Matt Busch
CS 4641

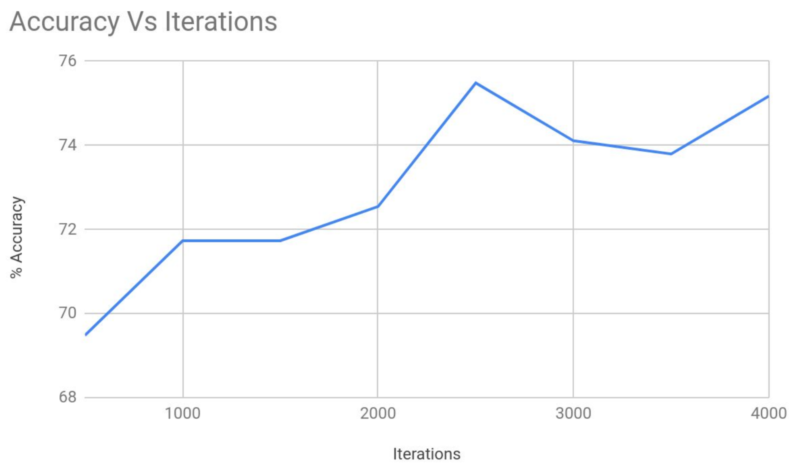
Project 2: Random Optimization

Part 1: Neural Network

For this analysis, I will be demonstrating 3 different randomized optimization algorithms by optimizing a neural network. I will be replacing the backpropagation of the neural network with randomized hill climbing, simulated annealing and genetic algorithm. These algorithms should find the best weights for the neural network on a dataset of wine rankings. This dataset comes from my first project, and is the wine goodness dataset where there are many different features and two classification, good, and not good. From my previous paper, we saw the neural network with backpropagation performed with an accuracy of about 73%, so we can compare these new algorithms to that 73% accuracy. I thought this dataset was the best to compare the algorithms to, because there is room to improve, but the neural network can do a decent job of classifying the data. Throughout my paper when I use the term accuracy, I am talking about the percent of the testing set that the neural network can successfully classify. I used ABAGAIL as my tool for running and analyzing these algorithm, and I ran a neural network with 10 hidden layers for each of the tests.

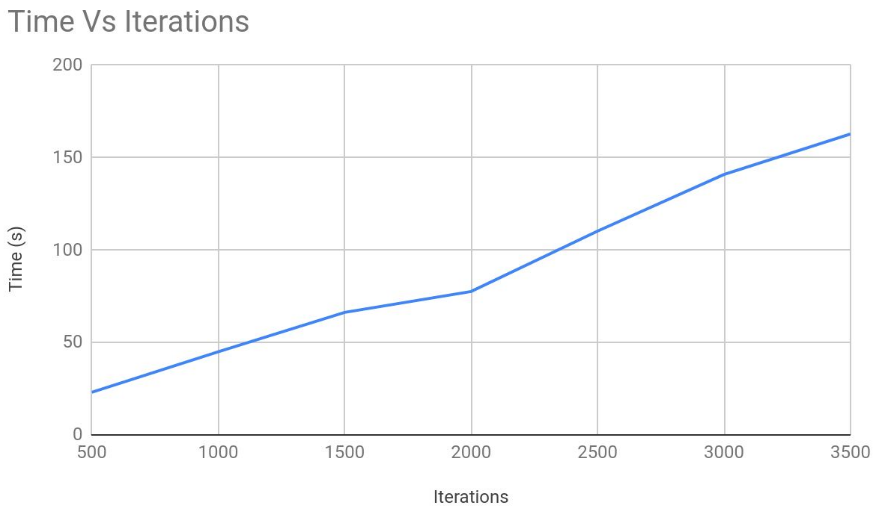
Randomized Hill Climbing

The first algorithm I ran was Randomized Hill Climbing. Randomized Hill Climbing is an algorithm that randomly finds a "neighbor" or a set of values close to currently selected values, and moves to the neighbor if the cost of the neighbor has a higher cost value than the current cost value. When the current cost is greater than all of its neighbors, it is the peak, and selected as the best set of values. The only hyperparameter for this algorithm I can test is number of iterations, so keeping the hidden layers at 10, I adjusted the iterations to create a form of learning curve, and recorded the training time.



Randomized Hill Climbing had an accuracy between 69% and 76%. The number of iterations determine how many times the algorithm gets restarted, so generally the more iterations the higher the accuracy. Eventually, the algorithm will find the true maximum, and would converge to the best accuracy, which may be around 75.5% - 76%. At the lowest number of iterations of 500 iterations, the algorithm does the worst with only about 69% accuracy. Also we can see that from 2500 up, there is a little bit of deviation, but maintains a very high accuracy.

Comparing the training time as I increase the iterations show a mostly linear result. As the number of iterations increase, the time it takes increases, as there are more restarts to try and find the optimal solution. The linear relation makes sense when you consider that each iteration should take about O(N) to complete, and adding iterations does not increase the complexity.



The testing time is negligible as it was usually much less than .1 sec to test.

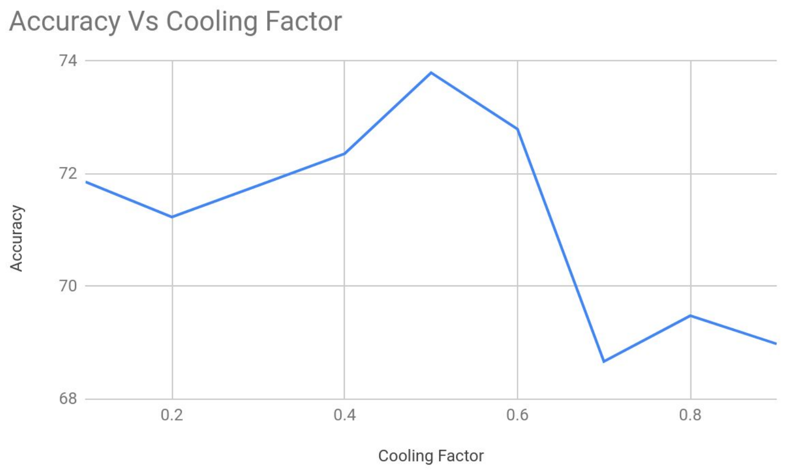
Randomized Hill Climbing Conclusion

The Randomized Hill Climbing algorithm performed well at determining weights for the Neural Network. When comparing to Backpropagation, Randomized Hill Climbing performed better, especially at the higher iterations. Backpropagation had about 73% accuracy, where the Randomized Hill Climbing was able to give a result of about 75.5%. At low iterations however, Randomized Hill Climbing performed poorly, and at high iterations, it took quite a bit of time. The iterations over 2500 are where the optimal values would be, and considering the time it took to run, 2500 iterations would be the best for this algorithm on this dataset. Eventually this algorithm could converge to the best solution, but with the random factor, that could be very quick or requiring an enormous amount of iterations.

To improve this algorithm, there are a few things that could happen. With less dimensionality, it would be easier to find the optimal solution. With high dimensionality, trying to randomly pick a neighbor to compare to would be harder. There could be many slightly better neighbors, but further down the line a different neighbor leads to the highest result. Another way to optimize is to have a weighted random algorithm, instead of just randomly picking a neighbor. The probability of picking each neighbor could be a normalized amount based on the increase in cost function of each direction. This allows for the randomness to remain, but the weighted aspect makes the algorithm likely to pick the neighbor that is the most in the correct direction.

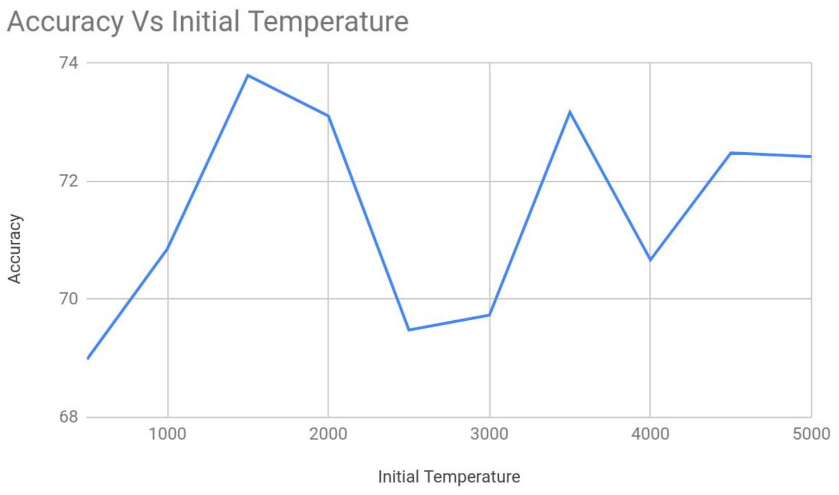
Simulated Annealing

Simulated Annealing is an algorithm very similar to Randomized Hill Climbing, but the big difference is that sometimes Simulated Annealing takes a “worse” choice, or a set of values with lower cost. It implements a “temperature” and a “cooling factor” to determine how often it will take the worse choice. The higher the temperature, the more random the algorithm will behave, over time the temperature decreases based on the cooling factor. To test how it would behave with my algorithm, I first selected the best cooling factor by setting the initial temperature to 1500, and iterations to 1500 and seeing where the best accuracy was. Using that, I set the accuracy to the best value and iterations to 1500 and iterated through initial temperature. Finally I got the best results from those two and built the accuracy vs iterations graph.

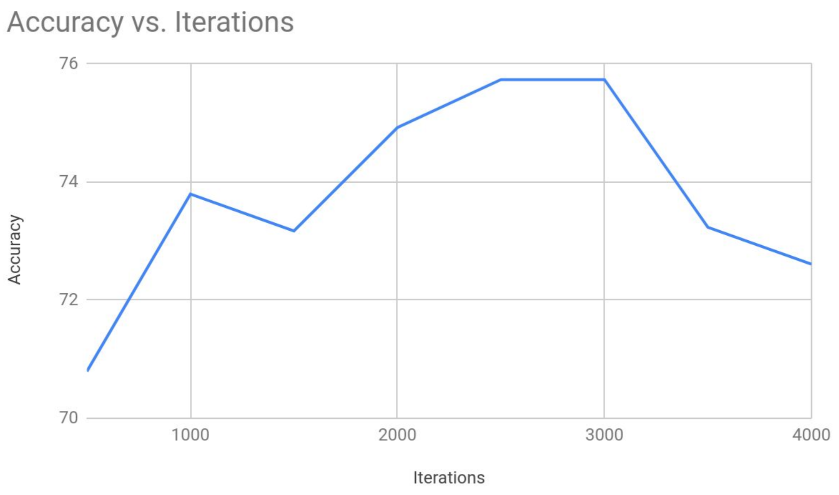


From the testing of the cooling factor hyperparameter, it appeared that .5 cooling factor gave the best accuracy. The closer to the edges, the worse the accuracy was, so a moderate cooling factor performed the best. From this information, I used a cooling factor of .5 to find out the best Initial Temperature Value.

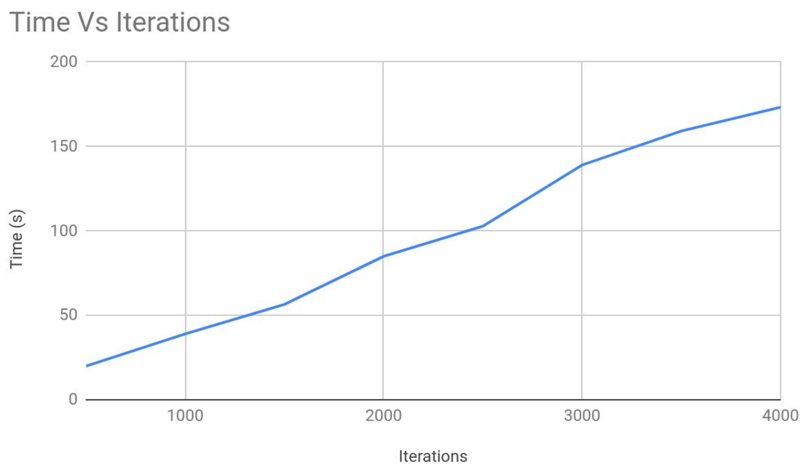
The accuracy on these runs from the value of the initial temperature fluctuate greatly. This indicates that Initial Temperature may not play as an important role in the accuracy, Depending on the runs, some may just be more lucky with the randomness than other runs. It is interesting that 1500 had the highest accuracy because that was what I used to discover the .5 cooling factor off of. This indicates they may be dependent on each other and if I picked a different initial temperature to select the best cooling factor, perhaps a different cooling factor may have been picked. The best results were with .5 cooling factor and a initial temperature of 1500, so that is what I used for the accuracy vs Iterations graph.



Simulated Annealing had an accuracy between 71% and 76%. The iterations vs accuracy curve actually decreased with too many iterations. The golden spot was between 2000 and 3000 iterations, and then it took a downward slope above 3000 iterations. The highest was around 75.5% similar to the high of the previous algorithm. At the lowest with only 500 iterations, the algorithm still has an accuracy of about 71% which is higher than the Randomized Hill Climbing lowest.



The time behaved similarly as it did with Randomized Hill Climbing. There was still a linear relation of time with respect to iterations. Each iteration of simulated annealing should average out to the same as any other, so the more iterations, the longer it will take. The time was almost the same for Simulated Annealing and Randomized Hill Climbing. Like with Randomized Hill Climbing, the testing time was negligible.



Simulated Annealing Conclusion

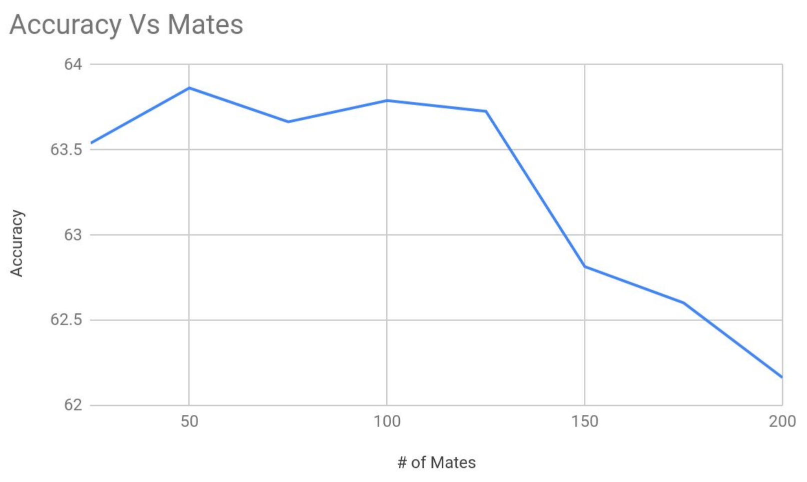
Simulated Annealing overall performed better than Randomized Hill Climbing. It was able to reach a maximum result in the same number of iterations, and taking the same amount of time. However it always was slightly higher accuracy than Randomized Hill Climbing for the same number of iterations. At higher iterations Randomized Hill Climbing had higher accuracy, but, because neither's accuracy continued to grow, the higher iterations are unnecessary. At its best, Simulated Annealing had an accuracy a hair under 76 % which is better than the backpropagation. At 2500 iterations, with 1500 initial temperature, and a cooling factor of .5, this algorithm behaved its best. I am surprised with the higher iterations the accuracy didn't continue to grow, but I imagine overfitting became an issue, and the algorithm was adjusting the weights of the neural network too heavily on the training set.

To improve this algorithm, it may be good to do a look ahead before taking the non optimal path. For example, if the initial temperature is high, and the algorithm chooses to go to a neighbor with a lower cost, it could save time going down a wrong path by looking ahead at the path and seeing if any of the neighbors are better than the chosen neighbor. If not, then it probably won't be advantageous taking that path, and could be a little more efficient.

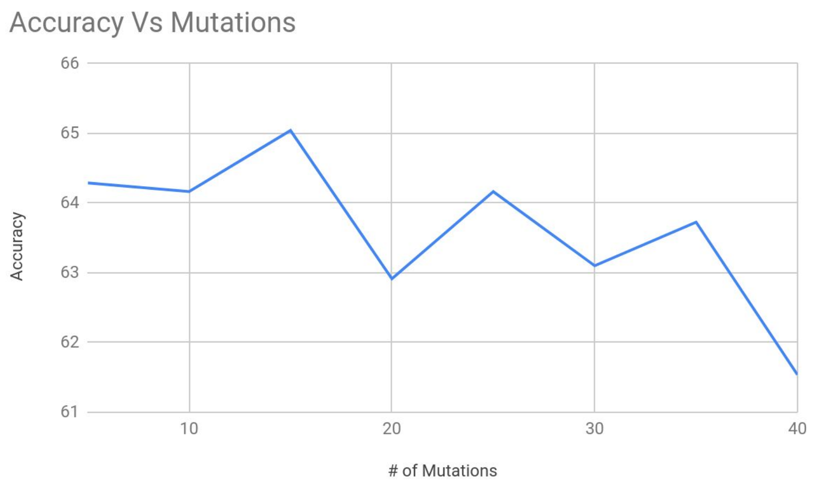
Genetic Algorithm

The Genetic Algorithm is an algorithm that uses the model of reproduction of organisms as a way to randomly optimize values. To use this algorithm, I first create a random pool of value sets that would represent the weights of the neural network. The values can be completely random, but more diverse sets in the pool is prefered. Then all the sets of values are compared and the best move forward, and the worst usually get eliminated. In order to continue to learn, a set of them will have random "mutations" or a change in one of the values in the set. Some are also paired up to create "mates" or a mixture of two sets. To implement this, once again I used 10 hidden layers in the neural network, and I experimented with 1500 iterations. I set the pool size to be 400, because too much larger took way too long to run, and smaller would not allow for as many mates or mutation. I started first looking for an optimal number of mates, and then used that to find the optimal number of mutations.

I first looked at number of mates by setting the mutations to 0. This would be a case where all the data was in the original pool, and looking at how mating can optimize that pool. As shown by the graph, when there were too many mates, the accuracy began to decrease, and the best accuracy was at 50 mates, or an eight of the total pool. I used the 50 to find the optimal number of mutations.

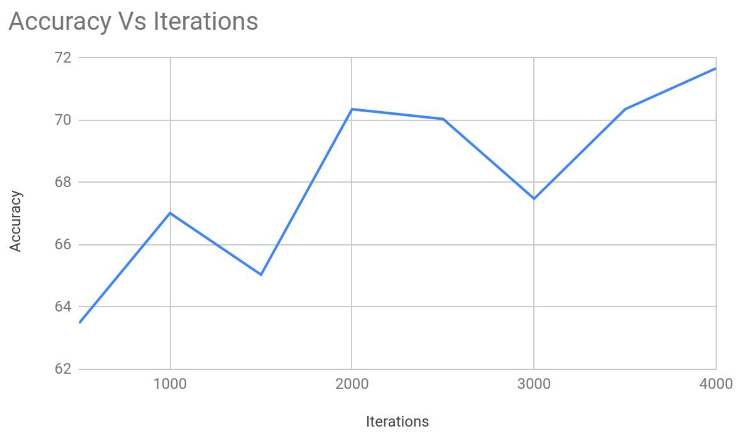


Like with the mates, I found that having too many mutations decrease the accuracy of the results. The best results came from low number, around 15 mutations, or a little less than a sixteenth of the total pool. Mutations can be helpful randomly, but too many mutations can ruin the good sets. The optimal values are 15

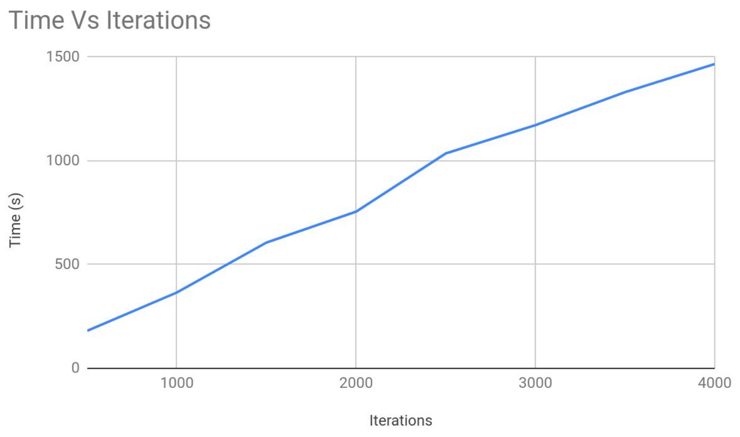


mutations, and 50 mates in a pool of 400 sets of values. I used this for the accuracy vs iteration curve.

The accuracy showed a general increase as the number of iterations increased. Overall from 500 to 4000 iterations, the accuracy increased about 9% but still only had accuracy under 72% for 4000 iterations. I assume that as the iterations keep increasing past 4000 the accuracy could continue to increase. Unfortunately As you will see below the algorithm was much slower than previous algorithms, so it was infeasible to increase the iterations much higher. The algorithm had the best accuracy at 4000 iterations, but all of these were worse than the other algorithms, and the baseline of backpropagation.



The time still has a strong linear relation, but several times longer than the previous algorithms. What took at most 200 seconds before is now taking about 1500 seconds, or about 25 minutes. Unfortunately that means if we want to keep adding iterations until we find a comparable accuracy, it could take over an hour to run the algorithm. When adding more mating the algorithm took longer as well, which is why I kept it on the lower end with only 50 mates.



Genetic Algorithm Conclusion

For under 4000 iterations, Genetic Algorithm did not do a great job creating weights for the neural network. After all the iterations, the best accuracy it could produce was under 72% which is less than the other algorithms and backpropagation. As the iterations increase, it is clear to see that the accuracy tends to increase as well indicating that eventually it could be used to classify the dataset but as far as being a feasible and useful algorithm, the time complexity of it indicates there are probably other better algorithms to use. Another issue that may have affected the accuracy is the size of the pool, number of mates, and number of mutations could all be dependant on each other, so my choice of pool size may have prevented better accuracy to ever arise. I picked 400 due to time management, and it performed as well as higher pool size in initial tests, but if I truly wanted to find the best 3 of the hyperparameters, I would need to make a 3D graph of all combinations, and select the best result.

Another tricky part of this algorithm is that you realistically want the best sets to move on to the next round, and the worst sets to be eliminated, but in reality, that is a greedy approach, and could eliminate the possibility of finding a great solution through mating with a low scoring set. To improve this, if you could feed in some information about the weights in the sets, and if there are any that are dependent on each other, it would be easier for the algorithm to effectively mate the good and the bad sets, instead of the random approach used now.

Part 1 Conclusion

Using the three algorithms, it is interesting to see how each performs differently. Randomized Hill Climbing seems to perform best with a high number of iterations, and never had a significant decrease in accuracy with additional iterations. Simulated Annealing performed very similarly to Randomized Hill Climbing and would usually produces slightly better weights for the neural net and producing slightly better accuracy, but eventually with too many iterations, would lose accuracy, most likely due to overfitting. Finally Genetic algorithm was a slow, and the least accurate of the group under 4000 iterations. Perhaps with many more iterations, it would continue to increase the accuracy until it is on the same level as Randomized Hill Climbing and Simulated Annealing.

If I were to make a decision for the best algorithm for this dataset, I would have to say Simulated Annealing modeled the data the best and produced weights for the neural net that had the best accuracy for the testing set of data. It was quick, and was even more accurate than the default of backpropagation. Given a long enough time, Genetic Algorithm could potentially achieve a better accuracy, but an important part of an algorithm is to do it efficiently and Simulated Annealing is very time efficient.

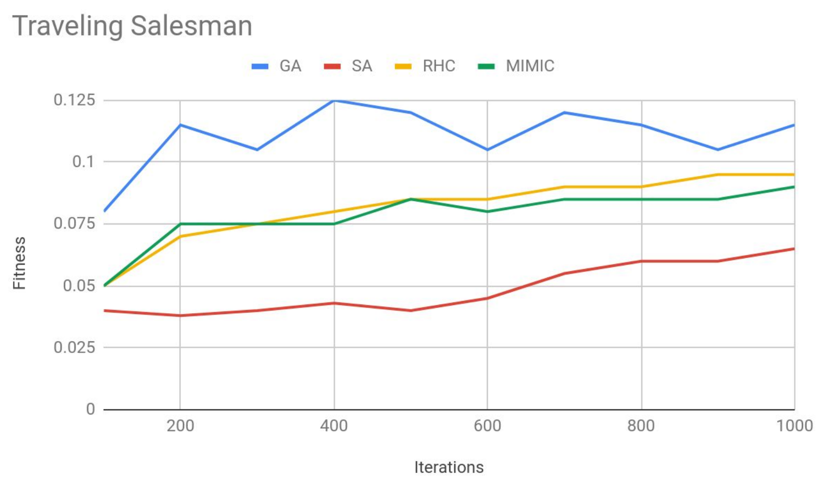
Part 2: Interesting Problems

In addition to optimizing neural networks weights, random optimization can be used for many problems, especially problems that are too complex to solve easily, like np-hard problems. In this next section, I will look at three different problems and use the three algorithms from before, as well as the MIMIC algorithm to solve the problems. We will look at the traveling salesman problem, knapsack problem, and finally the flip flop problem.

Traveling Salesman

The first problem I will use the algorithms on is the traveling salesman problem. This is a well known problem that involves graph search and finding the most efficient path. It stems from the issue of a salesman is trying to get from city to city or house to house in the most efficient manor, and return to the start. For my tests, I used a size of 50, which has an extremely large amount of options, so is impossible to test them all, which brings the need for an effective algorithm.

I plotted each of the four algorithms comparing the fitness of the function to the iterations. The higher the fitness, the better the results from the algorithm, and it was clear to see that the genetic algorithm performed the best of the four algorithms. At low iterations, it was able to quickly learn and have high fitness, and maintained an optimal level through all the iterations. With this problem, exploring is an important aspect, and the genetic algorithm is great at creating new ways of exploring.

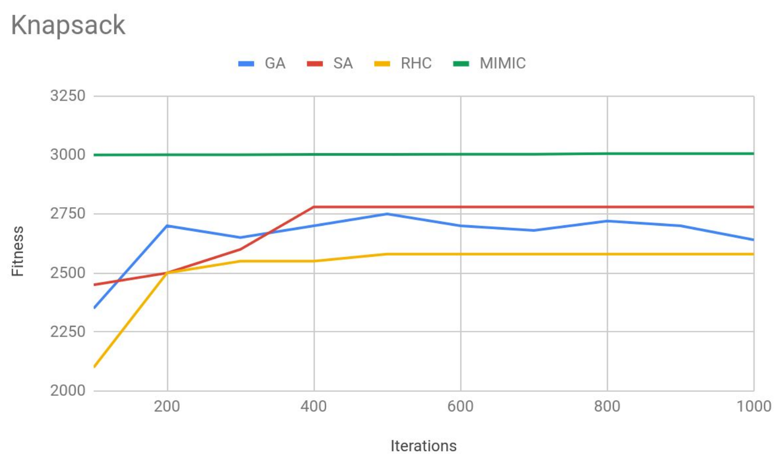


Comparing times, the MIMIC took the longest where at the higher iterations took around a minute to run, but all the other algorithms were pretty consistent, the genetic algorithm taking a second where the others were under a second.

KnapSack

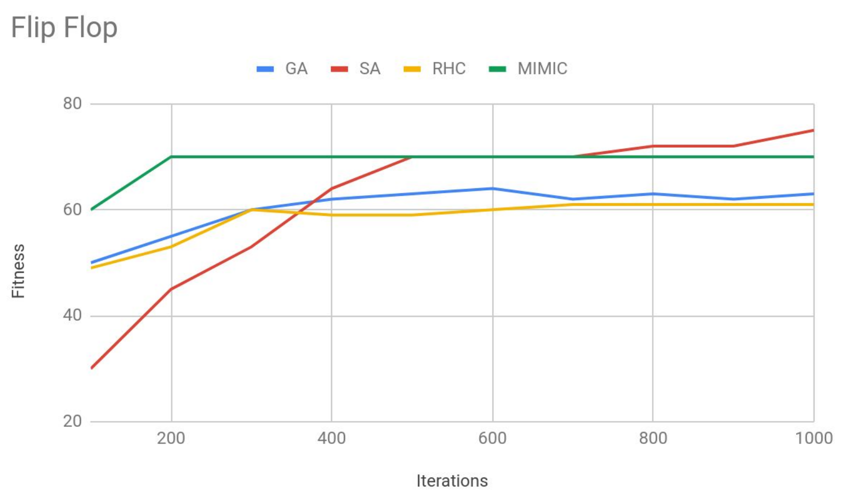
The knapsack algorithm is an algorithm where there is a bag (or knapsack in some areas) that can hold a certain amount of items, and there are a bunch of items with weight and values. The goal of the algorithm is to maximize the value of the bag but staying under weight. To test this we had a bag and 40 different items of different weight and values.

Looking at the plot for the fitness of each of the algorithms, MIMIC was able to reach optimal fitness quickly and kept consistent the entire time. This indicates that MIMIC was able to find the optimal values with only a few iterations. Based on the ability to get information from each iteration, it was able to quickly optimize the items. On the other hand, RHC/SA have a greedy problem, which given the nature of this problem is not an optimal way to solve it. For example, always grabbing the most valuable item may make you run out of space where two smaller but better fitting items would optimize the cost. MIMIC still took a lot longer to run than any of the other algorithms, but due to its ability to optimize after only 500 iterations, it reached its optimal result in only a few seconds.



Flip Flop

The flip flop problem is a circuit based problem where bits are on or off, and to turn one on, another must be turned off to reach a result. We can use our algorithms to be able to reach that, so I generate a random bit string of size 80 and a result.



For this algorithm, Simulated Annealing became the best algorithm with the highest fitness. Because of the nature of only flipping one at a time, it makes sense for an algorithm such as Randomized Hill Climbing, and Simulated Annealing to perform well. Simulated Annealing however, always seems to be better than Randomized Hill Climbing at higher iterations, due to its ability to explore better. MIMIC also performed very well in the beginning, having the highest fitness for most of the time. The hypothesis space is small enough however for Simulated annealing to randomly find the best result, and with more iterations, it would probably continue to grow until it converges at the optimal result, where MIMIC would probably be stuck at its current position. Once again, MIMIC was very slow at the higher Iterations, but was able to converge early after only about 200 iterations, when it was still in the seconds.

Part 2 Conclusion

There are many problems that can be solved using Randomized Optimization, and each one has a specific type of algorithm that can perform better for optimizing it. When faced with a new problem, it is important to understand the concept of the problem, and try to match it with an algorithm that excels in that area.