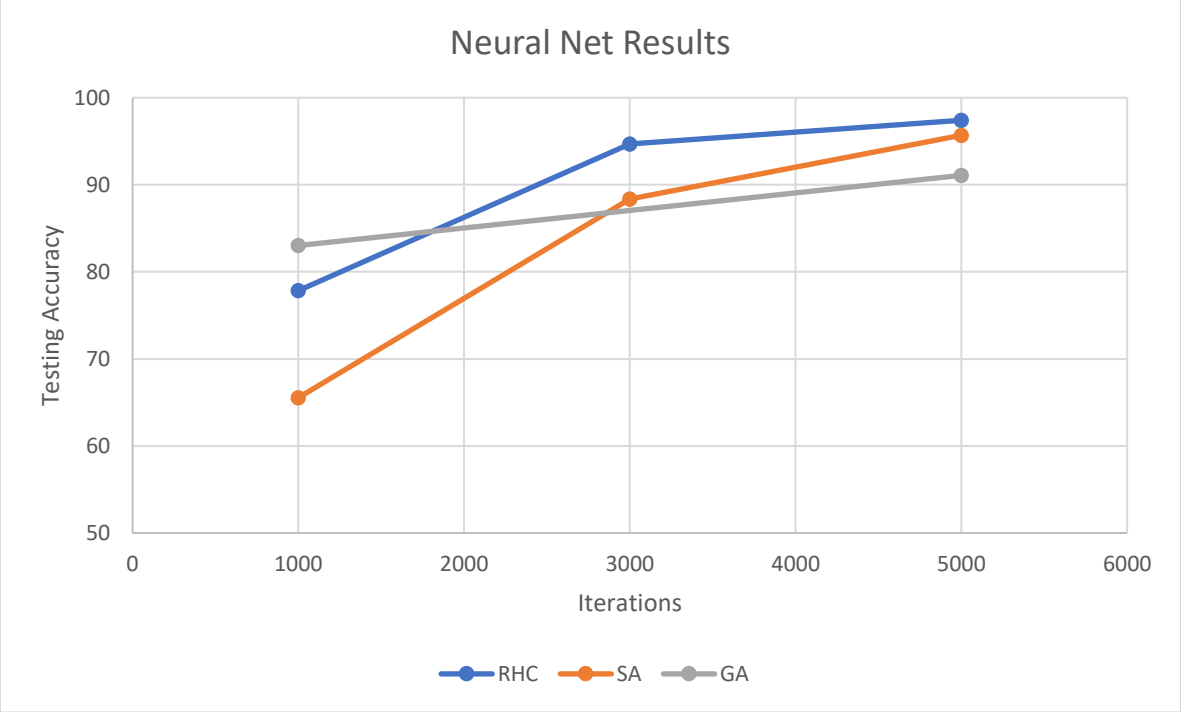
Pranshav Thakkar

GT username: pthakkar7

Randomized Optimization

For the first part of this assignment, we were tasked with using randomized optimization algorithms in place of back propagation to find good weights for the neural net that we used in the first assignment. I used Randomized Hill Climbing, simulated annealing, a genetic algorithm and MIMIC from the ABAGAIL library that was provided to us in the assignment description. I had used scikit-learn for the first assignment so I attempted to recreate the same neural net, with 15 hidden layers in this new environment.

Results:



For reference, back propagation gave me a testing accuracy of 96.97% with this neural net in the first assignment. Here, we see that each of the three algorithms improved as we increased the number of iterations, which makes sense as it gives the algorithms more information to work with and converge. My genetic algorithm gave me the best testing accuracy with 1000 iterations and ended up with the worst relative testing accuracy over 5000 iterations, but it was still around 91%. Randomized Hill Climbing gave an accuracy that was a little lower then GA at 1000

iterations but rapidly increased as the number of iterations increased to 3000 and ended up being the most accurate at 5000 iterations. Simulated annealing was considerably worse in terms of accuracy than the other two at 1000 iterations, but the accuracy also increased and ended up being slightly less than RHC at 5000 iterations.

I believe that the genetic algorithm increased in accuracy at a lower rate than the others because of the nature of the algorithm, where it has to maintain a population size where there are a fixed number of members that mate and a fixed number of mutations that occur. So, increasing the number of iterations helped the algorithm perform better, but not as exponentially as the other two algorithms. This algorithm could potentially have performed better in terms of accuracy if the population size had been increased slightly and the number of members that mated had been increased. This would provide a larger set of members to pick the best from while also increasing the variety of the population.

RHC and SA increased their accuracies at a high rate as the iterations increased in part due to the nature of the dataset. In particular, my dataset had 2500 training instances and 695 testing instances, while having 36 attributes. This makes it easier for these algorithms to converge quicker, as RHC just moves towards the best local point for each iteration, so the larger the number of iterations, the better the chance of it finding the best global point and climbing towards it. SA is similar to RHC in that it converges faster as the number of iterations increase. This algorithm was worse than the other two when I ran 1000 iterations, because it depends on a probability that is dependent on the temperature parameter. With fewer iterations, the temperature value has a larger fluctuation as it changes, and thus the probability can give results that are varied. However, increasing the iterations will make that probability stricter, and the algorithm can converge on what it believes to be the right values. The relatively small dataset also helps these two algorithms converge faster as there is less data for them to “climb”. Although, the dataset might have been another reason that the genetic algorithm improved its accuracy slower than the others, as there is a large attribute to instance ratio. This matters for the genetic algorithm as it has to consider the attributes while performing the crossovers, and this ratio makes it harder for it to distinguish between the best population members as fast as the other algorithms when the number of iterations increases.

Simulated annealing could have performed better if I had provided the optimal temperature to start at as the parameter. Testing more values of temperature would have helped me determine this, as there could be a certain temperature value that would help the algorithm converge faster or avoid having the lowest accuracy of the three algorithms at 1000 iterations. The correct temperature would help avoid the uncertainty that constituted that low accuracy. Another parameter I could have explored is the cooling for simulated annealing, which I did not vary in this assignment but would have been interesting to see if it made a difference at a certain number of iterations or a difference over time.

Runtime:

BackProp
5 sec

Iterations	GA
1000	2085.784 sec
5000	8728.068 sec

Iterations	RHC	SA
1000	46.4485 sec	45.923 sec
3000	121.471 sec	121.142 sec
5000	262.896 sec	258.386 sec

In terms of runtime, back propagation was by far the fastest, running in around 5 seconds. Randomized Hill Climbing and Simulated Annealing were very similar in terms of time, no matter how large the iterations were. They were still slower than back prop, but much, much faster when compared to the genetic algorithm. The genetic algorithm took around 35 minutes for 1000 iterations and about 2 and a half hours for 5000 iterations. I assume that the nature of my dataset is the reason why the genetic algorithm took such an abnormally long amount of time to run.

Ultimately, RHC and SA ended up performing about the same as back propagation at 5000 iterations while GA had about a 5% lower accuracy at 5000 iterations. This leads me to believe that higher iterations would lead to the algorithms converging further and delivering a test accuracy that is equal to or higher than back propagation. The key factor in this case, however, is time. Even for a relatively small dataset like the one I used, GA ended up being far too costly in terms of time and so I do not think it is the best algorithm in this case. I believe that using RHC or SA would be a better option if they are able to consistently deliver a better performance than back prop, even if they are slower. Running them for 10000 iterations, for example, could be a reasonable trade-off.

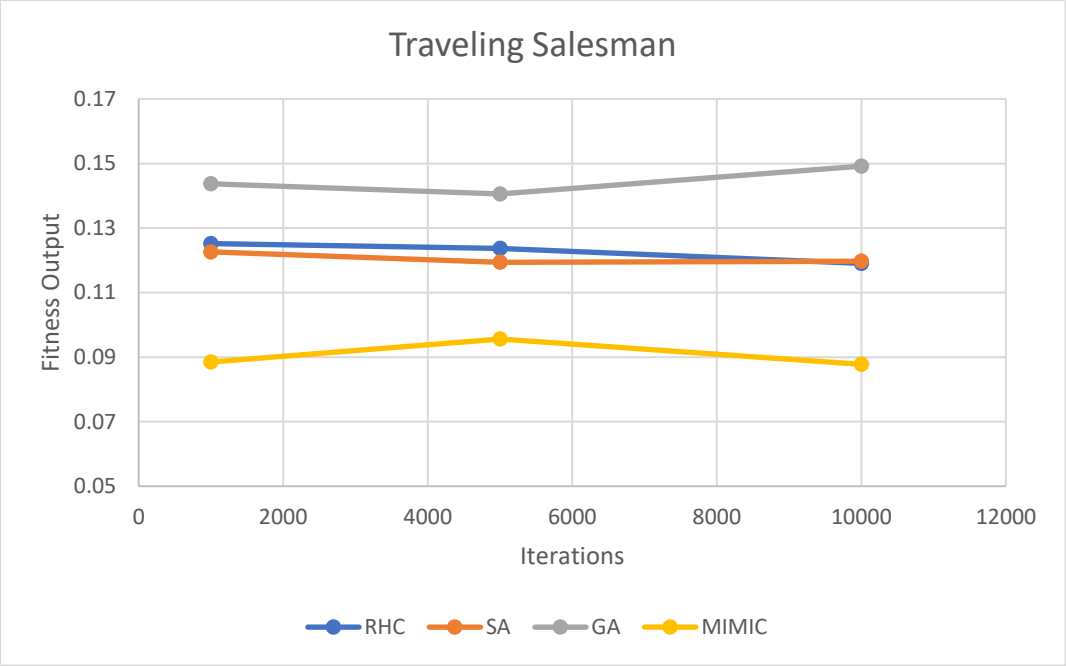
Optimization Problems:

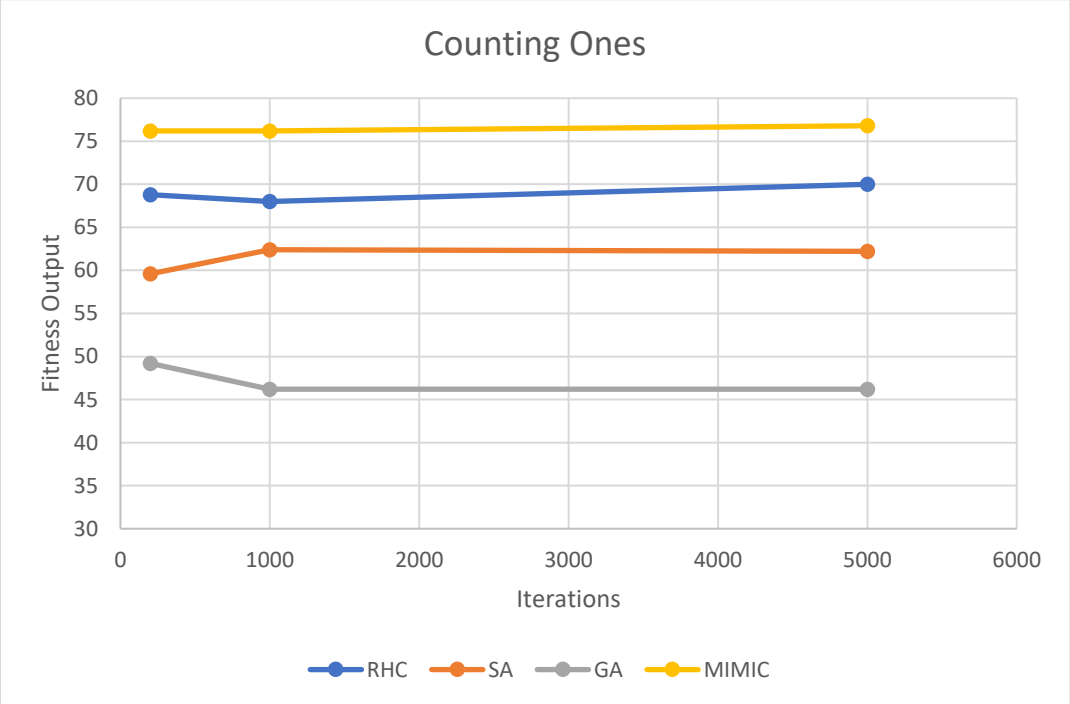
1. Traveling Salesman: The Traveling Salesman problem is the classic problem where there are an N number of cities, we are given the distance between each pair of cities, and we want to find the path of shortest distance where we visit each city and return to the origin city. This is a combinatorial problem and that is why genetic algorithm performs the best on this problem, as GA is a combinatorial algorithm with its mutations and crossovers that provide many variations to compare. This correlates directly to the problem where we need to consider the many paths available to us and pick the one with the shortest distance. It is interesting to see if RHC, which performs well with local minima/maxima, is able to string together the local optima to give the overall result as good as GA, or if it falters.

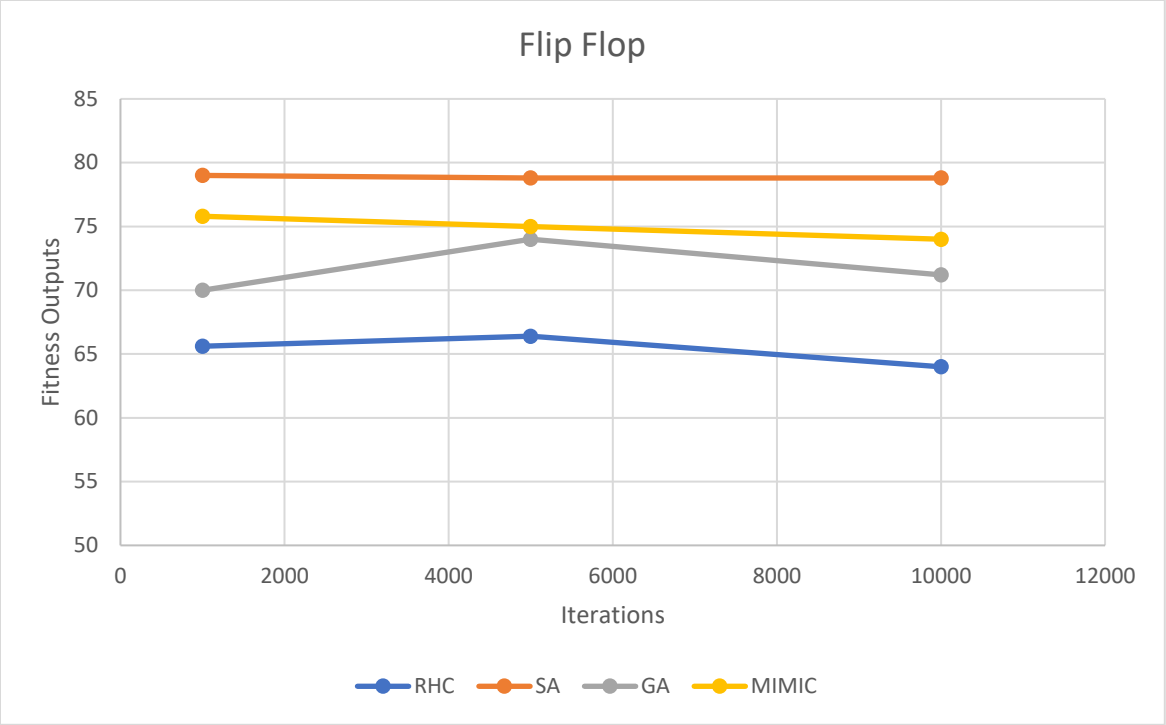
2. Flip Flop: The Flip Flop evaluation function takes in an N length bit string and keeps track of the number of times that the bit string flips its value from one digit to the next. For example, if a digit has the value 1, and the next digit is 0, we would increase the counter. If the current digit and the next digit are the same, we would not increase the counter. I think simulated annealing is a good algorithm for this problem as it correlates well with the 0 and 1 aspect of the bit string. Simulated annealing is probability based, and it differentiates between a probability of 1 if a condition is met and a probability based on the temperature function. That could work well in determining how many times the bit string flips based on the number of 1s and 0s there are, and the probability given that a bit is a 0 or a 1. It would be interesting to compare how SA performs with regard to MIMIC, which uses conditional probabilities to form dependency trees.

3. Counting Ones: The Counting Ones problem takes in an N length dataset and counts the number of digits that have the value 1. This problem is interesting because you would think that RHC would perform very well in this, as brute force seems like the easiest way to do this. However, MIMIC actually performs the best because it takes advantage of the fact that it calculates the conditional probabilities of whether there is a 1 after the current item or not. Then it makes a dependency tree based on those probabilities and makes it easy to predict whether the next item is a 1 or not, and thus can predict the total number of 1s based on the length of the dataset

Results:


Traveling Salesman
Fitness Output
0.17
0.15
0.13
0.11
0.09
0.07
0.05
0
2000
4000
6000
8000
10000
12000
Iterations
RHC
SA
GA
MIMIC


Counting Ones
Fitness Output
80
75
70
65
60
55
50
45
40
35
30
0
1000
2000
3000
4000
5000
6000
Iterations
RHC
SA
GA
MIMIC



For all these problems, I ran the tests at each iteration 5 times, and then took the average of the 5 results and plotted them on the graphs.

For the Travelling Salesman problem, the fitness output is calculated by determining 1/(the shortest distance produced by the algorithm). So, while we want the shortest distance, the highest fitness output is still the desired output. As expected, the genetic algorithm provided the best fitness outputs consistently on a fixed number of N cities. Its performance was significantly better than the rest and that makes sense since the algorithm is suited to the problem's combinatorial nature. Randomized Hill Climbing and Simulated Annealing perform about equally to each other, and their performance is medial in terms of the other algorithms. MIMIC actually performs the worst here, and I believe that is because the cities and the paths between them are discrete values and do not have any conditional independence. Thus, MIMIC does not gain any advantage over the other algorithms and gives the worst fitness outputs.

MIMIC's capabilities are highlighted by the Counting Ones problem, however, where it returns the best fitness outputs out of all the algorithms. This problem asks to determine how many '1's there are in a given dataset. In this particular case, we have a dataset of size N = 80 and thus the optimum fitness output is 80. MIMIC makes use of its distribution to predict the likelihood of there being a certain amount of 1s and the fitness outputs are very close to 80. RHC actually performs the second best, as one might think and was described in the problem description. It was interesting to observe that GA performed significantly worse than the other

algorithms. On inspection, it makes sense as the mutations that occur in the algorithm may change some of the data from 1 to other values, harming the fitness output.

The Flip Flop problem used a bit string of length N = 80, making the optimum fitness output equal to 80. Simulated annealing was the best performing algorithm for this problem, making use of its ability to predict and move towards the local max of the number of bits that flip. This algorithm is suited well to the problem as essentially it is based on a binary probability distribution that it uses to make decisions, and here we need to figure out how many times a binary bit string flips. MIMIC, the only other algorithm that uses probabilities out of the four that we have been using, comes in second in terms of output. RHC comes in last here as there is really no local optima for it to move towards, as whether a bit flips or not in the next digit is not very discrete.

The runtimes for all three problems was negligible for RHC, SA and MIMIC due to the data inputs being very small for all the problems. The genetic algorithm averaged around 30 seconds for each problem and I believe that the reason has to be the time that it takes the algorithm to do all the work that it does: forming a population, mating members of the population, accounting for mutations, and then choosing the best members to form the new population.

To improve the performance of the algorithms, the most important tool to have is domain knowledge. Knowing about the problems and how the algorithms attempt to solve them provide a lot of insight into which parameters to tune for each algorithm. In this assignment, it would have been a good investment of time for me to have analyzed the problems in more depth. That would have saved me some time in picking which parameters to tune, and in some cases, would have informed me whether I needed to tune that parameter or not.

Randomized optimization showed me the importance of how changing the number of iterations can make a big difference in some cases (like where the RandOpt algos were able to reach and surpass the performance of back prop in neural nets) and can sometimes mean next to nothing (in the optimization problems I chose, there were minor fluctuations based on iterations). That is where domain knowledge shows its true power. #NoFreeLunch