Machine Learning CS 4641 Project 2 Analysis

Eric Wang – 3/3/19

Introduction

Randomized Optimization aims to solve optimization problems through iterative randomized selection of inputs. The benefit to this as opposed to using the gradient is that RO does not require the function of the problem to be continuous or differentiable, and therefore is much more robust on multiple problems. This report aims to compare three RO algorithms to a traditional gradient descent algorithm in the context of a Neural Network, then to compare four RO algorithms on discrete problems that would not be able to be solved using gradient.

Neural Net Weight Search

The first report explored supervised learning algorithms to solve classification problems. Specifically, hyperparameters for five algorithms including a neural net were tuned to model classification problems from two datasets, one using converted categorical to binary features and another using continuous features. The dataset used for comparison is the latter, the Iris dataset from the scikit-learn Python package.

The Iris dataset consists of four continuous input features, detailing the petal width, petal length, sepal width, and sepal length. The fact that these input features are continuous matter a lot to the usage of a neural net since the weights in a neural net are continuous, and therefore an increase in the value has meaning beyond just a categorization/classification. Moreover, the values of the input features consist of a small range between 0 and 7, rounded to the nearest decimal point. All the values are the same order of magnitude, which works without normalization. For these reasons, the Iris dataset was chosen over the other dataset to highlight here.

The output label originally consists of 3 possible classes, representing 3 different species of Iris plants. However, for the purposes of this paper, we are training a neural net for binary classification. Therefore, samples from only the first two classes were taken, resulting in 100 samples with an even split between them. Using sklearn, a neural net with the hyperparameters was trained on 80 samples to test for differences between this new binary classification problem and the previous multilabel classification problem. Using GridSearchCV, new hyperparameters were found for this problem, which happen to only differ from the old problem by 1 parameter. The activation function used was identity, as opposed to before which was tanh. Otherwise, the net still has a structure of one hidden layer with five nodes, and still trained over a constant learning rate. *Figure 1* below shows the generated learning curve.

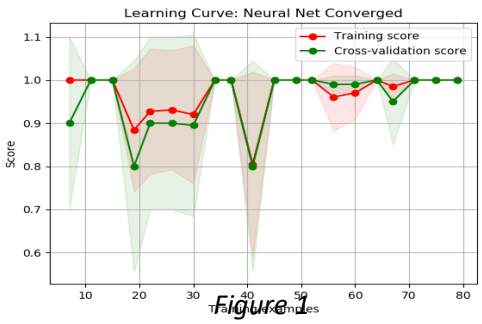

Figure 1

As shown in *Figure 1,* the neural network, trained over backpropagation and gradient descent, converges to both a training and testing score of 100%. This is not particularly surprising when compared to the previous model explored in the first report, which gave a training accuracy of 98.33% accuracy of 96.67%. As previously discussed, the Iris data lends itself well to neural networks, both due to the layout of the data and due to the nature of the problem. The different classes of irises have different enough measurements overall for a neural net to be able to accurately predict samples. This is highlighted in our model here, with 100% of the 20 testing samples accurately classified.

Randomized Optimization as Alternatives to Gradient Descent

The three optimization algorithms we are using to train the neural network are Random Hill Climbing, Simulated Annealing, and Genetic Algorithms, implemented in the JAVA package ABAGAIL. Normally, a neural net would use gradient descent through backpropagation to update the weights based on the error and train the model. We are

going to explore the usage of optimization algorithms on the weights instead, with the cost function for the weight combinations being the sum of squared error (hereafter referred to as SSE).

An accuracy of 100% is a high metric to pass for our randomized optimization alternatives. However, upon simple trial runs of each given arbitrarily large max iterations, we find that in fact, all three algorithms perform well on the iris classification problem, resulting in 100% accuracy after a varied number of iterations and runtimes, as shown in *Figure 2* below. The iris problem may be a little too simple to compare just accuracy as a performance metric. Therefore, rather than compare based on accuracy, we are going to compare based on number of iterations for convergence as well as runtime. This will allow us to see how computationally expensive these algorithms are compared to one another, especially since they all give us the global optimum at the end that leads to 100% accuracy.

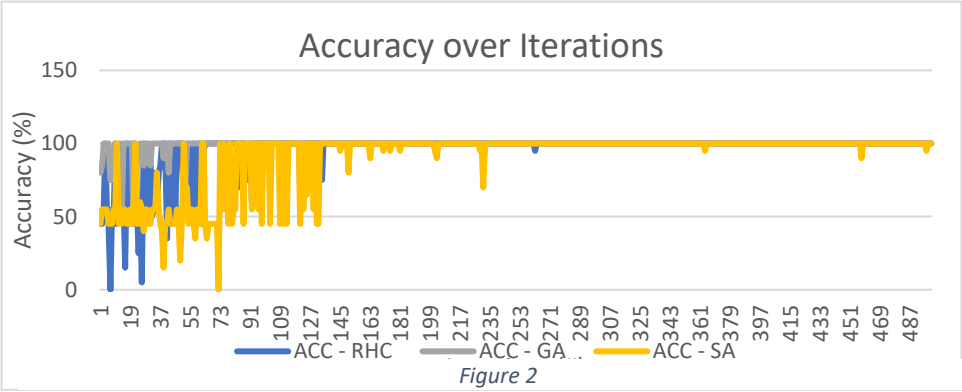

Figure 2

Before comparing the three algorithms, we need to have a benchmark to compare them to. In order to analyze the output of the backprop error calculations, we are going to implement the same neural net in ABAGAIL. *Figure 3* below shows the output after 1000 iterations. The sum of square errors drops to near 0 at 16 iterations and converges by 50 iterations. The runtime for the model comes out to be 4.7536s.

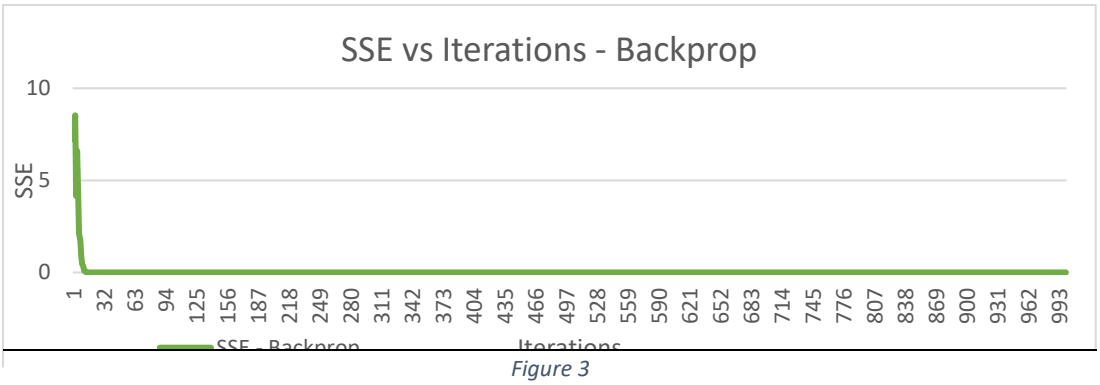

Figure 3

Randomized Hill Climbing

The first algorithm is the most basic. Randomized Hill Climbing defines a neighborhood of weights around the initial randomized input weight vector, chooses a sample from that neighborhood, calculates the cost for the sample chosen, and moves to the sample if it has lower error. This is run for 1000 iterations, outputting the sum of square errors per iteration as shown below in *Figure 4*.

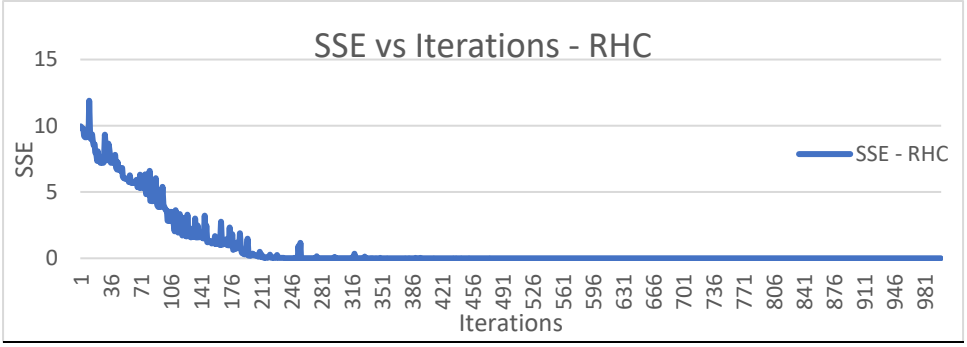

Figure 4

The error starts off higher than backprop at around 12, falling erratically as it finds its way down across a general downwards trend. The error falls below 0.1 for the first time at iteration 212, and continues to converge down to 0, save for a few spikes here and there, hitting 0.01 consistently by iteration 260. The last inconsistent point falls on iteration 396, and the error is consistently 0 from then on.

Clearly, Backprop outperforms RHC when it comes to iterations necessary for convergence. Backprop hit near 0 error at 16 iterations, whereas RHC took 260 iterations to do the same. This is because intuitively, backprop allows tracing of error, adjusting weights specific to where and how the error took place. Compared to RHC, which randomly chooses how to adjust weights based on luck in sampling from the neighborhood, it makes sense that backprop takes much less iterations than RHC. However, this does not take runtime into consideration. For 1000 iterations, RHC took 0.598s, which is 8 times faster than Backprop. This is because, despite taking more than 16 times the number of iterations, RHC only needs to calculate the error based on the inputs, effectively only needing to feedforward, whereas Backprop needs to calculate the error through feedforward, then propagate those errors back across all the weights on all the edges, adjusting them as it goes along. Therefore, Backprop iterations take much longer. With a larger dataset and more complex model, this difference would be highlighted even further, but even a small dataset on a simple model can highlight the tradeoff between iterations and runtime. With a simple cost function such as sum of squared errors for a neural net, the high number of iterations does not matter as much and leads to RHC having a better runtime than Backprop.

This implementation of RHC effectively has no hyperparameters. Restarts were not implemented, since our model was able to find the global optimum that led to 100% accuracy every time. In order to potentially improve iterations until convergence, General Hill Climbing could have been implemented instead. This would mean the algorithm calculates error for all neighbors and moves to the best neighbor every time. A GHC implementation could have lowered the number of iterations until convergence, potentially at the expense of runtime.

Simulated Annealing

The second algorithm effectively runs the same iterations as RHC but builds upon it by introducing two hyperparameters that allow the algorithm to occasionally choose an input combination that has a higher error. This is done in order to encourage "exploration" rather than just "exploitation". Rather than being greedy and just taking the best inputs at every point, SA encourages occasionally choosing another point, even if it performs worse, to search a space that could potentially be better than the space it started off in. This, in a sense, prevents the model from trusting the original random data point too much, which helps in finding a global optimum.

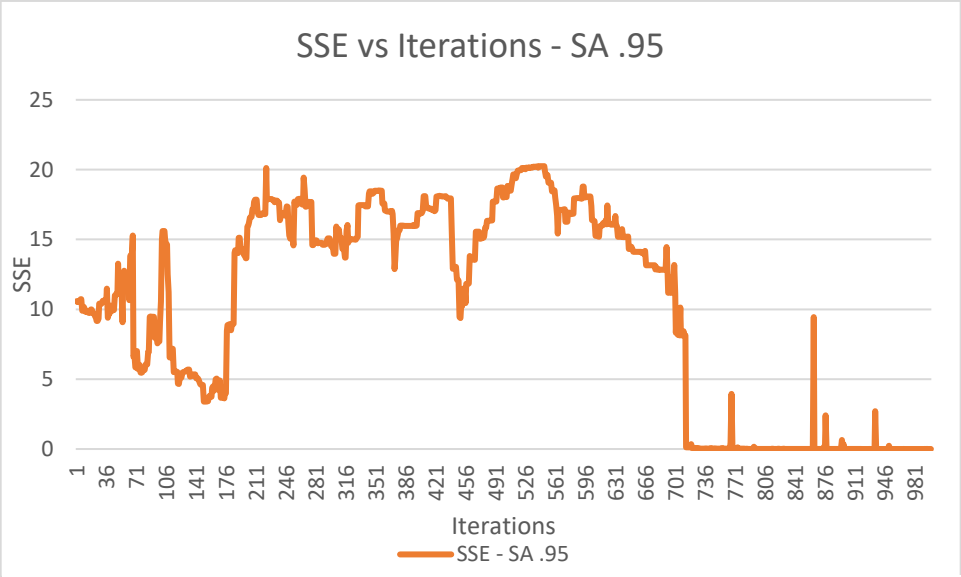

Figure 5

Figure 5 above shows the sum of square error output at each iteration for 1000 total iterations. This is an initial run, using a cooling rate of 0.95 and a starting temperature of 1E11. The temperature is an exponent that determines how likely the algorithm will choose an input that has a higher error, while the cooling rate is effectively a multiplier to change the temperature after each iteration. Clearly, with these default parameters, the algorithm does not perform very well relative to number of iterations for RHC and Backprop. The chances to choose a worse input can be seen in the changes in error across iterations, starting off near 10 but blowing up to above 20, even at over 500 iterations. It seems that for a problem as easy as this, the cooling rate is not quite fast enough to prevent the algorithm from exploring too much. The model does not converge until past 700 iterations.

This suggests that we will need to test for optimal hyperparameters to help us find the best model. The parameters tested are cooling rates between 0.95 and 0.55 at an interval of 0.1, and starting temperatures from 1E5 to 1E13, at an interval of 100.

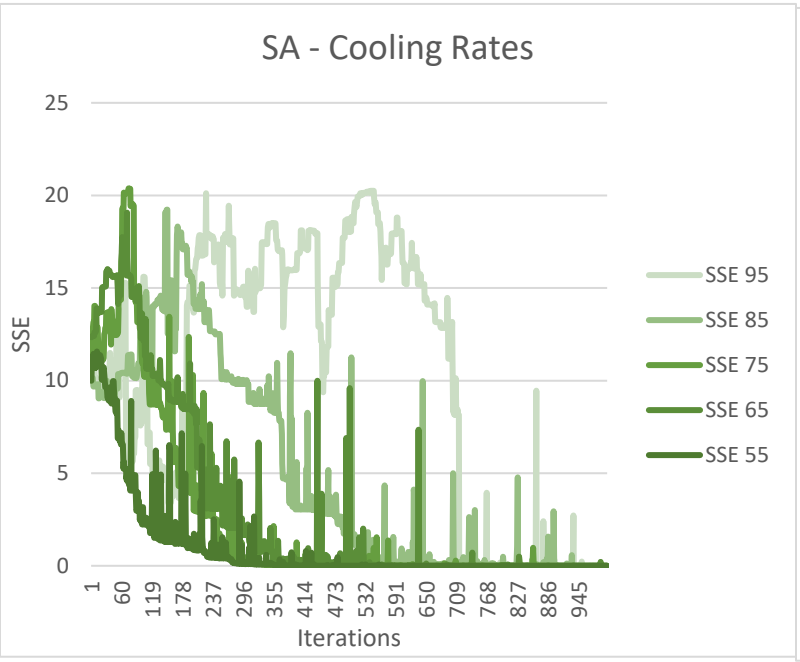

Figure 6

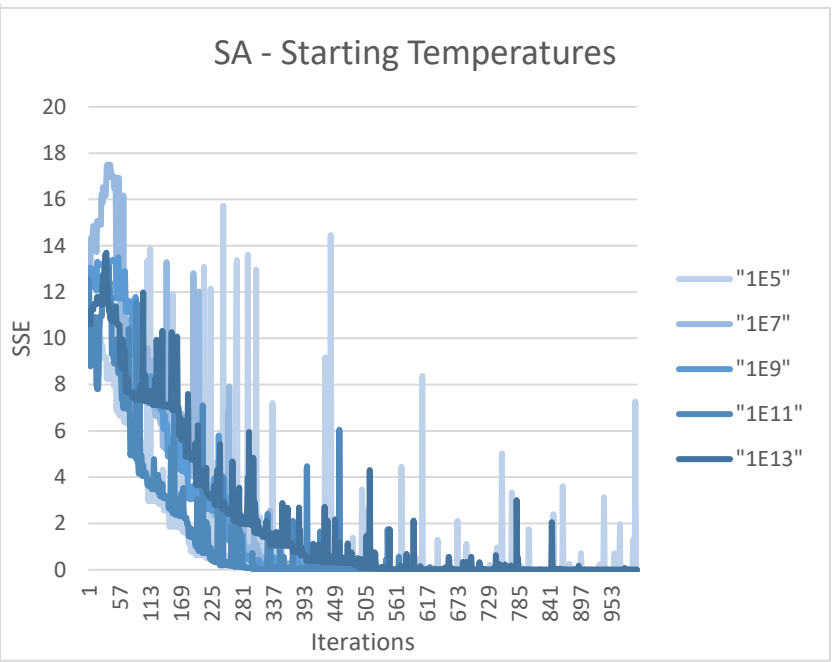

Figure 7

Figure 6 above shows the cost curves across 1000 iterations for simulated annealing run across the different cooling rates, with lighter green at higher (slower) cooling rates and darker green at lower (faster) cooling rates. The starting temperature was kept constant across the tests at 1E11. Compared to the initial run of cr=.95, the lower cooling rates perform much better. In terms of iterations, the lower the cooling rate, the faster the convergence. This intuitively makes sense since the lower cooling rates make the SA behave more similarly to an RHC, which in our case performs better in terms of iterations. The lowest, at cr=.55, has a general downwards trend, and converges around 300, which is much lower than the 700 for cr=.95 and closer to RHC at 260. Following, initial temperatures were tested at a fixed cr=.55, as shown above in *Figure 7*, where higher starting temperatures are darker blue and lower starting temperatures are lighter blue. The change of performance is not as drastic but still noticeable. At cr=.55, lower starting temperatures seem to be more erratic, while higher starting temperatures take longer to converge. On this run, the initial temperature of 1E11 seems to converge fastest at around 280. This parameter seems to be more arbitrary and works as long as we don't choose too high or too low of an initial temperature relative to our cooling rate. *Figure 8* below shows the error output for the best combination of parameters, cr=.55 and t=1E11. It's noticeably better than the initial run, starting to converge at around 280 iterations, and stabilizing past 480 iterations.

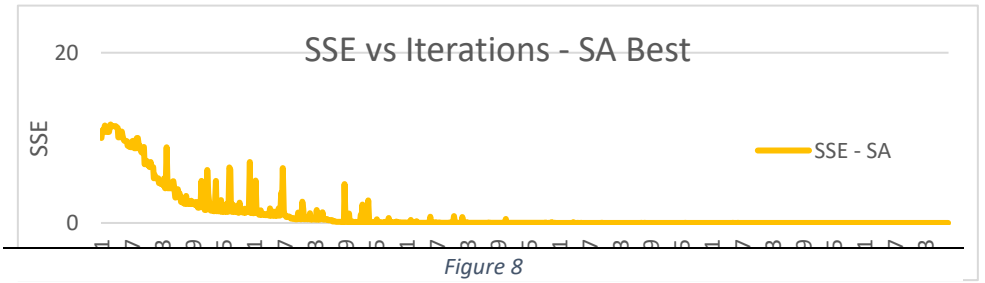

Figure 8

This model's performance, puts it slightly behind RHC, and thus also behind Backprop. This is because SA's strength is overcoming local optima through randomly choosing inputs in different spaces. However, our problem does not struggle with getting stuck in local optima, and easily finds the global optimum. Thus, SA's strength turns

into a weakness, as it spends more time jumping around, which ultimately don't help it do any better than RHC. However, SA does have a faster runtime at .281 seconds, roughly half of the runtime of RHC and thus 16 times faster than Backprop. This is definitely faster than Backprop, but at an absolute scale, this runtime isn't much faster than RHC. In fact, the same model could run slower than RHC in some cases. This is solely based on the randomized weights and how the model progresses.

One change could have been to redefine the neighborhood as a larger set of inputs across a wider hypothesis space. This could have encouraged more exploration. However, this is more relevant in a more complex problem with more local optima to get stuck in. With this problem, exploration is not as rewarded, and thus, SA performs poorly compared to RHC and Backprop.

Genetic Algorithms

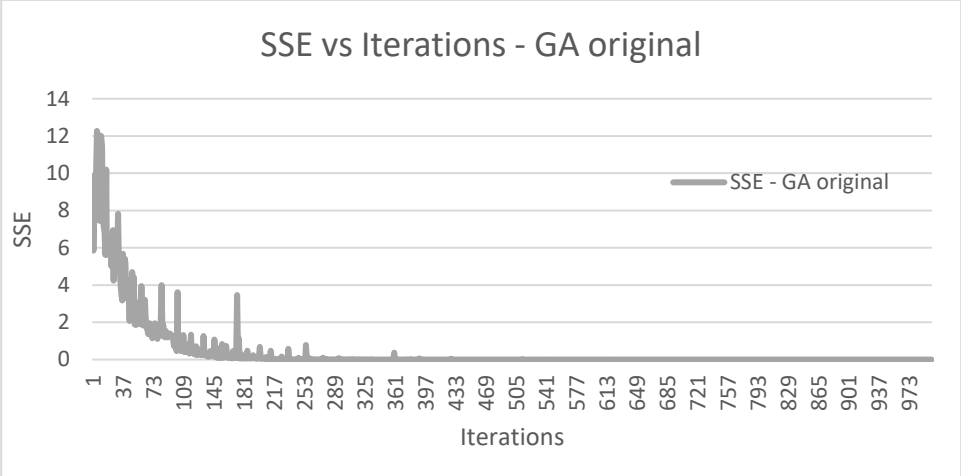

Figure 8.5

The third algorithm improves upon the simpler implementation of the first two by keeping track of a population of points as opposed to just a single sample per iteration. Each iteration then improves this overall population based on a specified number of crossovers and a specified number of mutations. The specific implementation used is for a crossover to be uniform across all bits in the two samples considered, and for a mutation to add one to the sample considered. The optimal value is then the best sample in the final population. *Figure 8.5* above shows the first run of this implementation of GA, with a population of 100 (the entire sample minus testing), 50 crossovers, and 10 mutations per iteration. This already produces a model with better performance than RHC and SA from before. The model starts with fairly high error but quickly approaches a near 0 error at around 140 iterations, before stabilizing and converging past 250 iterations. Before settling, the hyperparameters should be optimized to potentially find a better model.

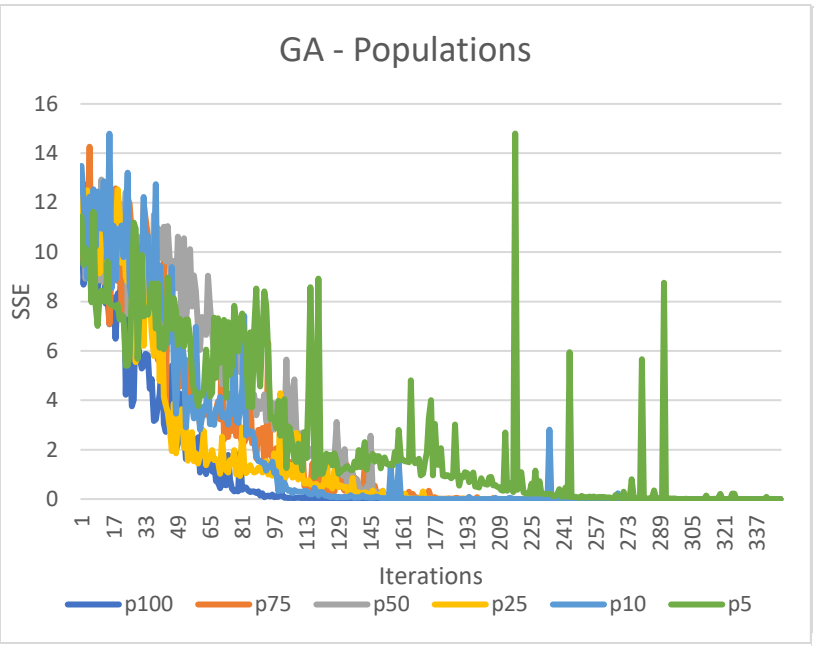

Figure 6

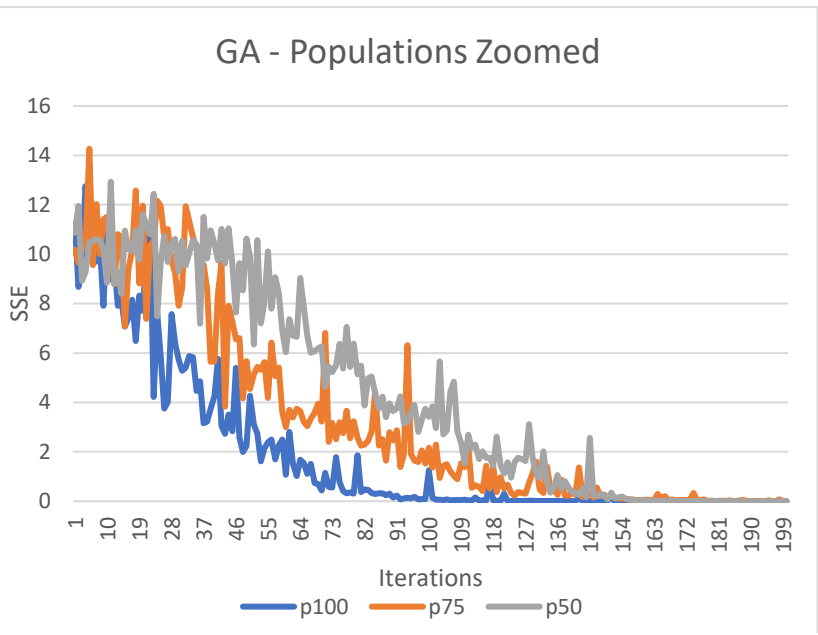

Figure 10

Figure 9 above shows the error output across multiple iterations for different populations. Crossover was kept at 50 and mutation was kept at 10, except for when they exceeded population and were adjusted down accordingly. As shown, low populations lead to erratic behavior, as a miscellaneous crossover or mutation that might be harmful could very easily lead an entire population astray. *Figure 10* shows a zoomed in version of just the highest three populations, showing clearly that p=100 has the best performance. This intuitively makes sense, since this means the algorithm has a higher chance of eliminating poorly performing samples and generating higher performing samples. Following, with p=100 as the best and co=50 as the default, Gas with different number of mutations were tested, as shown below in *Figure 11.* The pattern seems erratic, as m=50 and m=10, one half and one tenth of the population respectively, both perform fairly well, while the middle parameters do not. M=1 performs noticeably poorer and doesn't converge within 150 iterations. This may be attributed to variance in the initialization of the problem, but regardless, half of the population seems to do consistently well across repeated tests.

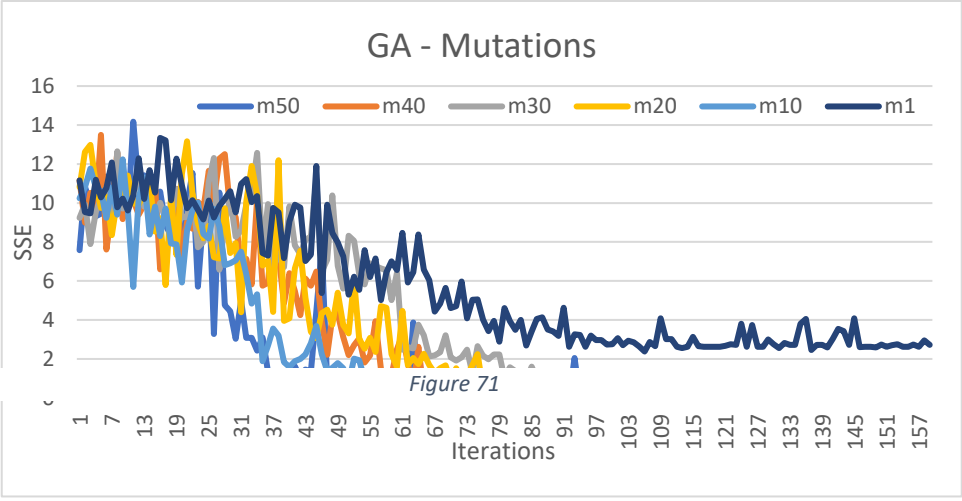

Figure 71

Following, with p=100 and m=50, crossover is tested as shown below in *Figure 12* and *Figure 13*. Similarly to mutations, the performance seems to be erratic, with co=100 and co=10 performing fairly well, while the middle values are not as great. Again, this may be attributed to variance in the problem initialization. Overall, co=100 seems to perform the best across multiple tests.

Overall, the best performing model is the GA with p=100, m=50, and co=100. The output is shown below in *Figure*

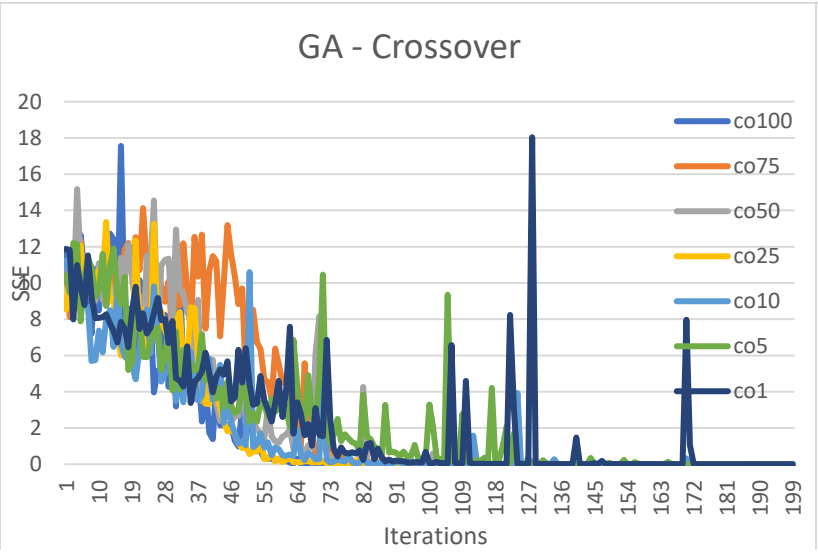

Figure 8

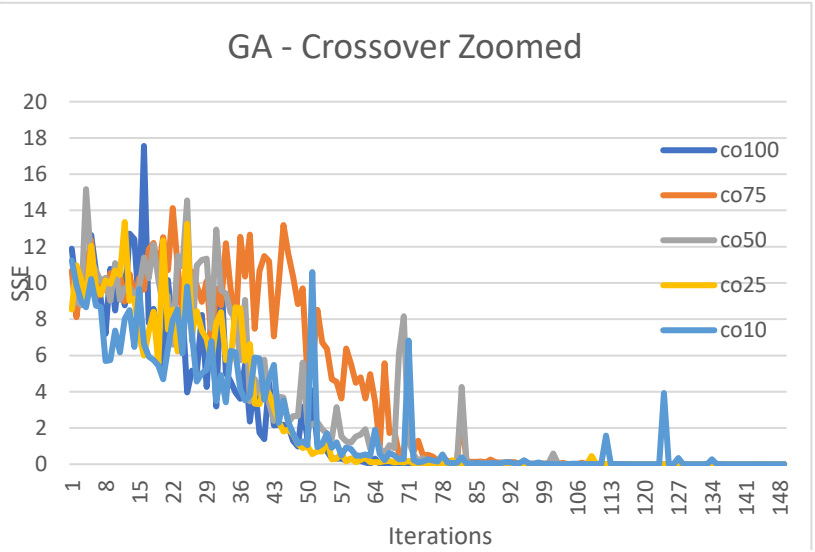

Figure 13

14, outperforming the initial model by converging to 0 error by 70 iterations, with no bumps up after that point. This is much better than RHC and SA, although still behind Backprop in terms of iterations. GA performs better by keeping track of a population of points rather than a single point. Therefore, the optimal point has a much higher chance of continuously being tracked, and a much lower chance of being lost once found. Moreover, mutations and crossover rates play a part in finding the optimum. They embody the idea of "exploration vs exploitation" that is introduced in SA, but implement it more naturally by allowing jumps of exploration (crossover and mutation) to still be tangentially related to the samples, so as to not stray too far off like SA has the potential to do.

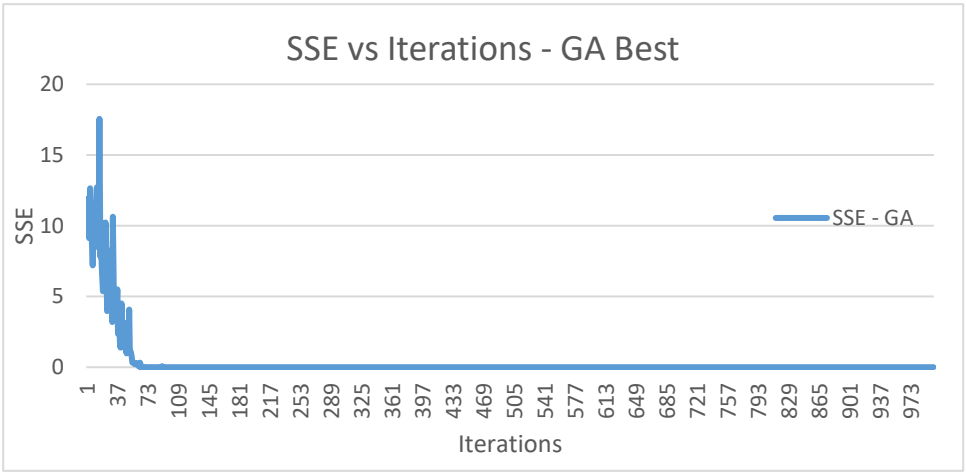

Figure 94:

That being said, exploration is not as useful for our problem, as previously explained. However, although the hyperparameter searches don't highlight too big of a difference in crossover and mutation rates, together the changes in these two parameters do boost performance overall. With a larger crossover and mutation rate, the pool of possible inputs is effectively expanded at a faster pace. This could be bad if we were only tracking one point, since we could expand to a point with high error. However, since we are keeping an entire population of points, this actually increases the chances that we stumble on inputs that have lower error. Therefore, our model converges to 0 error with much fewer iterations, and GA beats RHC and SA in terms of iterations. On the downside, GA took 3.772 seconds, which is over 6 times slower than RHC and over 13 times slower than SA. This is noticeably slower and is a similar principal to what was described before as a tradeoff between fast iterations and number of iterations. Since each GA iteration must keep track of a whole population and do multiple crossover and mutation, it takes longer per iteration. Therefore, even though it has less iterations, the overall runtime takes longer. Since the cost function in this case is easy to calculate, RHC and SA come out ahead in terms of runtime. GA still runs faster than Backprop by about 25%, since it's still less involved per iteration.

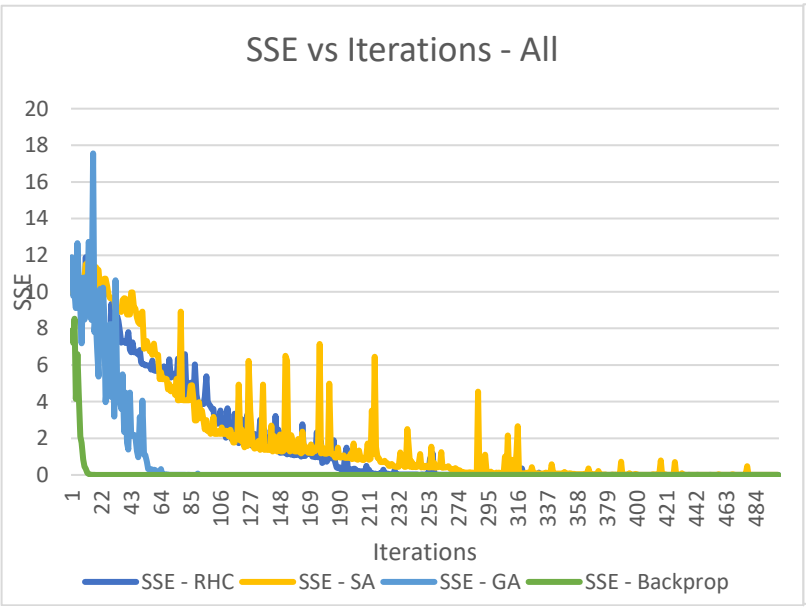

Figure 11

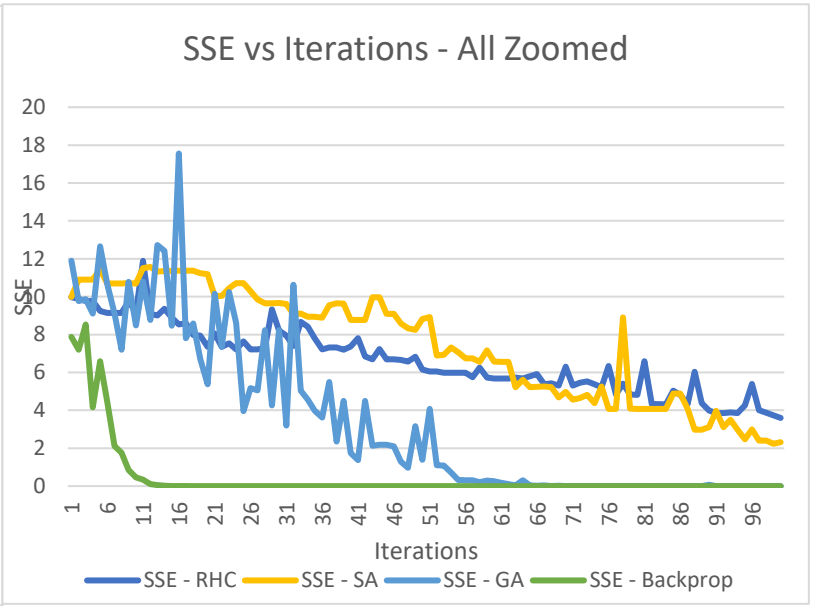

Figure 106

Figure 15 and *Figure 16 (zoomed)* above show a final comparison of the sum of square error output for all 4 algorithms: RHC, SA, GA, and Backprop. RHC and SA are similar in iterations until convergence and runtime, with SA behind in the former and ahead in the latter. GA beats both in terms of iterations until convergence but is worse in runtime. Overall Backprop is still the best in iterations until convergence but is the worst in runtime. This makes intuitive sense, since in the context of neural nets, weights are differentiable, so backprop follows the gradient and leads to lower error with fewer iterations. However, the difference in runtime shows, especially considering that all four algorithms lead to a model that has 100% test accuracy. This shows that in cases such as this where the problem is simple and the cost/fitness function is computationally cheap and easy to compute, algorithms like RHC and SA are enough to find the optimal answer in a short amount of time. However, the analysis shows that GA and Backprop do take less iterations and are less susceptible to harder problems, and thus may be desirable to use in cases where the cost/fitness function is harder/computationally more expensive to compute.

Discrete Optimization Problems

So far, RHC, SA, and GA have been compared with each other, as well as with Backprop with regards to a neural net. The advantages and disadvantages highlighted are under continuous conditions, yet usually optimization

algorithms are used to solve discrete problems. The advantages and disadvantages of each still hold true in the context of discrete problems, and these three discrete optimization problems are defined to highlight them. In addition, a fourth algorithm MIMIC is highlighted. MIMIC is a RO algorithm that keeps track of the underlying distribution of optimal inputs as opposed to a specific sample or population and uses that to continuously find better improvements on that distribution until ideally it converges, i.e. the distribution only returns optimal points.

Traveling Salesman Problem – GA

TSP is a common NP-hard optimization problem. This implementation takes 50 random points on a 2D plane and calculates cost as Euclidean distance between the points. The goal is to minimize distance, or rather, maximize the inverse of the distance taken to travel to each point at least once. Optimal parameters for each algorithm are found as before, resulting in the fitness vs iteration curves shown below in *Figure 17.* A representation of the exact problem being solved is shown below in *Figure 18.*

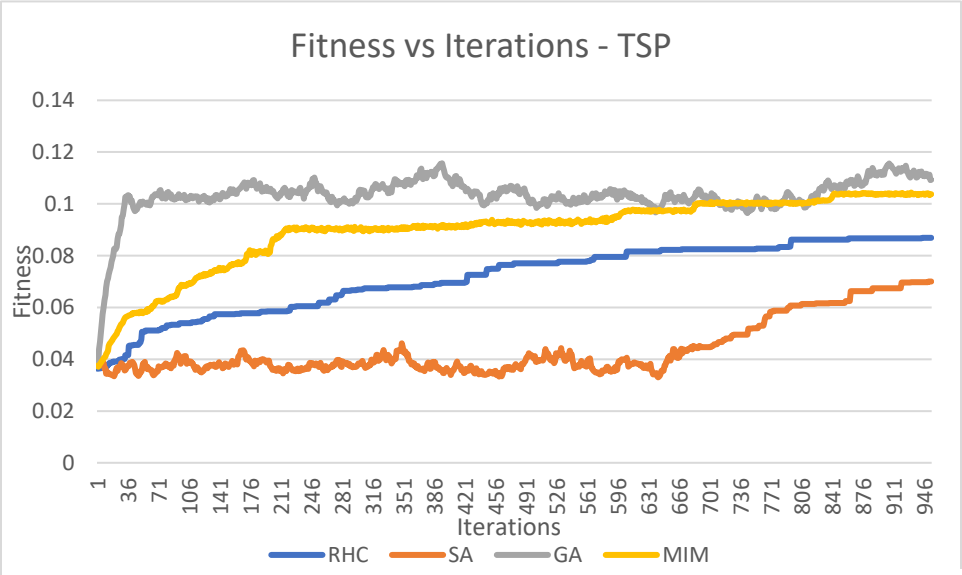

Figure 12

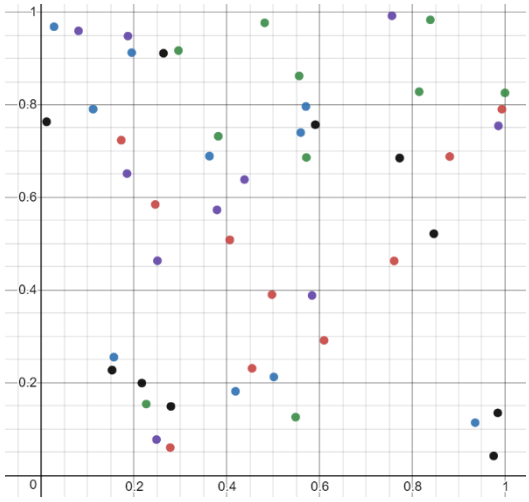

Figure 18

SA works poorly, taking nearly 700 iterations before even improving its fitness, which corresponds to its exploration rate being high. With such a complex problem with multiple combinations of distances and routes, SA takes a long time before settling cooling and settling in a space that allows it to find slightly better solutions. However, because of the resetting nature of SA, it fails to find a great solution in the 950 or so iterations it ran. RHC performs better, slowly improving over time, which corresponds to the more random nature of it. However, it also performs weakly on a hard problem such as this. As previously mentioned, both SA and RHC perform well on easily calculable and easily solvable problems, providing fast and optimal solutions. However, with a hard problem like this with many solution spaces and computationally expensive fitness functions, they pale in comparison to the other two.

MIMIC starts off improving fairly rapidly, but after about 230 iterations, only makes incremental increases. This shows that clearly poor inputs are easily cut off near the beginning, shaping a distribution that captures high performing inputs pretty well. However, as the cuts get smaller and smaller, it gets much harder to find massive improvements, and thus only increases by small amounts. GA goes through a similar effect, although it is much more pronounced. Since GA keeps track of a population, it very quickly finds the pool of top performers within 40 iterations. From there, since crossovers and mutations are random, it becomes more susceptible to variance compared to MIMIC, even dropping below its performance at around 700 iterations. However, it ultimately comes back around as it follows a general upward trend in increasing the fitness function. This intuitively makes sense, since although the crossovers are random, they represent a sort of conditional dependence between the inputs. By

crossing over, the samples preserve some parts of that dependency, which may correspond to a part of the path that is shorter. Therefore, GA excels at solving TSP compared to the other three algorithms.

Flip Flop – SA

Flip Flop is a simple problem that is solvable in polynomial time. This implementation starts with an 80 bit bitstring that is evenly spread, and the goal is to find a bitstring that maximizes the number of alternating digits, e.g. for input of 5 bits, 10101 or 01010 instead of 11111. The fitness function therefore is the number of alternating digits, bounded at exactly 80. Optimal parameters for each algorithm were found as before, resulting in the fitness vs iteration curves shown below in *Figure 19.*

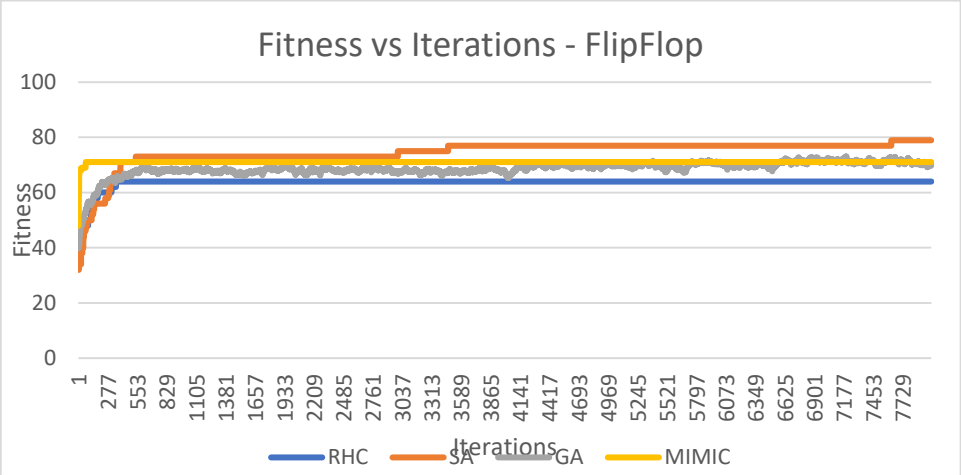

Figure 139

All four algorithms work fairly well, since the problem is simple. RHC has the worst performance out of the four, converging below 65 early on without improving. MIMIC and GA both perform slightly better, converging around 70, although MIMIC reaches that point much earlier on and stops, while GA continuously shifts back and forth. SA is the clear winner here, making use of exploration to expand its search spaces and find 79, just 1 away from the global maximum.

Although this problem is simple, it has two global maxima and many other local maxima, which results in some algorithms getting stuck. RHC works by searching step by step, but without extra measures to avoid getting caught in local maxima, it pales in comparison with the other three. MIMIC and GA work well by capturing a larger part of the input space all at once, which prevents the optimal point from jumping back and forth between different basins of attraction. They work to capture these basins of attraction all at once, whether that's with developing an underlying distribution with MIMIC, or just keeping track of multiple points with GA. However, these two still get stuck in local maxima, as they aren't as good as SA when it comes to exploration. With a problem that has as many local maxima as this one, SA excels by allowing jumps to different input spaces to occur at random. This also sidesteps the problem of differing attraction basins and allows jumps even in nearly converged spaces to potentially reach the global maximum. This can be seen by looking at the curve. The fitness stays the same across thousands of iterations, looking like it's converged, until it suddenly jumps to a more optimal point. Thus, SA is best for Flip Flop.

Max K-Coloring – MIMIC

Max K-Coloring is an NP-Complete optimization problem. The goal is to find a path of coloring so that no colors are adjacent. In effect, it is a much more complex flip flop problem that involves k levels of adjacency across L connections per N vertices, which in this case is 8 levels (colors), 4 connections, and 50 vertices. The fitness function is a little more complicated, but it captures time taken by the algorithm, along with whether a solution

was found. An important note is that both RHC and SA failed to find a solution at all, even with arbitrarily high iterations. Optimal parameters for the other two algorithms were found as before, resulting in the fitness vs iteration curves shown below in *Figure 20.*

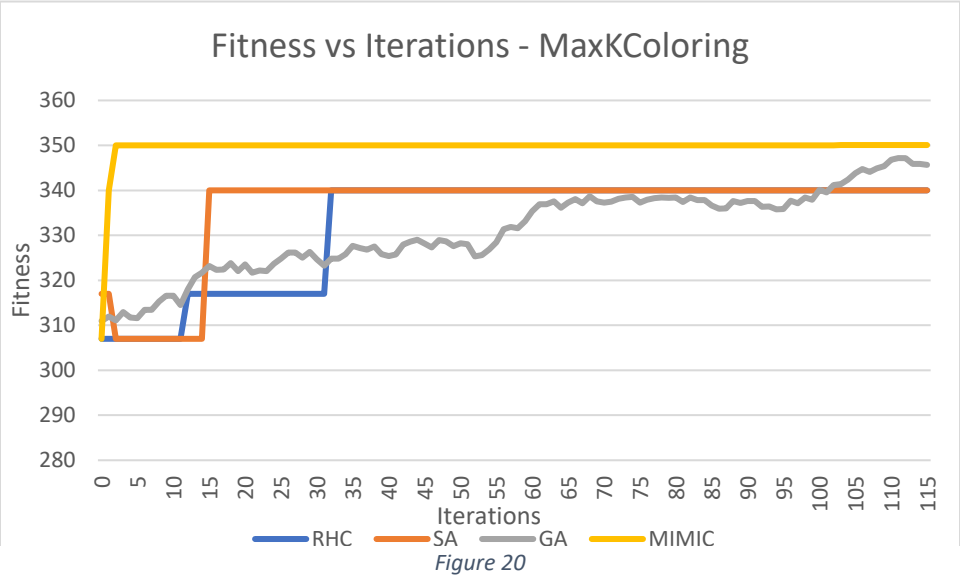

Figure 20

RHC and SA failed to find a solution to the problem, which is understandable due to the problem's hard nature. This problem also has many local optima, so therefore RHC struggles to find its way around randomly step by step. SA should be marginally better at defending against being stuck in local optima, which the curve does show since it improves quicker than RHC, but that doesn't help it much if it ultimately cannot converge upon an actual solution. This shows that a solution to this problem needs something that can keep track of more than a single point at a time.

MIMIC and GA were both able to find solutions to the problem, with MIMIC being the clear winner in finding an optimal solution within 3 iterations. GA improves upon its solution across iterations but doesn't quite reach as optimal of a solution in as fast of a time as MIMIC. These algorithms excel at this problem because the inputs are easy to eliminate. A coloring that breaks the rules is easily apparent, and MIMIC and GA are able to shave off the wrong inputs easily. With GA, the crossovers and mutations are still random, and thus sometimes notable improvements may be lost. However, with MIMIC, it easily keeps track of the entire underlying distribution and only gets better, and therefore outperforms GA by a lot.

As shown in these problems, RHC and SA are ideal for simpler problems, and trade off power for their simplicity and speed. MIMIC is ideal for harder problems, using a lot of computing power but being able to capture more about a problem and its underlying structure. GA lies somewhere in between. It doesn't capture structure but is able to hold on to more information in the process of finding a solution, and therefore also is able to solve harder problems. Overall, each algorithm has its own pros and cons.