Randomized Optimization
Tiffany Xu

Introduction
Randomized optimizations are useful for problems that gradient descent cannot solve or iterating through all possible values would take too long. When the input space is too large, when the function is very complex, or when the function is not differentiable, randomized optimization algorithms come into play. Below we will analyze the effectiveness of randomized hill climbing, simulated annealing, and genetic algorithms by using these to replace neural network's backpropagation to find weights. Then, randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC will be used on three different optimization problems to compare and contrast their effectiveness on these different types of problems.

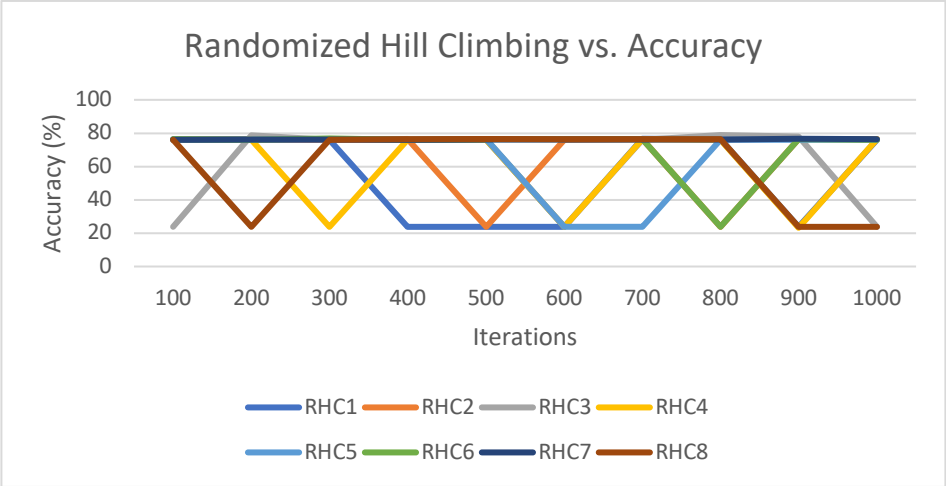
Part 1. Neural Network Weights with Randomized Optimization
The dataset reused from the supervised learning project is the income dataset. This dataset contains 14 attributes from a census survey (e.g. age, occupation, marital status, country) that are used to predict if a person's income is below or above $50,000. The structure of the neural network remains the same from the previous project with 11 hidden nodes in the hidden layer. The training and testing sets are split 80%, 20% respectively.

I used Weka to complete the previous project but will be using ABIGAIL for this project. Because ABIGAIL is easier to use with numerical attributes, I modified the non-numerical columns to be numerical, where each possible value in a column is replaced with a unique number corresponding to that value. I also reduced the amount of data by half to 16,280 because the full dataset was proving to take too long to run. The error measurements are mean squared error and accuracy measures are the total number of instances correctly classified over all instances.

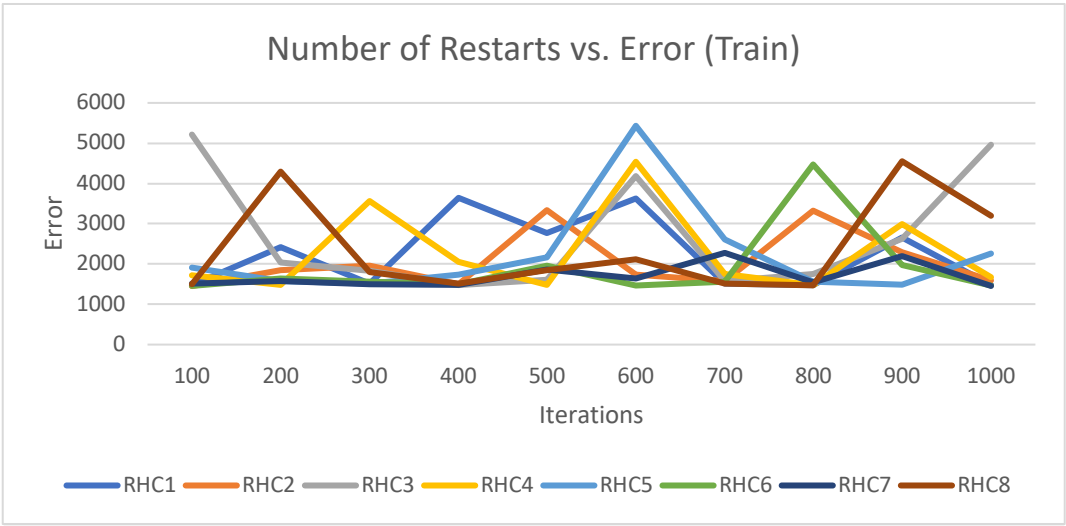
Randomized Hill Climbing
Randomized hill climbing works by randomly finding a position, calculating the values of all neighbors, and moving to the neighbor with the greatest value. This process repeats until the value cannot be improved and the algorithm stops. Although this algorithm is simple to understand and very fast to run, it is prone to landing in a local maximum. To combat this, the algorithm should have restarts to determine if the maximum found was a local maximum or a global maximum.

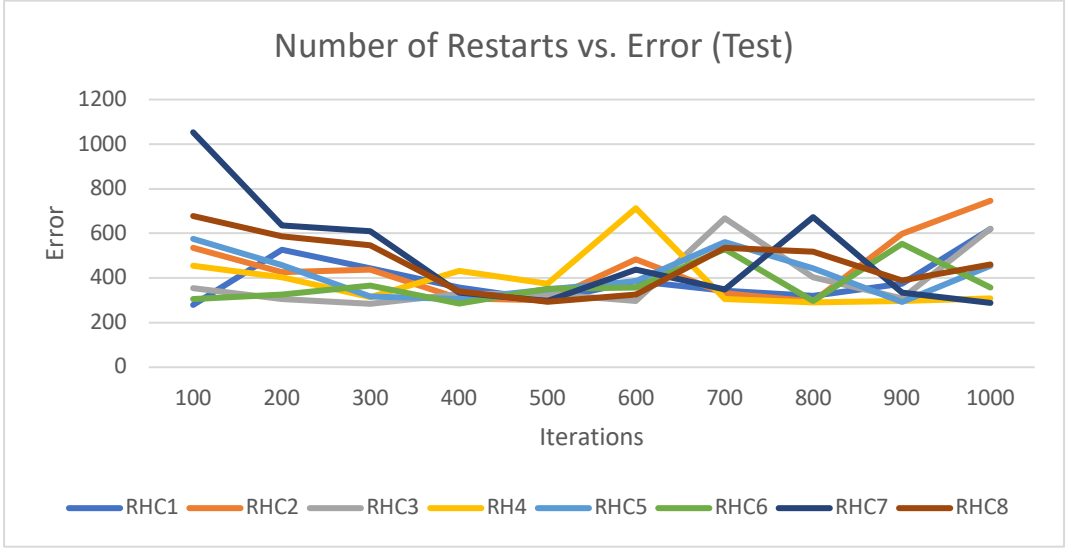
I restarted the randomized hill climbing algorithm 8 times and ran each for 1000 iterations. An interesting pattern appears where the values alternate between approximately 76% and 23%. It seems as if a local optima is found for the weights that correspond to the 23% accuracy, and the global optimal weights correspond to the 76% accuracy. This is most likely due to the fact that some attributes are more important than others in deciding the classification, and if bad weights are given to these attributes, the accuracy will be low. The vice versa is also true. This result is not surprising since randomized hill climbing is vulnerable to finding local optimum because it is just dependent on chance that the randomized point is with in the basin of attraction of the global maximum.

Randomized Optimization
Tiffany Xu



To get another perspective and better feel for the issue, I graphed the iterations against the mean squared error. There doesn't appear to be a trend in error as the iterations increase, but there does appear to have random spikes and drops in error, corresponding to the flip-flip in accuracies. The 7th random restart (dark blue) has the lowest error, but because there is no pattern in the number of restarts, I could have just as equally found the lowest error in the first random restart. On the testing data, the neural network actually performed better, so the weights were not overfitting and there was not too much variance.



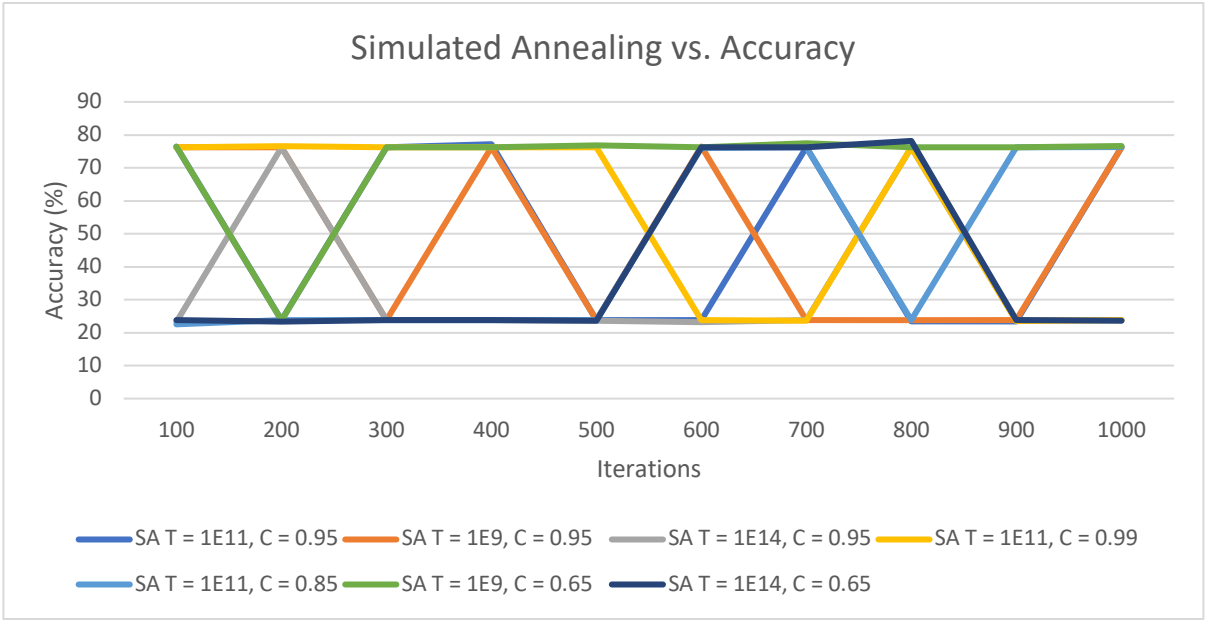


Randomized Optimization
Tiffany Xu

Simulated Annealing

Simulated annealing is an analogy for blade making, where the repeated heating and cooling of the blade yields an even stronger blade. The idea is that sometimes the algorithm should go in a worse direction to find an overall better solution. The "temperature" starts high and decreases based on a cooling rate. When the temperature is high, the algorithm acts like a random walk, exploring all valleys and parts of the function. As the temperature decreases, however, the barriers become greater and the algorithm starts to act like hill climbing. Thus, the algorithm is allowed to explore before it begins exploiting a subsection of the fitness function.

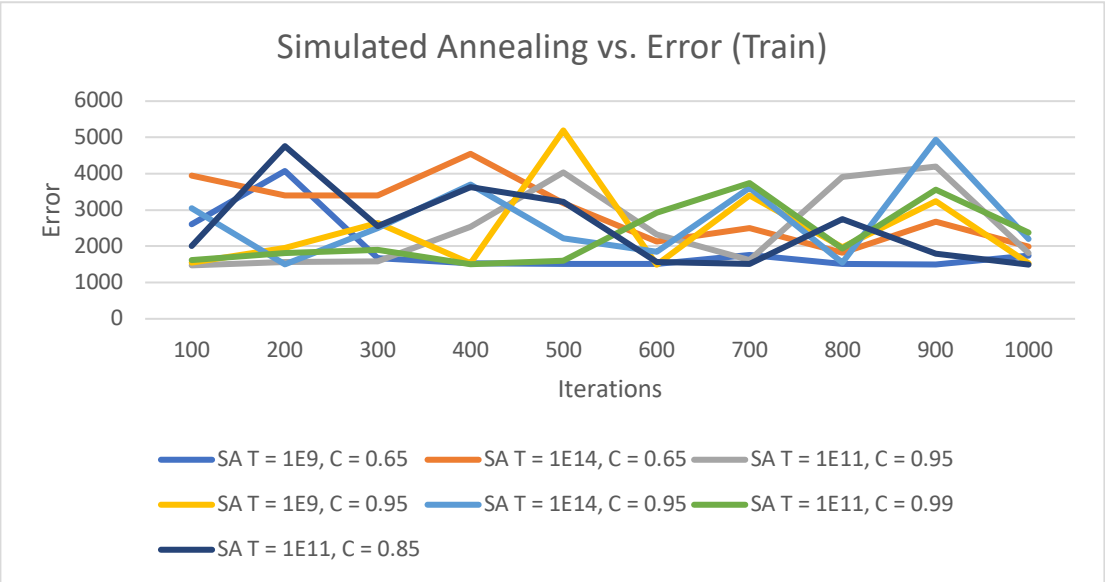
The two hyperparameters of simulated annealing are starting temperature and cooling rate. A higher starting temperature results in more exploration but the cooling rate determines how long this "exploration" phase lasts before points are only moved to if they improve the fitness. The lower the cooling rate, the faster the temperature drops. I changed the temperature among the values 1E9, 1E11, and 1E14, and the cooling rates as either .65, .85, .95, or .99. In the previous project, when there were multiple hyperparameters, I found the best value in one dimension, and using that value, I tried to optimize the other dimension. This was a flaw in logic and a random search among the parameters would've been better because the global optimum configuration isn't guaranteed to be along the maximum of a single dimension. Thus, for this project I used random combinations of the two parameters. The graph below has the same truss shape found in the randomized hill climbing section. I thought simulated annealing would do better at this section because the change in temperature allows for more exploration and shouldn't be as vulnerable to local optima, but perhaps the starting temperatures were too high and caused the algorithm to jump around too much.

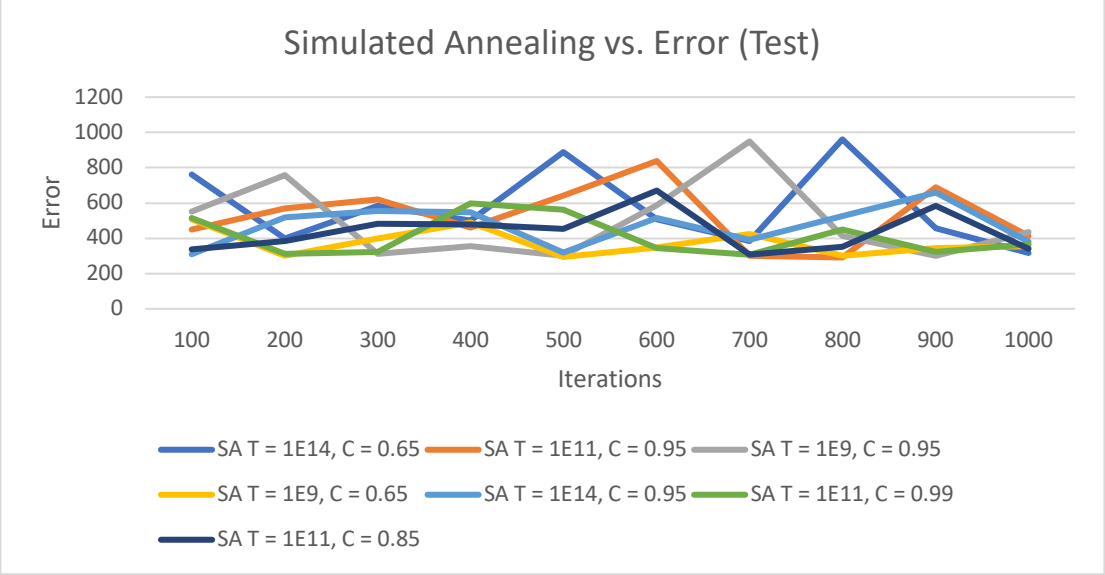


If we look at the graph below with the sum of squared error, it is a bit easier to determine which configuration is best. The medium blue line in the training graph with the starting temperature of 1E9 and a cooling rate of 0.65 has a consistently lower error after the 300th iteration. In the test set, this configuration also had a consistently lower error than the other configurations.

Having a lower starting temperature and a relatively fast cooling rate did better by not acting like a random walk, and quickly narrowing down the scope of the function explored with the reduction in temperature. Similar to the randomized hill climbing section, the error for the test set also decreased substantially, but the range of error among the configurations remain stable.

The weights are good in the sense that they do not overfit the data, since they generalize better to the test data.



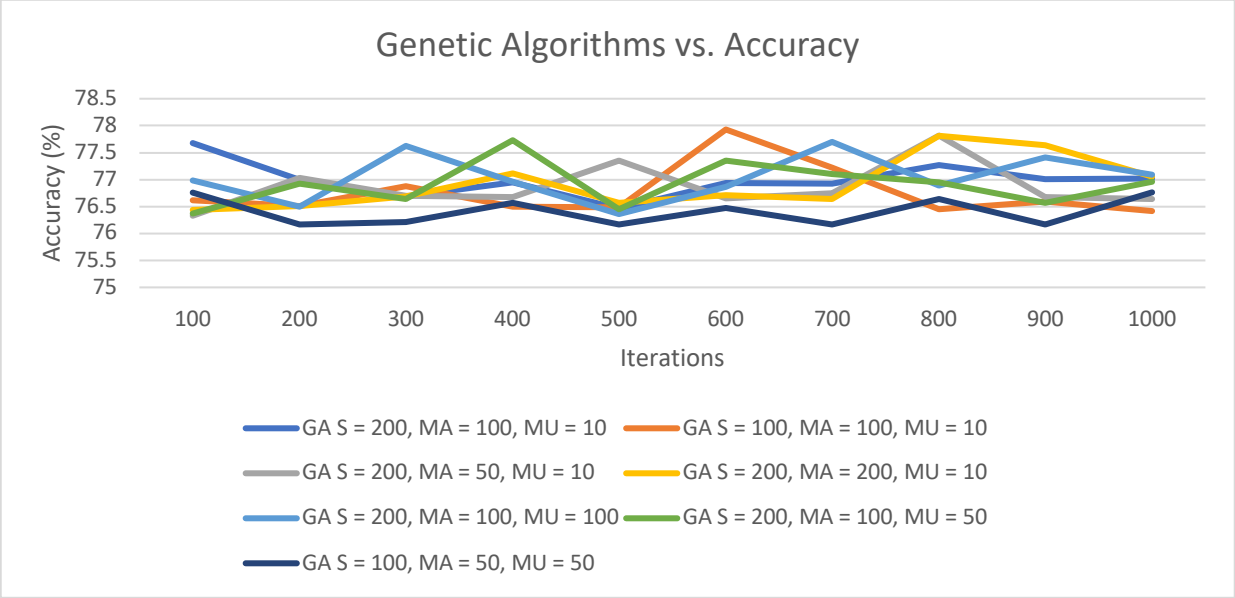


Genetic Algorithms

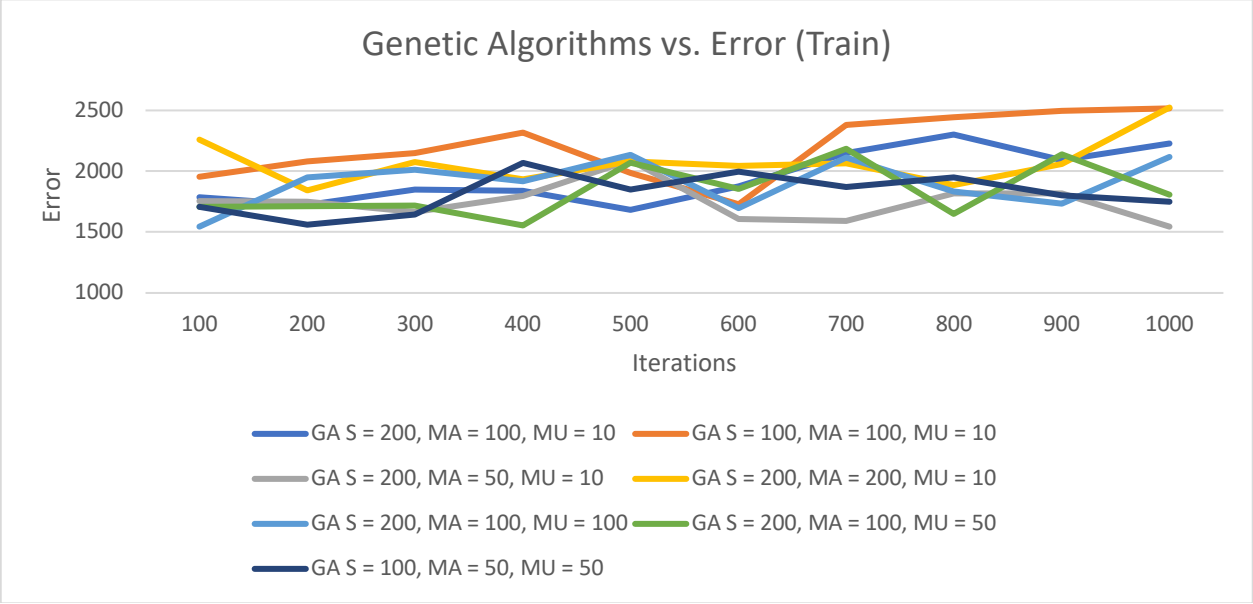
The genetic algorithm is modeled after evolution, which is known to successfully adapt species to their environments. A set of instances is known as a population, and each generation or iteration is formed by either mutation or crossing over the /hypotheses/instances. Mutation is similar to local search, where the hypothesis moves a small step in a certain direction. Cross-over is like mating, where the best hypotheses are combined in hopes of finding an even better hypothesis. Cross-over is especially useful for complex hypotheses with subspaces that can be optimized and reused in the future generation. These new hypotheses then replace the less fit population and repeats until the results converge.

The hyperparameters for this algorithm includes the starting population (S), the number of new instances formed from crossover mating (MA), and the number of new instances formed from mutation (MU). With the same reasoning from the simulated annealing section, I randomly chose

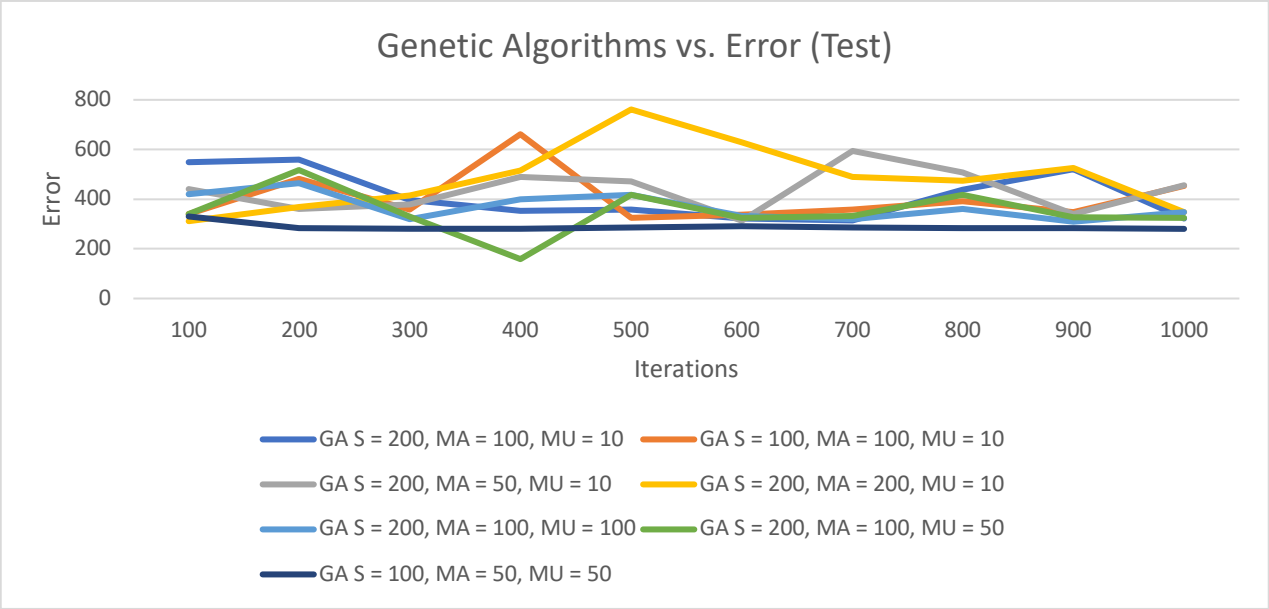
configurations of starting population, mate, and mutation (restricted to the fact that mate and mutation must be smaller than the population).



Unlike the previous two, genetic algorithms were not prone to the truss shape, and therefore was not vulnerable to a local optimum for this problem. Although the values look quite random, the y-axis actually covers a small range—the accuracies vary between 76% to 78%. Similar to the accuracy graph, the mean squared error graph below also did not vary as much. The best configuration based both on the accuracy and error graphs is the green line with a starting population of 200, a mate of 100 and mutate of 50. However, in the test graph the configuration of starting population of 100, and mutate and mate of 50 had the lowest error. This configuration was also low in the train graph, but the accuracy for classification was worse. Thus, the configuration of S = 200, MA = 100, and MU = 50 is still the best. Having a greater starting population allows for more variety and better crossovers possibilities; there should still be a small mutate rate to maintain the variability in the hypothesis space, but not so much that it becomes random. The mate rate should also not be too large as to lose all original hypothesis, but to be large enough to replace some of the worse instances with better combined ones. Once again, the error decreased in the test set, so the neural net weights were not prone to overfitting on the training data.



Randomized Optimization
Tiffany Xu

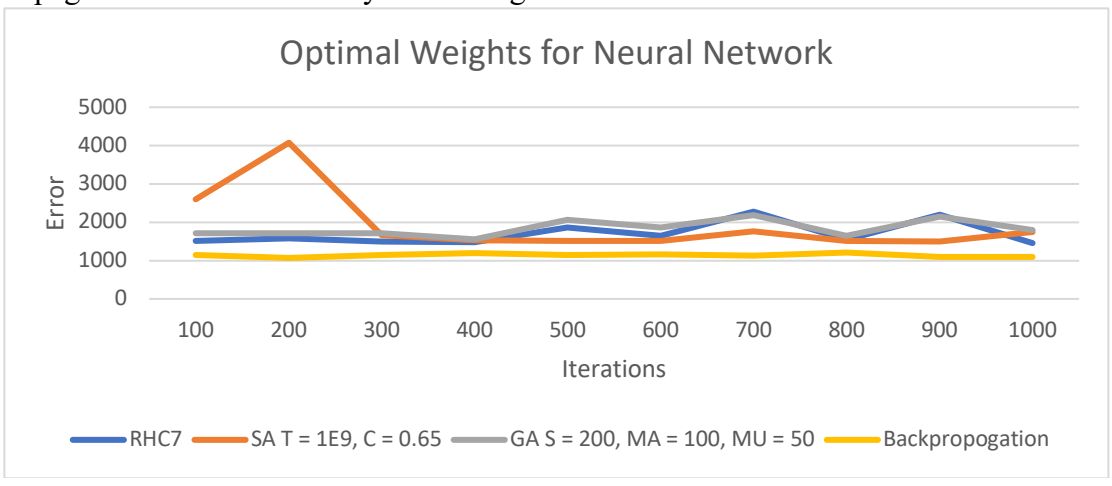


Summary

Of the three randomized optimization algorithms for the income dataset, the genetic algorithm was the best for classifying the data. It was not prone to the flip-flop in accuracies and remained consistent in a narrow range of accuracies. However, the genetic algorithms took much longer to run, with simulated annealing being the next slowest, and randomized hill climbing being the fastest.

Interestingly enough, when comparing the mean squared error for the optimal configurations of each, backpropagation beats all the randomized optimization weights, but simulated annealing does best among the randomized optimization problems. I would've thought that genetic algorithms would be the best, but its overall error is actually one the highest. This is why it is important to look at both error and accuracy over multiple configurations and iterations. By just examining the graph below, it would be easy to say that simulated annealing, after 300 iterations, is the best randomized optimization algorithm for replacing backpropagation, but based on accuracy, this is not the case.

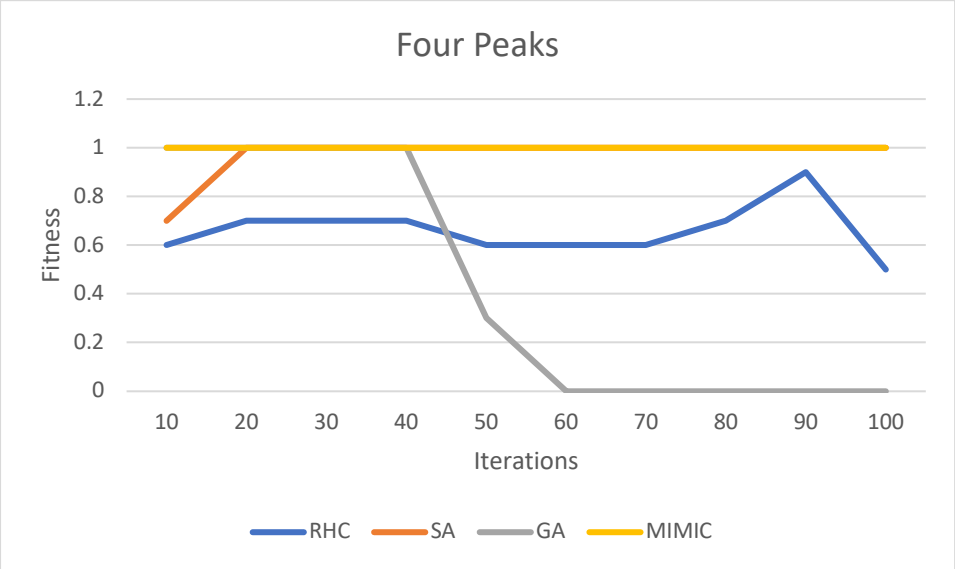
Overall, backpropagation still does better than using randomized optimization to determine weights. It remains stable and also has the lowest error. Perhaps there is a reason backpropagation is the default way to set weights on neural networks.



Randomized Optimization
Tiffany Xu

Part 2. Three Interesting Problems
Four Peaks (MIMIC)
MIMIC takes advantage of probability distributions and is one of the few algorithms that keep track of structure. The only difference between the first iteration and the 500th iteration in random hill climbing, for example, is a different point that might be better than the 1st. The algorithm doesn't learn about the optimization space itself or use that information to be more effective. Because the four peaks problem is highly dependent on the structure of the graph, MIMIC should do the best on this problem.

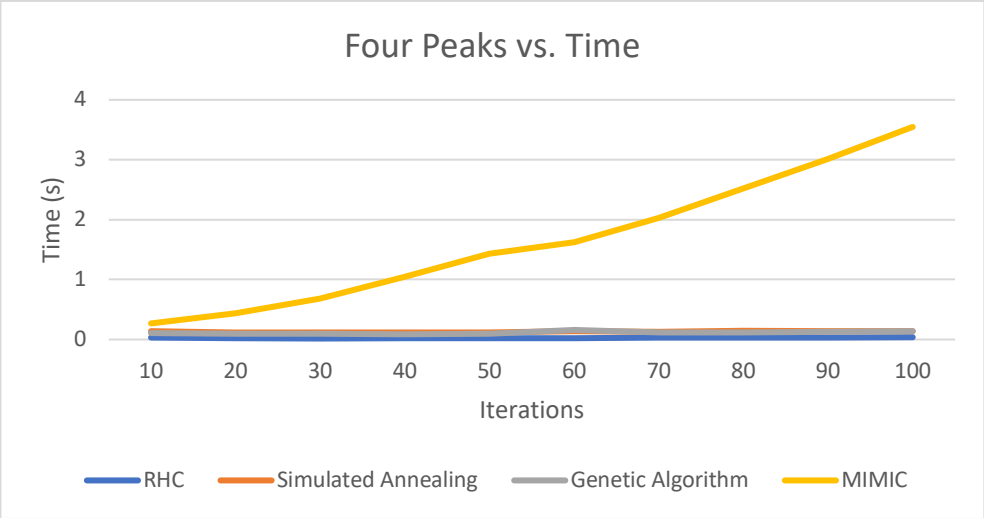
Four peaks is a bitstring problem with two global maximum peaks, and two local maxima peaks. Given a bitstring of length N and a position T, the problem aims to maximize contiguous '0' or '1's up to position T, with the opposite bit tailing until the end of the bitstring.



MIMIC did indeed perform much better than the other three algorithms, and immediately converged to the best solution. Simulated annealing and genetic algorithm reached the maximum fitness between 20 and 40 iterations, but did not stay there. Perhaps the temperature was set too high, and the cooling rate wasn't fast enough so that the instance went to a worse location. Randomized hill climbing, although never reaching the maximum, was more stable than simulated annealing and genetic algorithm. Because there are two local maxima, it is understandable that randomized hill climbing could be prone to falling into those basins of attraction.

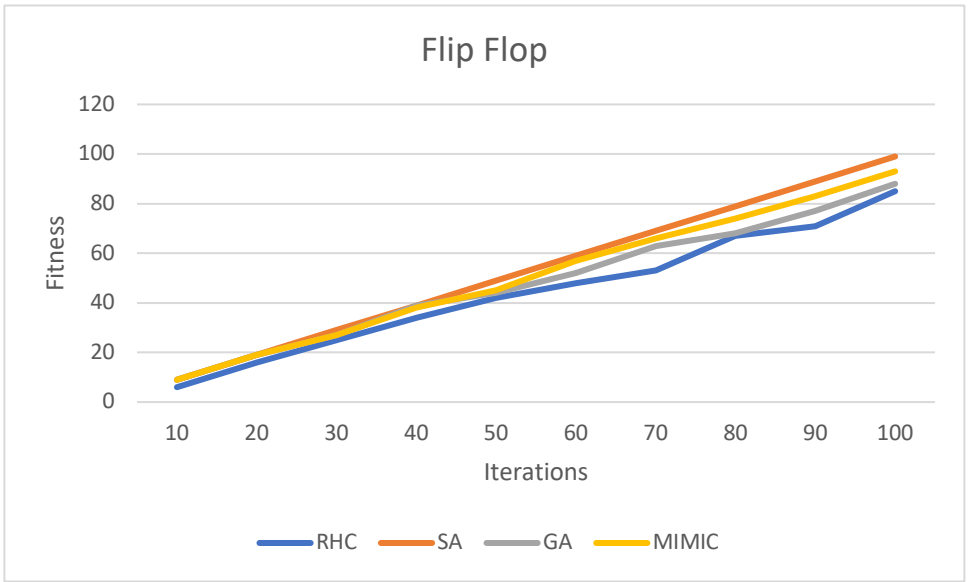
MIMIC worked best because it modeled the probability distribution for good points, and continued to sample even better locations. In our case, the algorithm found the best positions within the first 10 iterations. Although MIMIC found the best solution in just 10 iterations, each iteration takes longer than the other algorithms. The graph below shows that MIMIC is substantially more time consuming than the other three algorithms, especially as the iterations increase. However, for problems like this, where structure is important, the increased iteration time is worth it given the considerably less amount of iterations needed to converge to an optimal solution.

Randomized Optimization
Tiffany Xu

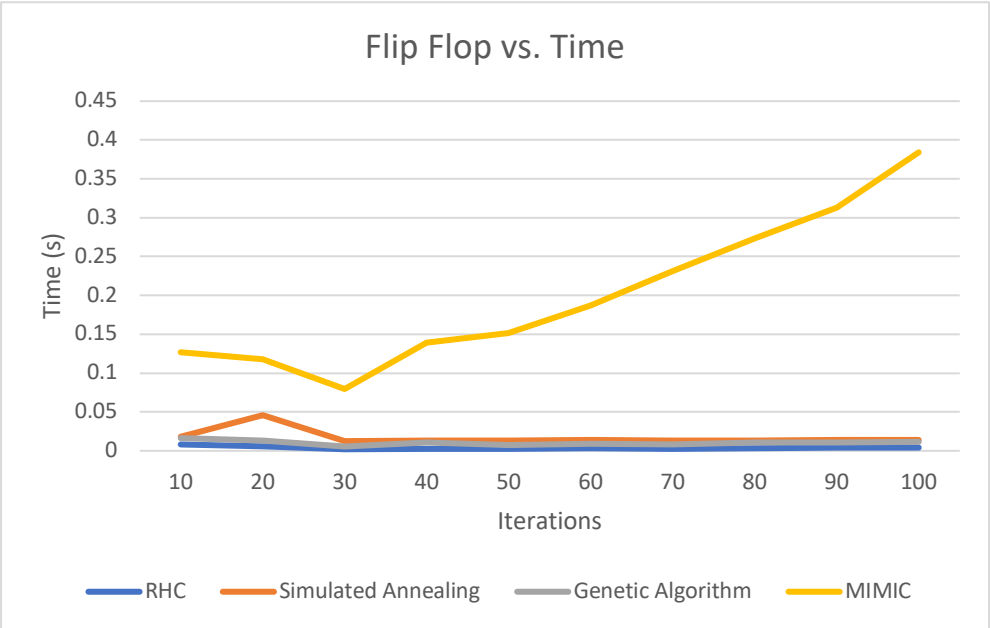


Flip Flop (Simulated Annealing)
Flip flop is also a bitstring problem that aims to maximize the number of switches or flips between 0 and 1. The first bit is always considered a flip, so "0111" would return 2. This problem has many local optima, and simulated annealing should be the best because of its ability to explore the many valleys of optima before narrowing down on (hopefully) the best valley. Whereas randomized hill climbing, if landing at a local maximum basin of attraction, will have no chance of escaping (unless with a random restart), simulated annealing can escape this local maximum with a high temperature.

Consistent with the hypothesis, simulated annealing did indeed do best on this problem, although all algorithms did pretty well, especially as the iterations increased. MIMIC did the second best, but took much longer to run, so simulated annealing was the best algorithm for the flip flop problem.

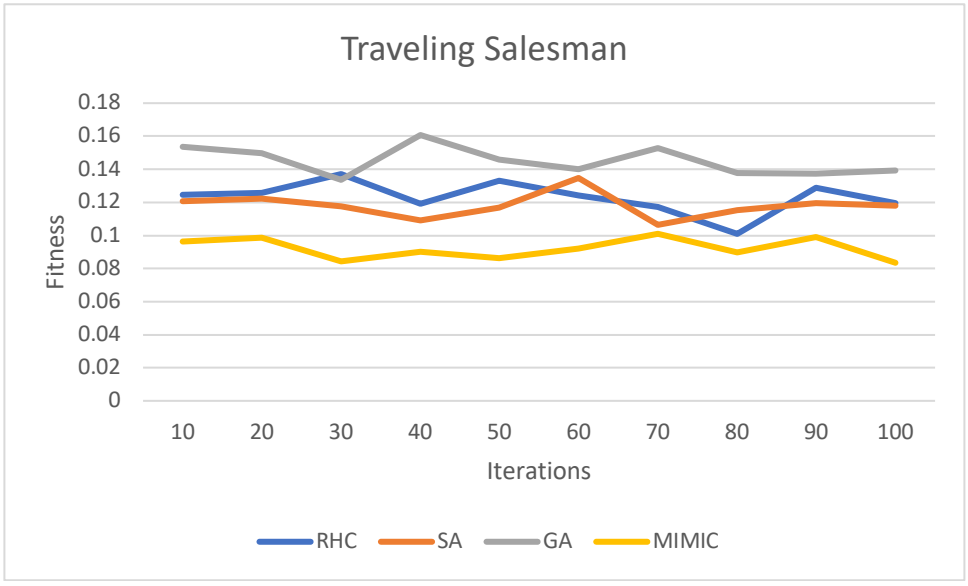


Here we can see that MIMC takes longer to run, especially as the iterations increase. Of the remaining three algorithms, simulated annealing takes longer, but by only fractions of a second.



Traveling Salesman (Genetic Algorithms)
Given a list of locations and the distances between them, the travelling salesman question asks what the shortest route is to explore every location, and then return back to the starting location. Genetic algorithms should do the best because of its ability to grab optimal subspaces and combining to find an overall even better fitness.

Genetic algorithms did indeed do best on this problem. Genetic algorithms can be thought of as having parallel multiple restarts, and so it explores the problem space and potential answers and moves onto the next generation with the more fit individuals. The other algorithms only look at one solution at a time, which would take an enormous amount of time to explore potential solutions for this problem (especially as the graph grows bigger).



Randomized Optimization
Tiffany Xu

Out of all the optimization problems, MIMIC look the longest on this problem because the hypothesis space was very big, and MIMIC must produce a probability distribution for the likeliness of the solution being in some subspace. Not only can we conclude that genetic algorithms is best for these types of problems, we can also say that MIMIC is lease suited for problems with a large problem space.