input Clustering (MIMIC) is a randomized search algorithm that seeks to remedy some of the shortcomings of the previous three algorithms. In particular, it seeks to capture any underlying structure to the problem that might be beneficial in finding a solution. The first three methods essentially provide starting and end points with little explanation for how we moved from one to the other. MIMIC assumes the solutions belong to a probability distribution that describes the fitness of all elements included in it. By taking the elements with the best fitnesses and determining the pairs of features with the highest mutual information, we can estimate a new probability distribution in which the expected fitnesses are higher than the previous. This process is repeated until the distribution has been found that produces the optima.

III. Neural Network Optimization

For the first part of the assignment, we were tasked with optimizing the parameters of one of the neural networks used in Assignment 1. I chose to use the model built for the EEG Eye State dataset because it was more balanced than the other dataset. The model from Assignment 1 had two hidden layers, each with 128 nodes, used ReLU activations on the hidden layers, and used the Adam optimizer. I chose to optimize a neural network with two hidden layers of 64 nodes each, because it reduced the number of parameters by over 70%. In order to keep the comparison fair, I reran the network from Assignment 1 with the new hyperparameters. It should be mentioned that, since the parameters of a neural network are continuous values, the algorithms had to be configured accordingly. The fitness function I used calculated the inverse of the log loss. Log loss is often the value minimized when training classification models, and since I decided to maximize fitness (as opposed to minimize cost), I used took its inverse.

A. *Neural Network*

Since the purpose of Part 1 is to compare the performances of the random search algorithms with the neural network from Assignment 1, I will start by presenting those results to give a baseline for the upcoming comparisons. As previously mentioned, I used a smaller network for the optimization, so I reran the neural network using the new configuration. Overall, the results were very much the same as for the larger network.

The learning curves of the neural network are shown in Figure 1. The dip in the training curve at 6000 samples is strange, but it continues to climb with the remaining sample size. This could just be due to an odd split of the training data. I also find it interesting that both the training and validation accuracies are continuing to increase together. This seems to mean that more training data is needed to fully take advantage of the model's predictive power.

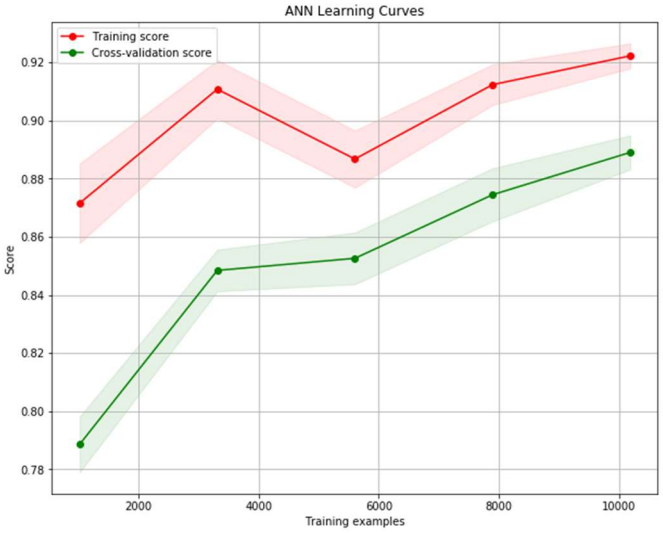

Fig. 1. NN Learning Curves

Figure 2 shows the ROC curve and AUC value of 0.96 for the neural network. This is actually better than the AUC of the larger network used in Assignment 1, which was 0.96.

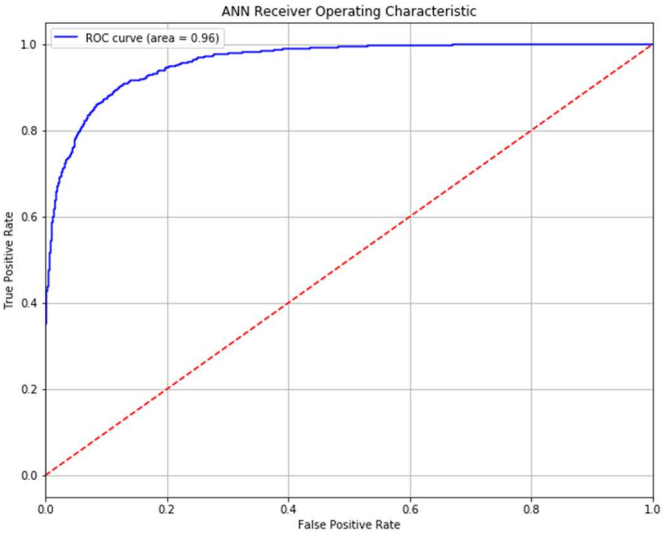

Fig. 2. NN Receiver Operating Characteristic

The confusion matrix for the neural network is shown in Table I. The accuracy was 0.8870, the precision was 0.8591, the recall was 0.8839, and the F1 score was 0.8713. This model had a fitness score 5.765. I should mention that the metrics for the larger network were worse across the board, with accuracy of 0.8629, precision of 0.8389, recall of 0.8458, and F1 score of 0.8424. None of these differences is major, and it could easily be attributed to differences in the splits of the data. However, it's worth noting that a much smaller network is producing results on par with the larger one.

TABLE I. EEG NN CONFUSION MATRIX

	Actual Positive	Actual Negative
Predicted Positive	860	113
Predicted Negative	141	1133

B. *Randomized Hill Climbing*

One of the first things to consider when using randomized optimization techniques is how to select an initial solution. For discrete-valued features it is straightforward, because there are options from which to choose. With continuous-valued features there are infinite such choices, which is the case with neural networks. Fortunately, there is a convention for initializing the weights of a neural network, and that is what I used. I initialized the weights from a normal distribution with a mean of 0 and standard deviation of 0.1. Another important consideration for RHC is the neighborhood that will be used for exploration. For discrete-valued features, this usually means iterating through each possible value and evaluating the fitness with that value. However, for continuous-valued features this is not feasible. I tried a few different methods for choosing a neighborhood, but I eventually decided to generate a value from the same normal distribution used to initialize the weights. I also decided to let the algorithm run for a specified number of iterations rather than terminating as soon as the solution stopped improving. Otherwise, the searches terminated rather quickly. My guess is that, due to the large number of parameters and the limited size of the neighborhood, it was difficult to find a new value for a parameter that caused an improvement, so the search ended. I even tried creating neighborhoods of up to 100 values generated by standard uniform distribution between [-1, 1] and evaluating each of the solutions produced with them. Nothing seemed to help. By allowing the search to run for a specified number of iterations, I could avoid early termination. However, I am aware that doing so shifts away from hill climbing and starts to resemble an exhaustive search. It was a tradeoff I consciously made for this problem.

Running the search for 10,000 iterations with 10 restarts, I was able to find a solution with a cost of 1.650, which resulted in the following confusion matrix. The accuracy was 0.683, the precision was 0.654, the recall was 0.566, and the F1 score was 0.607.

TABLE II. RHC CONFUSION MATRIX

	Actual Positive	Actual Negative
Predicted Positive	551	291
Predicted Negative	422	983

I also logged the times at which better solutions were found to try to get an idea of how long the search would need to find a solution that matched that of the neural network. We can see that the search improved steadily for the first few minutes, then it plateaued with only tiny, incremental improvements, before climbing again at

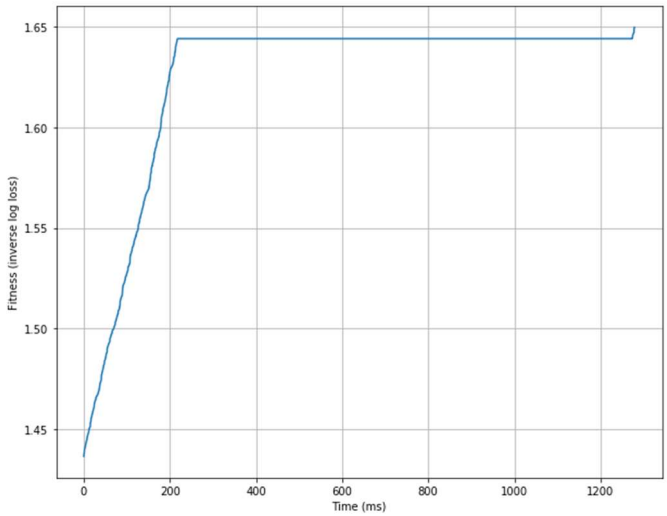

Fig. 3. RHC Fitness vs. Time

C. *Simulated Annealing*

Like with RHC, I initialized the weights from a uniform distribution with a mean of 0 and a standard deviation of 0.1. A big difference from RHC was the choice of neighborhood. I decided to generate 6 values from the same distribution used to initialize the weights. I then added the value to the current parameter value and evaluated the fitness. Because the search does not terminate if no good neighbor is found, I could afford consider a set neighbors that offered no improvement to the current solution. For SA, the biggest factors to finding better solutions were the starting and final temperatures and the cooling factor. One of the problems I was having early on is that too many bad solutions were being accepted. By the time the search started rejecting poor solutions, the quality had degenerated so badly that it did not have enough time to recover and find good solutions. The differences in the fitnesses were so small and the temperature so high that everything had an acceptance probability of nearly 1. To remedy this I reduced the starting temperature and final temperature, and I changed the way the acceptance probability was calculated. Rather than simply take the difference of the two fitnesses, I multiplied the difference by 1000 to attenuate the acceptance probability early in the search.

The final hyperparameters were a starting temperature of 1, a final temperature of 1e-5, a cooling factor of 0.999, and a maximum of 50,000 iterations. The results below were produced with this configuration. The solution quality was 2.400, and the confusion matrix can be seen in Table III. The accuracy was 0.801, the precision was 0.773, the recall was 0.764, and the F1 score was 0.768. These numbers are much better than the RHC and could possibly approach the neural network if given more time.

TABLE III. SA CONFUSION MATRIX

	Actual Positive	Actual Negative
Predicted Positive	743	230
Predicted Negative	218	1056

The fitness over time using SA can be seen in Figure 4 below. Unlike with RHC, it shows constant improvement throughout.

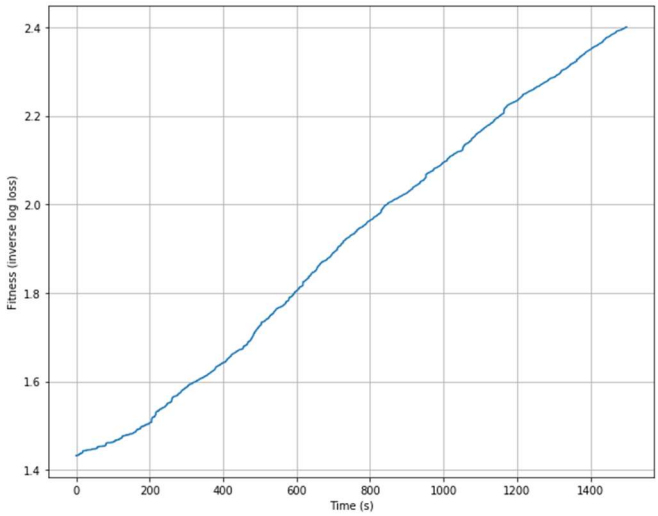

Fig. 4. SA Fitness vs. Time

D. Genetic Algorithm

Genetic Algorithms require more design decisions than RHC and SA, the main ones being the population size, how to initialize the population, the mutation rate, and how to perform crossover. I decided on a population size of 200, a mutation rate of 0.1, and maximum number of generations as 1000. I also initialized the population using a normal distribution with mean 0 and standard deviation 0.1. To perform crossover, I opted for uniform crossover. For each element, I generated a number from a uniform distribution between [0, 1]. If the number was less than 0.5, child 1 got its corresponding gene from parent 1, otherwise it got its corresponding gene from parent 2. Child 2 got its gene from the other parent. Because we are tuning weights of a neural network, which are stored as matrices, it made a little more sense to me than simply splitting the solution in half and giving one half to one child and one to the other. I originally tried to construct a solution as a list of weight and bias matrices and only combined weights and biases with those from the corresponding layer of the other parent. Intuitively this makes more sense to me, because the weights in a layer work together, and combining the weights from hidden layer 1 with the biases from hidden layer 2 just doesn't seem like an effective strategy. Unfortunately, the proposed method was too costly computationally, and I opted for the uniform crossover method. This way, at least some of the underlying structure of the parameters is maintained in crossover.

Table IV shows the confusion matrix for the genetic algorithm. The corresponding metrics are accuracy of 0.720, precision of 0.678, recall of 0.672, and F1 score of 0.675.

TABLE IV. GA CONFUSION MATRIX

	Actual Positive	Actual Negative
Predicted Positive	654	319
Predicted Negative	310	964

Figure 5 shows the fitness of the fittest member of the population as a function of time. We can see that it started out a little slow but really started to pick up after about 10 minutes, then started to slow again around 30 minutes in. I suspect this is due to the properties of the genetic algorithm, particularly with continuous features. The slow start makes sense, since everything was initialized randomly from an infinite number of possibilities. It took a few generations for the good candidates to start to exert an influence over the entire population. The slow down at the end also seems intuitive, because at some point, combining values from existing solutions will no longer produce improvements if the correct values are not part of any solution. The only way to get new values is through mutations, which was set to 10% for this exercise. Perhaps for continuous features it would make sense to increase the mutation rate as progress slows or maybe to sporadically "mutate" entire children to add some more diversity to the gene pool, so to speak.

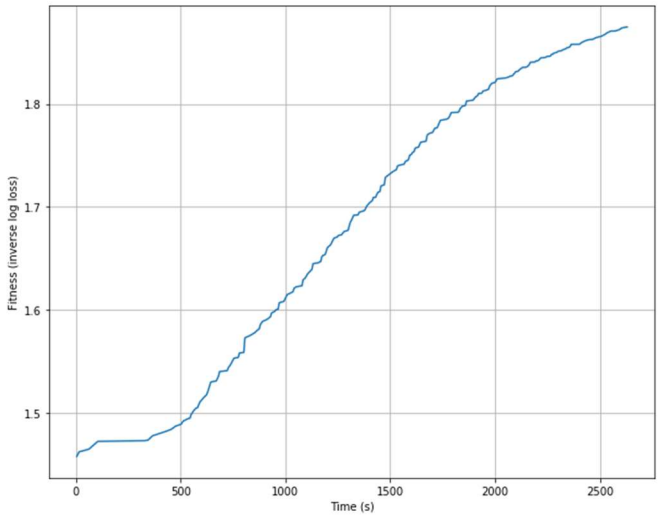

Fig. 5. GA Fitness vs. Time

E. Analysis

I think the biggest takeaway from this part of the assignment is simply how impressive backpropagation and gradient descent really are. It only took 10 seconds to train the neural network to achieve an accuracy of 88%. The three randomized optimization algorithms each ran for over 20 minutes and the best result was 80% accuracy. If a problem is differentiable, that is the route I would take. However, this is not to say that randomized optimization algorithms are never useful. There are plenty of cases where a solution has no derivative. Most standard optimization problems simply want to maximize or minimize a value, with no notion of what the correct or best answer is. Other problems, like many NP-Complete problems, are intractable and cannot be solved exactly. Often, we must settle for a "best guess." This is where randomized optimization algorithms are useful, and that is the focus of Part 2.

IV. Comparison of Algorithms

For Part 2 of the assignment, we are to compare 3 different problems that highlight the strengths and weakness of the four algorithms detailed above. The first problem highlights the advantages of the genetic algorithm, the second highlights the benefits of simulated annealing, and the third highlights the strengths of MIMIC.

A. Max One (or Max Sum)

The first problem is a relatively easy one. The objective of Max One is find a solution with the maximum number of ones. Of course, the optimal solution is to have n ones for array of length n. It is interesting problem because, as humans, we already know what the optimal solution is. But it still does a great job of demonstrating the qualities of the different search algorithms. I ran the algorithms with n set to 50.

First, we have the results of the fitness over time for MIMIC. The hyperparameters used were a population size of 200, and a top-N of 0.25. We can see the average fitness of the population increases until it reaches the optimal solution of 50 at around 20 seconds. While the solution quality is great, the time needed to achieve leaves something to be desired.

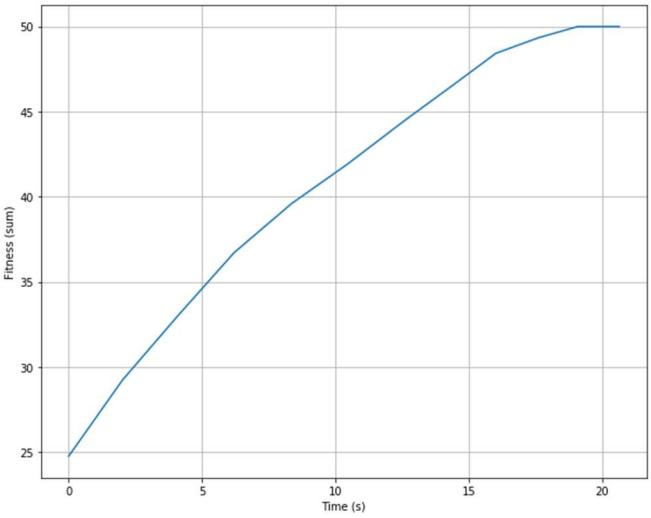

Fig. 6. Max One MIMIC Fitness vs. Time

Next we have the randomized hill climbing algorithm. We used 50 restarts for run. The first that jumps out is the big increase at the beginning and then again at the end. The initial solution was pretty evenly split, so the algorithm must have gotten lucky and selected six elements in a row that were 0. The next thing to notice is how quickly the search terminated. All 50 iterations were done in only 16 ms. I ran this a few times with various numbers of restarts, and the results were almost the same each time. Even increasing the number of restarts to 2,000 only resulted in a fitness of 35. It seems that, once about 2/3 of the elements are flipped to 1s, the probability of selecting an element that is already a 1 is high. The search cannot improve, so it terminates. RHC is very susceptible to local optima in this problem.

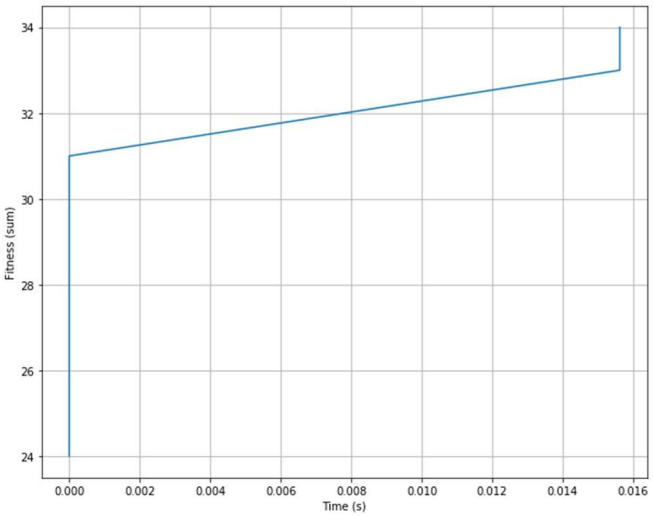

Fig. 7. Max One RHC Fitness vs. Time

The next algorithm run for this problem was simulated annealing. A starting temperature of 100, a final temperature of 0.01, and a cooling factor of 0.99 were used. he hyperparameters. Its results combined the correctness of MIMIC with the speed of RHC, resulting in an optimal solution in only 16 ms.

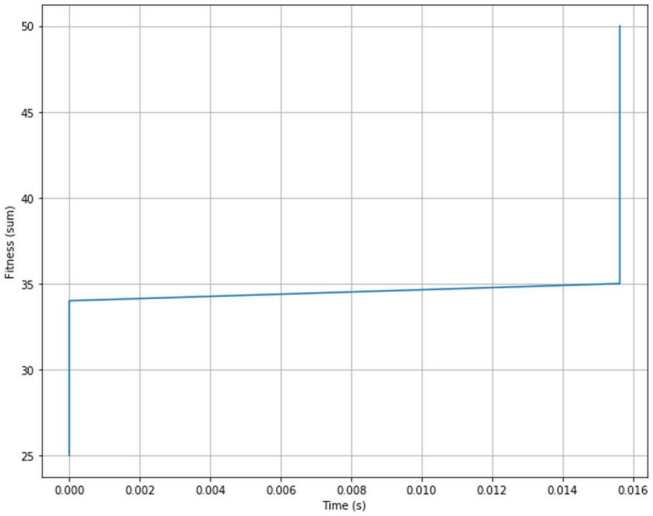

Fig. 8. Max One SA Fitness vs. Time

Finally, I took a look at the genetic algorithm. I used a population size of 30 and a mutation rate of 0.1. It returned the optimal solution in about 48 ms, which is just a little slower than SA. However, it still returned the optimal solution, which RHC did not do, and it did so much faster than MIMIC. So while the genetic algorithm did not perform the best overall for this problem, I still think it did a good job of highlighting the strengths of genetic algorithms. In this case, there was no underlying structure to be concerned with. An element was not related to its neighbor, so the crossover could be performed in

any manner without disturbing some underlying property of the solution.

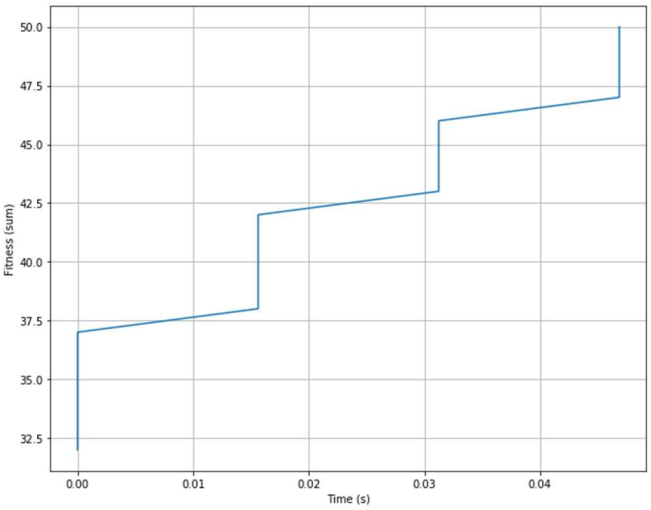

Fig. 9. Max One GA Fitness vs. Time

B. Traveling Salesman Problem

The second problem I chose to analyze is the Traveling Salesman Problem, or TSP. The goal is the find the minimum cost path that visits each node in a graph only once. There are many variations of this problem, but I chose to use a metric TSP, meaning the triangle inequality holds. The graph is complete, so every pair of vertices is connected by an edge, and it has 10 nodes. The minimum cost path is known to 277,953. TSP is probably one of the more famous NP-Complete problems, and much time and effort has been spent studying it. The problem is hard because the number of solutions is factorial in the number of the nodes in the graph. For small n it can be solved using exhaustive search (or branch and bound methods), but as n grows this quickly becomes intractable, which makes it a great candidate for randomized optimization. As is often the case for TSP, for RHC and SA, the neighborhood was defined as two uniformly randomly selected nodes. The positions of the nodes in the solution were swapped, and the fitness was measured. If the solution quality improved, the new solution was kept. If not, it was discarded. For the genetic algorithm, crossover was done by taking half of one parent in order, and then taking the remaining nodes in the order in which they appear in the second parent. Mutations were performed by swapping the positions of two nodes. For MIMIC, a modification was made to the way the samples were generated. Since a node can only appear once in the path, the probability of a node being selected was set to 0 once it appeared in the sample.

The first algorithm I ran was MIMIC. I once again used a population size of 200 and a top-N of 0.25. This curve very closely resembles the curve from the Max One problem above. MIMIC converged very quickly this time, in about 2.5 seconds, to a path cost of 286,474, which is within about 3% of the optimal solution.

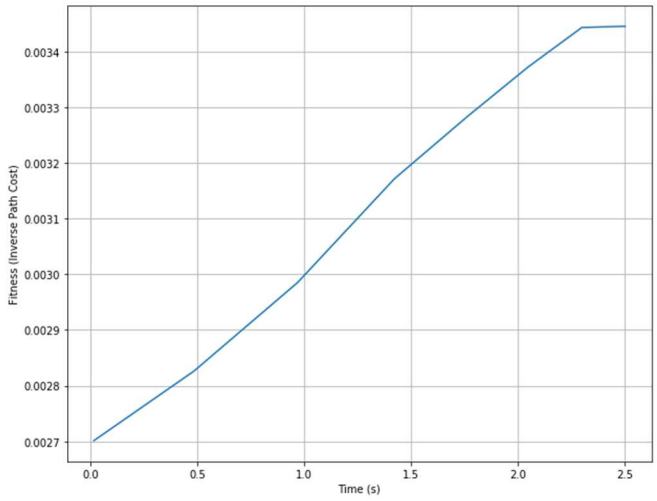

Fig. 10. TSP MIMIC Fitness vs. Time

RHC was the second algorithm run on the TSP. We see similar results as with the Max One. It makes an initial improvement, but due to the way the neighborhood is defined, it is very difficult to find improvements, and the searches terminate very quickly. It might make sense to run the search for a certain number of iterations like in Part 1 or to expand the neighborhood. RHC produced a minimum path cost of 294,877, which is within about 6% of the optimal solution, or about twice as bad as MIMIC.

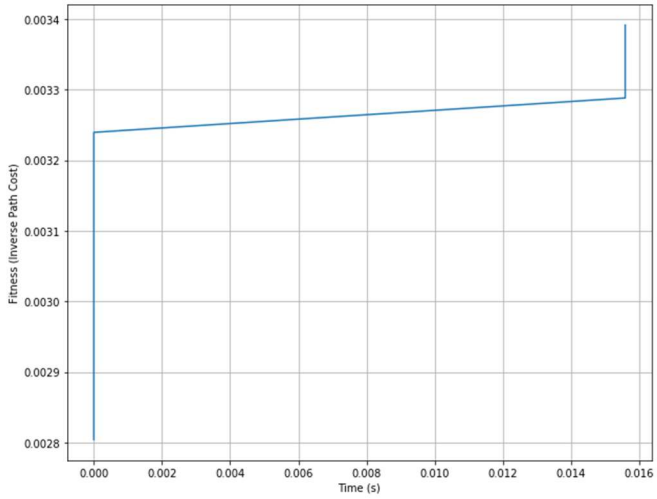

Fig. 11. TSP RHC Fitness vs. Time

Next, I ran simulated annealing. The starting temperature was set to 100, the final temperature was set to 0.01, and the cooling factor was set to 0.999. SA was able to find the optimal solution in just under 0.5 seconds.

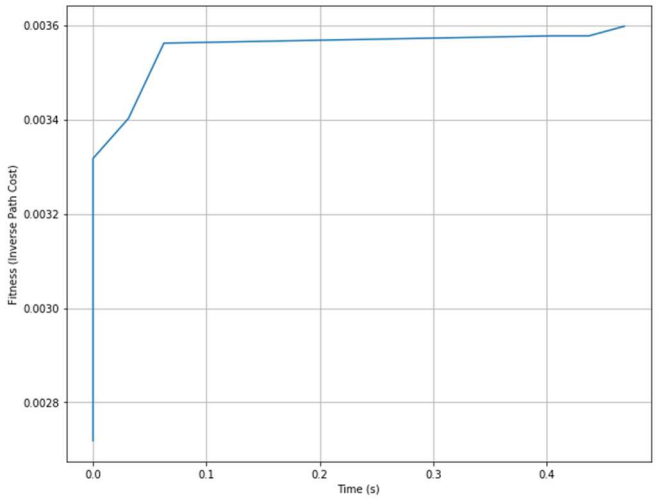

Fig. 12. TSP SA Fitness vs. Time

Finally, I ran the genetic algorithm. I used a population size of 200 and a mutation rate of 0.1. It returned the optimal solution in about 0.23 s, which is faster than simulated annealing. However, there are a few reasons why I still think simulated annealing is the better algorithm for TSP. The first is that, in under 0.1 seconds SA has a solution that was very close to optimal. It steadily improved upon that until finding the optimal solution at around 0.45 seconds. While GA did find the solution faster, its quality seems to be a bit lower for a bit longer than SA. I know we are only talking about fractions of seconds here, but on larger problems, there are often limits to how long the algorithm can run. Sometimes finding a very good solution very quickly is better than finding the best solution slowly. And I think that is where simulated annealing is slightly better than genetic algorithms. The other reason I prefer SA is that it consistently returned the optimal solution. GA did quite regularly, but in multiple instances where it did not. Perhaps a better chart would be to show the average result over multiple runs, instead of just the best result. For further analysis this is something I would consider.

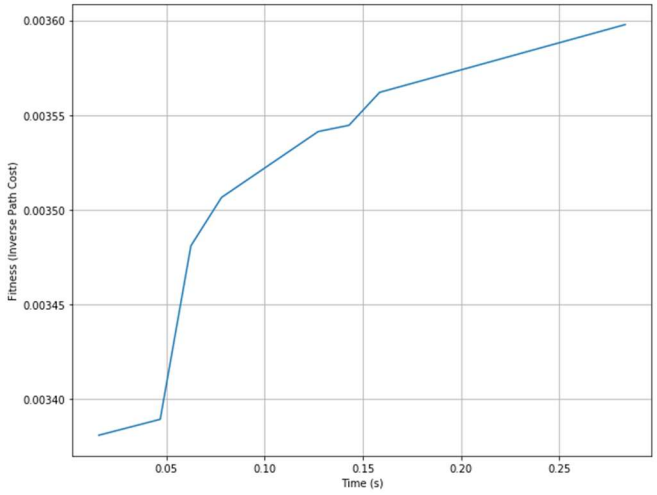

Fig. 13. TSP GA Fitness vs. Time

C. *4 Peaks*

The final problem I chose to analyze is the 4 Peaks problem, which highlights the strengths of the MIMIC algorithm. Given an N-dimensional input vector, the 4 Peaks evaluation function is

$$f(X,T) = \max[tail(0,X), head(1,X) + R(X,T) \quad (2)$$

where

$$tail(b,X) = \text{number of trailing } b\text{'s in } X \quad (3)$$

$$head(b,X) = \text{number of leading } b\text{'s in } X \quad (4)$$

$$R(X,T) = \begin{cases} N \text{ if } tail(0,X) > T \text{ and } head(1,X) > T \text{ or} \\ \quad tail(1,X) > T \text{ and } head(0,X) > T \\ 0 \qquad\qquad otherwise \end{cases} \quad (5)$$

The name comes from the 4 local maxima, or peaks, two of which are also global maxima. The maxima are achieved when there are T+1 leading 1's followed by all 0's or vice versa. The local maxima are found when there are only 1's or 0's. For my implementation, I set N equal to 20.

MIMIC was once again the first algorithm tested, and I once again used a population size of 200 and a top-N of 0.25. The optimal solution of 37 was found in just under 4 seconds. One thing I noticed is that, as I increased N, I became less likely to find the optimal solution with MIMIC. I find this interesting particularly because the solution returned was still always better than the local maxima. It was always greater than N but still suboptimal. I started wondering if there is a bug in my implementation, but I do return the optimal in many other situations, and the algorithm's performance on the other problems was as expected. This is something I would further explore given more time.

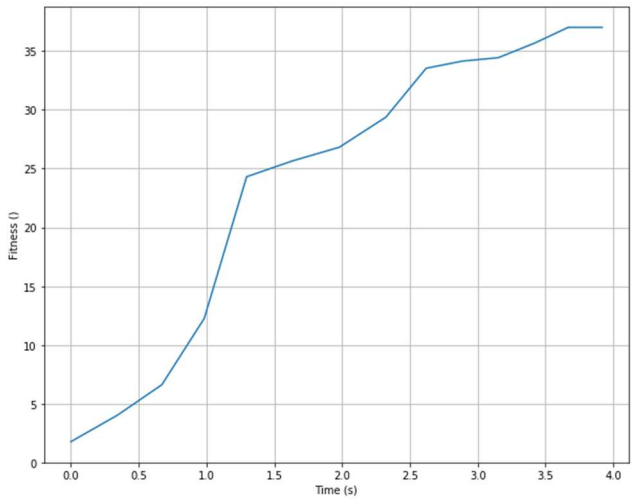

Fig. 14. 4-Peaks MIMIC Fitness vs. Time

The results of RHC were the same as with the other two problems. Clearly, the implementation was suboptimal given the problems. And though it looks as though only two solutions were found, the algorithm did in fact return seven solutions. It

just found them so closely after one another that it hardly shows on the plot.

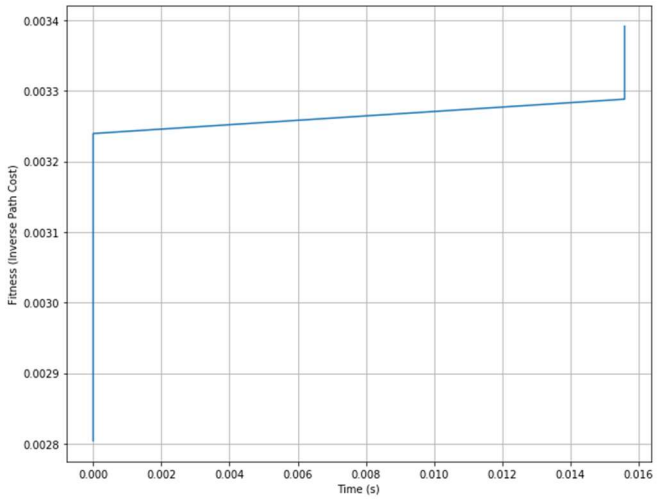

Fig. 15. 4-Peaks RHC Fitness vs. Time

Next, we have simulated annealing, which once again was the best performer. The starting temperature was set to 100, the final temperature was set to 0.001, and the cooling factor was set to 0.999. SA was able to find the optimal solution in 31 ms.

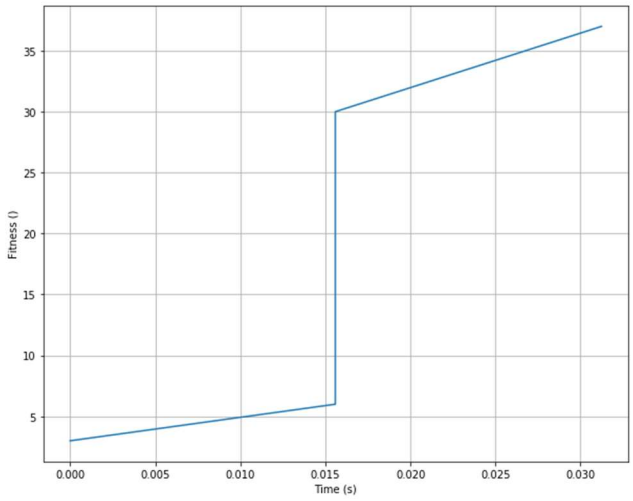

Fig. 16. 4-Peaks SA Fitness vs. Time

Closing it out was once again the genetic algorithm. The population size was set to 150 and the mutation rate to 0.1. The optimal solution was returned in about 16 ms.

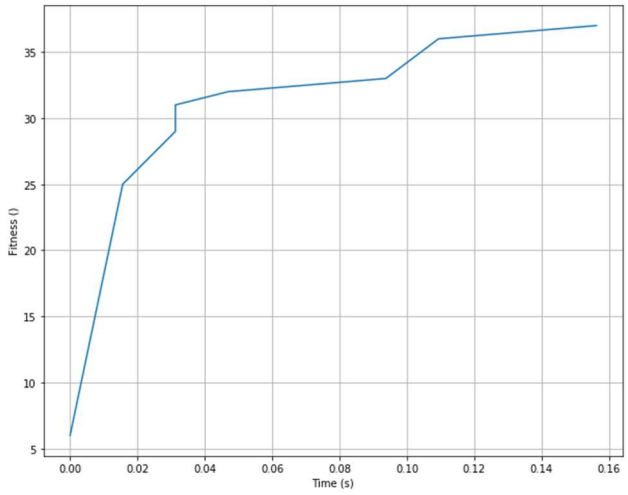

Fig. 17. 4-Peaks GA Fitness vs. Time

This problem was chosen to highlight the strengths of MIMIC, and once again a different algorithm performed better. Once again, I will attempt to justify this selection. MIMIC is preferable when there is an underlying structure to the solution. This is clearly the case with 4-Peaks, as what is happening in one part of the solution directly impacts what is optimal in other parts. It is also beneficial when evaluating the fitness function is costly. This is also the case with 4-Peaks, as the fitness function is comprised of 3 smaller functions. For this problem, a better metric may have been function evaluations, as was done in the paper.

V. Conclusion

As mentioned in Part 1, the first thing that was really emphasized was the effectiveness of gradient descent in finding optimal solutions, though this wasn't really a surprise. It was actually quite interesting to see randomized optimization algorithms used in that way and to see the relatively good results, though at a high cost of time. As is often the case, each algorithm discussed has its own strengths and weaknesses and should be evaluated accordingly. Since this was my first time using MIMIC, I would like to experiment with it more, particularly on problems like TSP, in which the structure of the solution is important. Finally, when it comes to randomized optimization algorithms, though maybe not always the best, simulated annealing seems like a good place to start.