Neural Network Weights Optimization

Dataset (Recap)

The wine quality dataset from the last assignment is used. In this dataset, the inputs are all numeric values (e.g. PH values, fixed acidity) and the outputs are binary classes of 'good' or 'not good'. In the last assignment, I concluded that neural network seems to perform badly on this problem space comparing to other algorithms like k nearest neighbors or boosted decision tree. I also concluded that this problem is very difficult or might be impossible to solve by using just one single generalized solution (like neural network or supported vector machine). For this assignment, I want to try solving this problem using neural network with different weights optimization techniques.

Neural Network Parameters

In the last assignment, the neural network that achieved the best on this dataset had 2 hidden layers, with 6 nodes in the first and 9 nodes in the second. Therefore, the neural network in this assignment will have the same dimension.

Neural Network Weights Optimization Problem

For this assignment, I will be using the library ABAGAIL. In this library, there is a predefined neural network weights optimization problem. In this problem definition, the fitness function is **1/error**. The error here is defined as the sum of squared error of all the predictions. A neighbor of a set of weights is defined to have one of the weights plus a random decimal number sampled in the range of [-0.5, 0.5], while the other weights remain unchanged. Mate/ cross-over of two sets of weights is defined to be a set of weights with each weight randomly sampled from either one of the parent set at the corresponding index. Mutation of a set of weights is defined to be a random neighbor of the current set of weights.

Randomized Hill Climbing (RHC)

Main Idea

The main idea of the RHC algorithm is that taking a random starting point in the problem space, and slowly move to a neighbor that gives a more optimal evaluation based on the defined fitness function.

Graphs

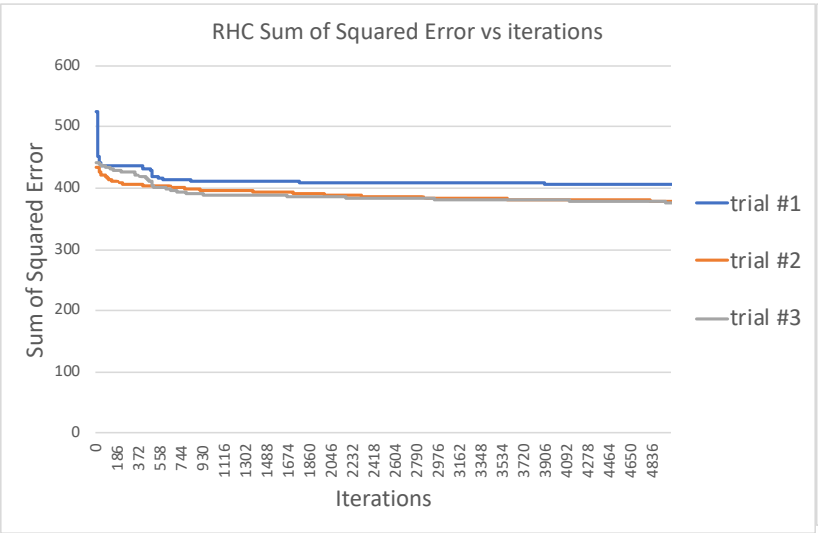


Figure [1.1] *Iteration = 5000*

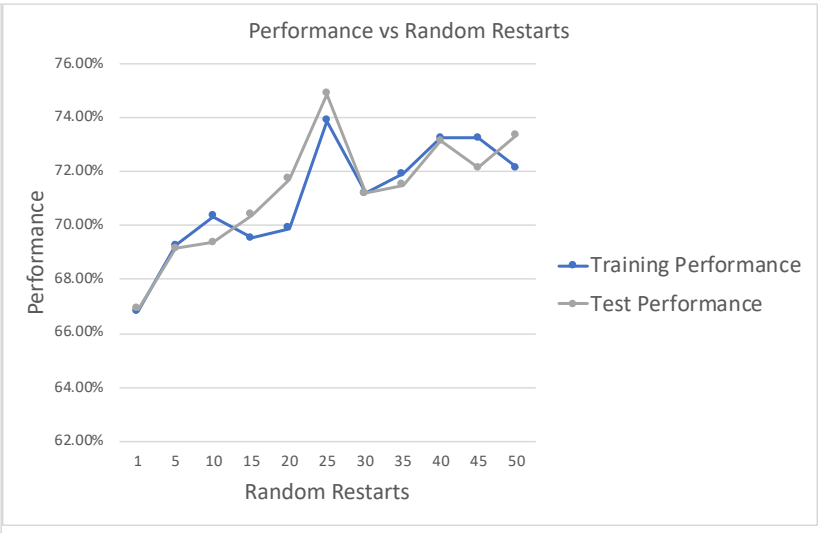


Figure [1.2] *Iteration = 4000*

Test Performance (Time)	Trial			
Random Restart	1	2	3	Average
25	70.276%(1667.203s)	69.663%(1724.841s)	72.319%(1731.837s)	70.753%(1707.96s)
40	70.582%(2762.298s)	71.91%(2773.755s)	70.787%(2751.836)	71.093%(2762.63s)

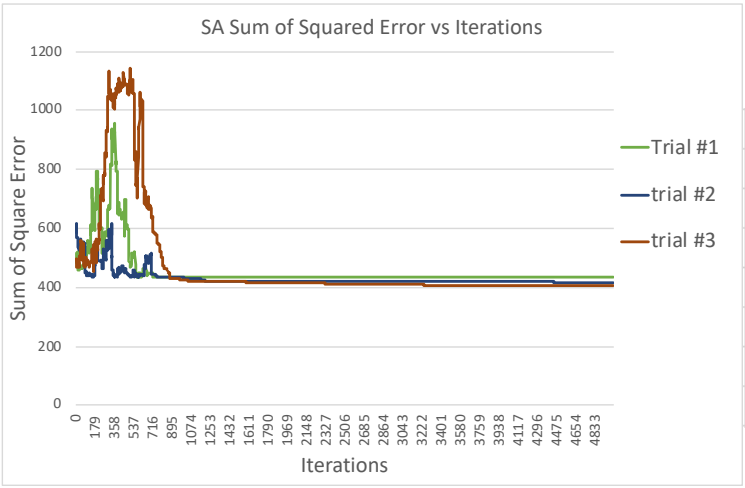
Figure [1.3] *Iteration = 4000*

Simulated Annealing (SA)

Main Idea

SA algorithm is similar to the RHC algorithm in terms of starting from one random starting point, and for each iteration, a neighbor is picked and evaluated for the next step. However, in SA, if the neighbor picked is less optimal than the current, there is still a chance of stepping to it. Essentially, the higher the temperature, the more likely to step to a less optimal neighbor, and the cooling rate controls how fast the temperature cools down. In ABAGAIL, the chance of stepping to a less optimal neighbor is e^((neighbor fitness - current fitness)/temperature). In this function, if the (neighbor fitness - current fitness) term stays the same, the higher the temperature the higher chance. It is also true that a higher temperature will allow a higher chance to step to a neighbor that has a much lower fitness value than a lower temperature will. This feature can effectively help the algorithm to escape local optimum.

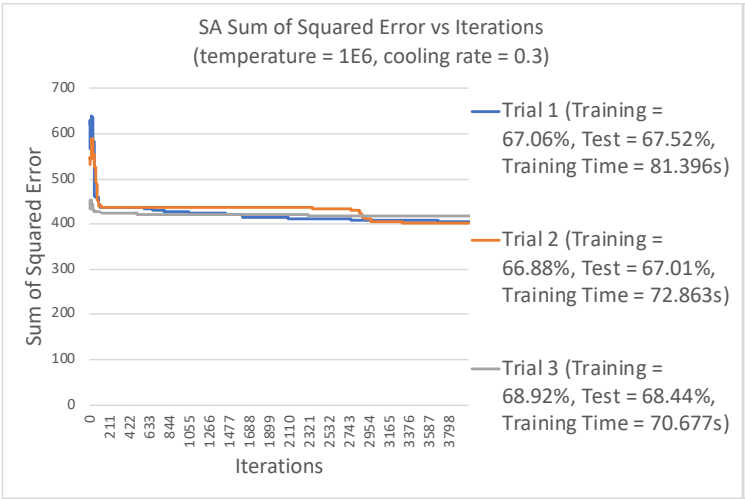
Graphs



Test Performance	Cooling Rate		
Starting Temperature	0.3	0.6	0.9
1.00E+02	66.70%	66.70%	69.36%
1.00E+04	67.72%	67.72%	67.93%
1.00E+06	71.09%	69.25%	69.66%
1.00E+08	69.77%	67.82%	69.05%
1.00E+10	67.01%	67.01%	68.34%
1.00E+12	67.93%	68.03%	67.52%

Figure [2.2] *Iteration = 4000*

Figure [2.1] *Iteration = 5000, Starting Temperature = 1E11, Cooling Rate = 0.95*



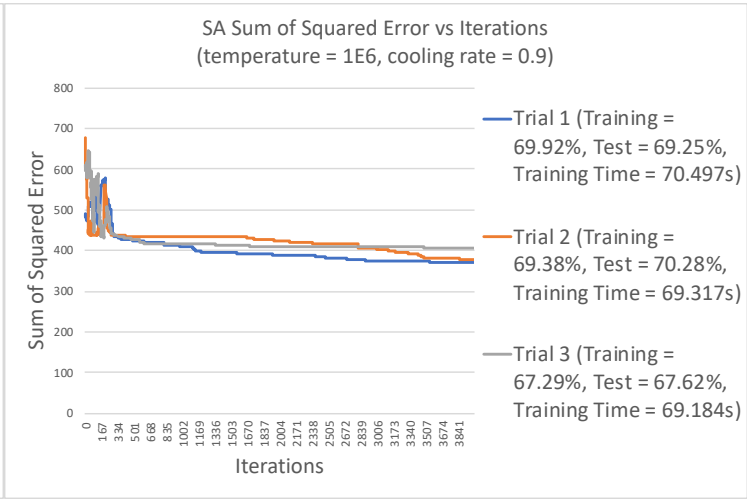


Figure [2.3] *Iteration = 4000*

Figure [2.4] *Iteration = 4000*

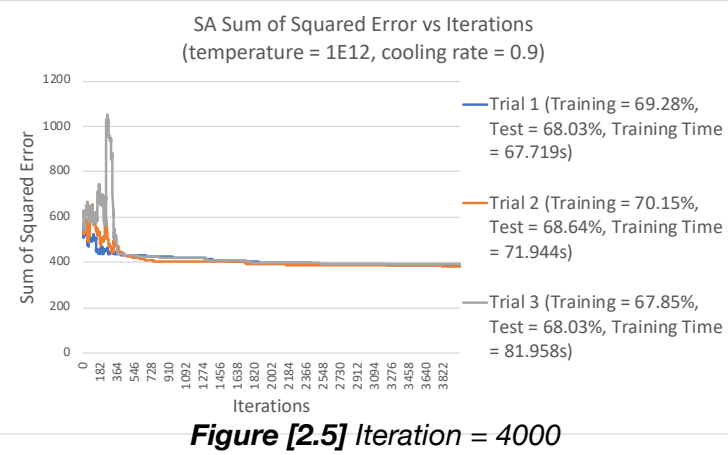


Figure [2.5] *Iteration = 4000*

Genetic Algorithm (GA)

Main Idea

In GA, there are three parameters that represent this concept: population size, mate number and mutation number. When the algorithm starts, the population size number of points are randomly scattered on the problem space. In ABAGAIL, For each iteration, part of the new generation of population is generated by producing mate number of new points by crossing over a pair of points in the last generation with high fitness values. The rest of the new generation is then filled by selecting the points with the highest fitness values from the last generation. Finally, randomly select mutation number of points in this new population for mutation. Mate/ cross-over mainly functions as the exploiting feature with the assumption that the cross-over of two sets of weights with high fitness values would produce a set of weights with higher or equally high fitness value. Mutation mainly functions as the exploring feature, which can help the algorithm explore the problem space and escape a local optimum.

Graphs

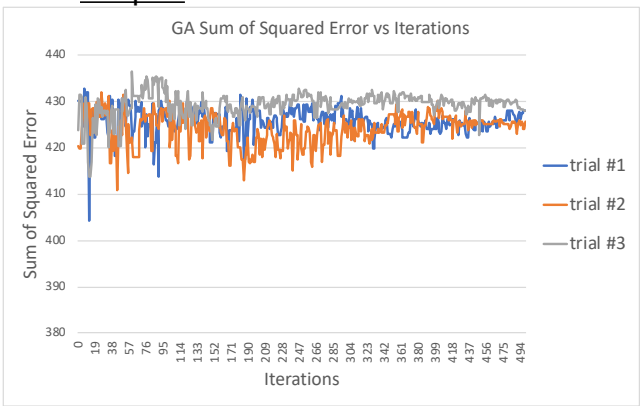


Figure [3.1] *Iteration = 500, Population Size = 200, Mate Number = 100, Mutate Number = 10*

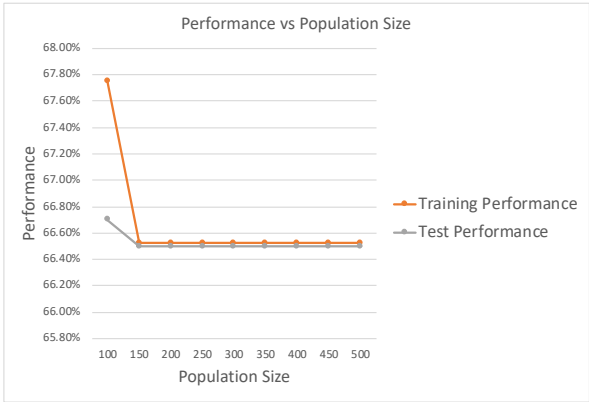
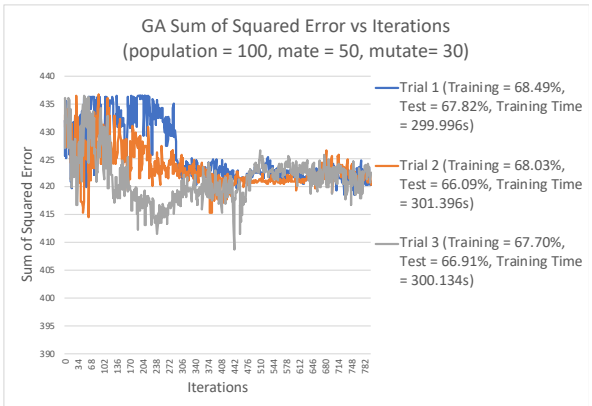


Figure [3.2] *Iteration = 800, Mate Number = 100, Mutate Number = 10*

Test Performance	Mutate #				
Mate #	10	30	50	70	90
10	66.50%	66.50%	66.50%	67.01%	66.50%
30	66.50%	66.50%	66.50%	66.60%	66.50%
50	66.39%	66.60%	67.72%	67.21%	66.50%
70	66.50%	66.50%	66.50%	66.50%	66.50%
90	66.50%	66.80%	66.50%	66.50%	66.50%

Figure [3.3] *Iteration = 800, Population Size = 100*



Fitness Value	mutate #				
mate #	10	30	50	70	90
10	0.0022915	0.0023774	0.0023523	0.0024272	0.0024075
30	0.0023918	0.0024501	0.0023348	0.0023329	0.0022978
50	0.0023168	0.0024687	0.0023799	0.002437	0.0023197
70	0.0023971	0.0023998	0.0023382	0.0023403	0.0023323
90	0.0023788	0.0024091	0.0023554	0.0023812	0.002333

Figure [3.4] *Iteration = 800, Population Size = 100*

Figure [3.5] *Iteration = 800*

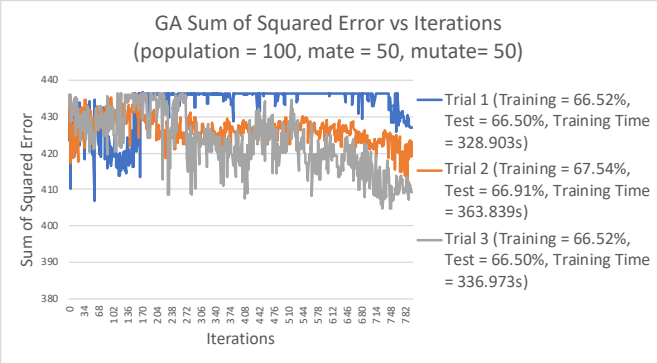


Figure [3.6] *Iteration = 800*

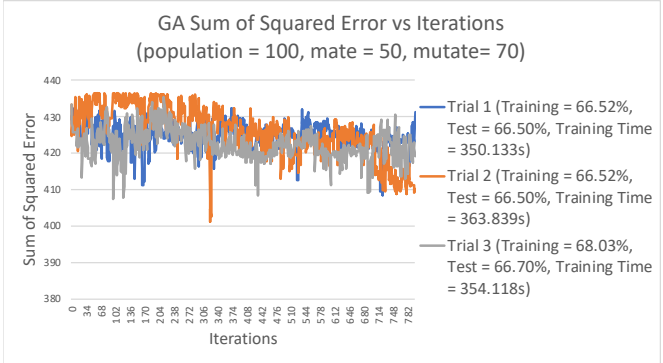


Figure [3.7] *Iteration = 800*

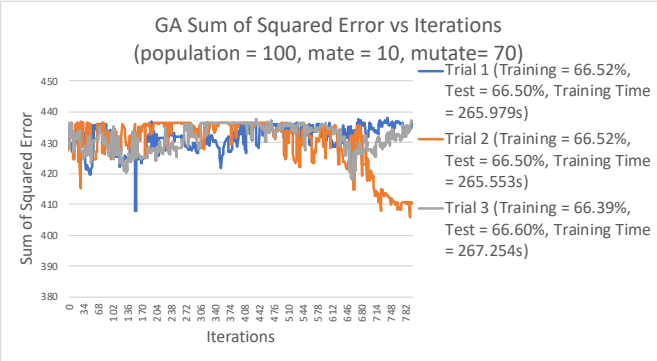


Figure [3.8] *Iteration = 800*

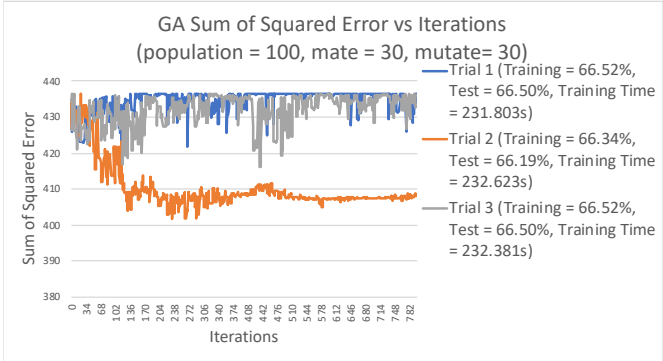


Figure [3.9] *Iteration = 800*

Analysis

Randomized Hill Climbing

Figure [1.1] shows the sum of squared error vs iterations graph. As expected, the error decreases as iterations increase. The main purpose of this experiment is to decide on a good iteration number. Since in ABAGAIL, the stop condition is iteration instead of convergence, this experiment is needed to gain an idea on an appropriate iteration number for convergence. Based on this graph, iteration number is set to 4000. Although it looks like the algorithm is converging at around 2000 iterations, the error keeps decreasing even at 3000 and 4000 iterations just with a lower rate. It seems like it is at something like a cliff or saddle point because it is not going completely flat like how it would at a local optimum or plateau. It is decreasing just by a little bit after many iterations, which means that the next step took many iterations to find; thus suggesting a cliff or saddle point. 4000 is a good iteration number because it lets the algorithm converge as much as possible without taking an unacceptable amount of time.

Figure [1.2] shows the performance vs random restarts graph. In this graph, the performance has an increasing tendency as there are more random restarts (ignoring the random spike for now), and the increasing tendency gets slower toward the tail of the graph as the tail seems to oscillate at around 40 to 50 random restarts. The spike at 25 random restarts merely means that the algorithm initialized a starting point at a good point on the problem space, but since the starting point is purely random, this spike just demonstrates that such a solution exists, and it is difficult to find.

After this, two more experiments are done with random restarts at 25 and 40, which are two of the spikes in Figure[1.2]. Three trials of each are done just to smooth out the randomness. Thus, Figure[1.3] is constructed. This table shows that the spike in Figure[1.2] is, as expected, a result of randomness. This table also helps smooth out the curve in Figure[1.2] because on average, more random restarts results in a better performance, even though there might be spontaneously very good results with lower random restarts. It can be concluded that 25 random restarts is absolutely enough to

achieve the best this algorithm can reach. **Randomized Hill Climbing was able to achieve 74.87% test performance with 25 random restarts and 4000 iterations.**

Simulated Annealing

Figure[2.1] has the same purpose as Figure[1.1], which is determining the ideal iteration number. In the beginning, when the temperature is still high, the curve jumps up and down because it was just moving almost freely, but after the temperature cools down at around 1000 iterations, the behavior looks exactly like the one from RHC because SA with low temperature is basically RHC. The iteration is thus decided on 4000 for the same reasons as RHC, and also because enough iterations should be given for the algorithm to converge in the situations when the temperature is high and cooling is slow.

Figure[2.2] is a table recording test performances of the neural network resulted from different combinations of starting temperature and cooling rate (In ABAGAIL, the higher the cooling rate, the slower of cooling [new temperature = current temperature * cooling rate]). The highlighted ones are the ones that had a value of greater than 68%. It shows that 0.9 is a good cooling rate, and 1E6 is a good starting temperature because most of values on the corresponding column and row are highlighted. The highest performance is at the intersection of starting temperature at 1E6 and cooling rate at 0.3. Ideally, I would also think high temperature with slow cooling rate should explore more problem space and result in the better solution. Therefore 3 more experiments with 3 trials for each are done with these sets of parameter values: (1E6, 0.3), (1E6, 0.9) and (1E12, 0.9) *[in the format of (temperature, cooling rate)]*

These experiments resulted in Figures[2.3, 2.4 and 2.5]. The behaviors are all expected. Figure[2.3] has a behavior of dropping to convergence right when the algorithm started due to its low temperature and fast cooling, averaging a test performance of 67.657%, which means the peak on the table is just due to a good randomness. Figure[2.4] has a behavior of jumping around a little bit before dropping to converge because the cooling is slow, averaging a test performance of 69.05%, which means that, as expected, these two parameter values do result in a relatively better solution. Figure[2.5] has a behavior of jumping around with a big range before dropping to convergence because the starting temperature is set very high, averaging a test performance of 68.23%, which is ok. The huge error boost actually suggests a cliff.

These experiments and graphs show that simulated annealing isn't able to achieve an outstanding result using any combinations of starting temperature and cooling rate, but there is a tendency to achieve a better result using a middle range starting temperature like 1E6 and a slow cooling rate like 0.9 or even 0.95. It can also be concluded that jumping around (high starting temperature) doesn't seem to help the algorithm so much because in the algorithm implementation, when the starting temperature is high, the algorithm is jumping around with no memory at all, so it might even escape a local optimum to go to an even less optimal local optimum. Given this analysis, a lower temperature and a slow cooling rate should be used to avoid jumping rapidly but to jump smartly. **Simulated Annealing was able to achieve 71.09% test performance with starting temperature of 1E6 and cooling rate of 0.3**, which confirms my analysis.

Genetic Algorithm

Figure[3.1] has the same purpose as Figure[1.1] and Figure[2.1], which is to determine iteration numbers. Figure[3.1] is resulted from 500 iterations because genetic algorithm takes a way longer time to run. Given this graph, the iteration number is set to 800 because in this graph, the curves are almost converging, so given 800 iterations, it should be able to reach a convergence. The behavior of the graph is also interesting because the curve keeps jumping up and down just with the jumping range slowly decreasing, which means that all the points in the population are getting more similar (converging in terms of most common population instead of lower error, which will be explained later), so the mutation and cross over don't produce a significantly different generation any more, thus converging.

Figure[3.2] is a very simple test performance vs population graph, and the graph obviously show that increasing population size seems to be not helping the test performance at all. It is worthy to note that since one of the class distribution is 66.5%, so a test performance of 66.5% basically means classifying everything as one class. Thus in the future experiments, population will stay at 100.

Figure[3.3] is a table recording test performances at different combinations of mate number and mutation number. However, in most cases, the best the neural network can do after training using GA is simply classifying everything into the majority class. Therefore, I constructed Figure[3.4] for a deeper look into what is happening in the algorithm. In this table instead of test performance, fitness value is recorded. The highest ones in both tables are highlighted, and because Figure[3.3] represents the test performance and Figure[3.4] basically represents the detailed training performance, the highlighted cells are reasonably different. In Figure[3.4], all the fitness values are not even that different from each other, which would mean that the neural networks resulted from different combinations of mate number and mutation number don't even have very different weights. Finally, to take a closer look at the behavior of this algorithm at different parameter values, more experiments are done using parameter values that have a slightly better result.

Figures[3.5 - 3.7] are graphs from the supposedly better parameter values. These figures don't even seem to converge, and all the graphs, although with different parameter values, all seem to have very similar behaviors. Some of them seem to begin to converge before they would break out of the convergence again (orange curve in Figure[3.6] at around 700 iterations, gray curve in Figure[3.7] at around 700 iterations). These patterns and the constantly low training and test performance confirm with my theory of this optimization problem being an extremely complex problem space because GA has no knowledge about the gradient of the space, and, unlike other algorithms, its goal is not to find the most optimal solution, but a good-enough solution. However, due to the complex space, a good-enough solution reachable by ABAGAIL GA is, seemingly, just classifying everything into the majority class (66.5%). Since there are many possible definitions for GA(cross-over, mutation, parameter choices), the ones used here in ABAGAIL just might not fit my problem space. ABAGAIL GA has a simple characteristic: only points with high fitness values around the same optimum would result in a converging behavior. For example, given a 2-dimensional space and optima (1,100) and (100, 1). If two points sampled are (1, 99)

and (99, 1), which both have high fitness values, would produce either (1, 1) or (99, 99) through cross-over, which are actually even farther from any optima. Thus breaking into a non-converging behavior. This complex problem space really highlights this characteristic in a negative way. In the current problem space, a similar behavior is captured. These behaviors actually tell me that the problem space has extremely many local optima. Therefore, when 100 points are scattered onto the problem space, every point might be next to a unique local optimum, and cross-over and mutation just bring the 100 points all away from their current optima to places where other optima are nearby. Even when cross-over finally makes them converge to some optima, a mutation might again set many points next to other new optima. Thus each point would just spend the entire time jumping from one optimum to another one. Given this analysis, the cause for the non-converging behavior here could be high population number (each point next to a unique optimum) or high cross-over and mutation number, especially mutation numbers, which are actually true looking at all the graphs. In Figure[3.2], the performance actually gets worse as the population gets larger, but the algorithm is able to at least capture the "good-enough" weights of just guessing the majority class. When the mate and mutation numbers are high, especially the mutation number, Figure[3.5-3.7] proves that the algorithm is never able to converge, but just jumps around randomly. Figure[3.1] forms an apparent contrast from these figures. This one was done with population size of 200, and just mate number of 100 and mutation number of 10, which would be equivalent to mate number of 50 and mutation number of 5 when population size is at 100. This is why Figure[3.5] has the best test performance among these figures, but it still failed to converge due to the large mutation number. Overall, ABAGAIL GA doesn't seem to be a good algorithm for this problem at all. **The GA was just able to achieve a test performance of 67.82% with population at 100, mate number at 50 and mutation number of 30.**

Conclusion

This problem space has extremely many local optima and saddle points with just narrow paths to more optimal solutions. To get to the best solution possible, what is needed is the power and perseverance of overcoming plateaus, various saddle points, shoulders, local optima, etc., but the cliffs are deadly to SA because one step can cause it to fall (gray curve in Figure[2.5]). Therefore, RHC with random restarts and many iterations is the most optimal solution so far because it has both the power to overcome local optima, shoulders, plateaus (random restarts) and cliffs(iterations). This property reminds me of the learning rate and momentum for gradient decent, which is just a variation to randomized hill climbing with more control over each step, or SA with low starting temperature (not fall off a cliff), extremely slow cooling rate (overcoming local optima, shoulders, etc.) and even more iterations. **The best solution so far was achieved by RHC with the parameters of 25 random restarts and 4000 iterations at a test performance of 74.87%**. Although this is not an ideal performance, it is the best possible performance using these algorithms. For future possible improvement on this problem, SA or GA should be used first to provide an educated starting point for the Gradient Decent. Random restarts should also be used so that the large complex problem space can be effectively searched.

Other Optimization Problems

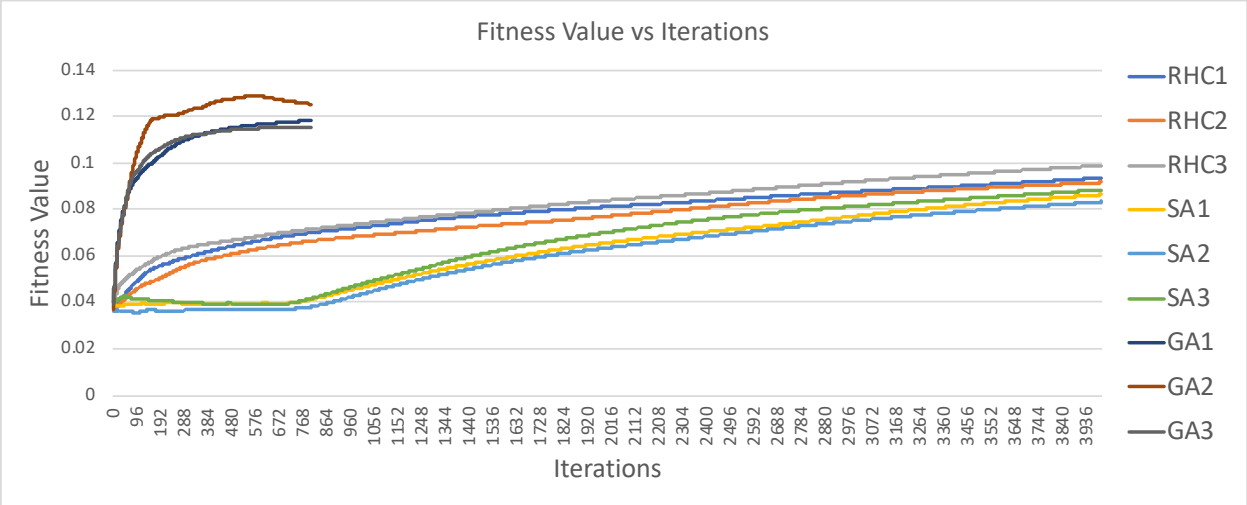
Traveling Salesman Problem (TSP)

Description: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city? This is a well-known NP-hard problem in combinatorics optimization.

Problem Definition:

- Fitness Function: Given a list a cities (vertices), the inverse of the traveling distance while traversing though the list and going back to the first city (vertex)
- Neighbor: Swapping any two vertices in the list
- Cross-over: For each index *i* in the list, parent *a* has city *a[i]*, parent *b* has the city *b[i]*, and child *c[i]* needs to be filled. If *a[i]* is in child list already, but *b[i]* is not, fill *c[i]* with *b[i]*. If the opposite, fill *c[i]* with *a[i]*. If both *a[i]* and *b[i]* are not in child list, pick the one that has a closer distance with the previous city. If both *a[i]* and *b[i]* are in the child list already, pick a random city that is not in the child list.
- Mutation: Randomly pick a neighbor

This will highlight Genetic Algorithm because of two characteristic of this problem: NP-hard and combinatorics optimization. NP-hard means this problem is very difficult to solve, so many approximation functions are developed for TSP just to get a good-enough solution. Genetic Algorithm is also known be almost the best at giving a good-enough solution on a difficult problem. This concept of Genetic Algorithm highly aligns with the idea of solving TSP. Since TSP is a combinatorics optimization problem, permutations of the cities are the basis of the problem space, which is also a key concept in the nature of GA because the idea of combining and permuting is fully embedded in the concepts of cross-over and mutations. In this problem, RHC and SA might be difficult due to the huge problem space. For a N size list, there are N! different permutations. For each point, there are N^2 possible neighbors, and probably only very few of them can provide a better evaluation, especially in the later phase (narrower optimal paths).



The graph came out exactly like expected. The RHC had a good start, but the increase gets slower due to the reason discussed above: a good neighbor is getting more difficult to find. The flat region in the beginning of SA proves that good neighbors are very hard to find. When the temperature doesn't allow less optimal steps any more, the curve starts to behave like RHC. GA, on the other hand, doesn't rely on the derivative and the neighbor concepts. It is able to reach some highly fit points within few iterations because the cross-over and mutation can effective search through the problem space and put the population on good solutions. The good-enough solutions are actually much better than the solutions from SA and RHC.

RHC			
	0.1269992		
SA			
	cooling rate		
temperature	0.95	0.3	
1.00E+12	0.1065244	0.0993344	
1.00E+03	0.1105505	0.1084201	
GA			
	mutate		
mate	10	20	30
150	0.1431483	0.1543941	0.1589961
100	0.1484362	0.1477963	0.1564928
50	0.1600182	0.1426176	0.148193

To ensure the performances of these algorithms, more tests are done with different parameter values (the table on the left records the fitness value, and each value is an average of three trials to smooth out the randomness). RHC in this one doesn't have random restarts as a parameter, but from the data, it is already obvious that the GA is much better than the other two algorithms with all combinations of parameters and way less iterations too.

Flip Flop

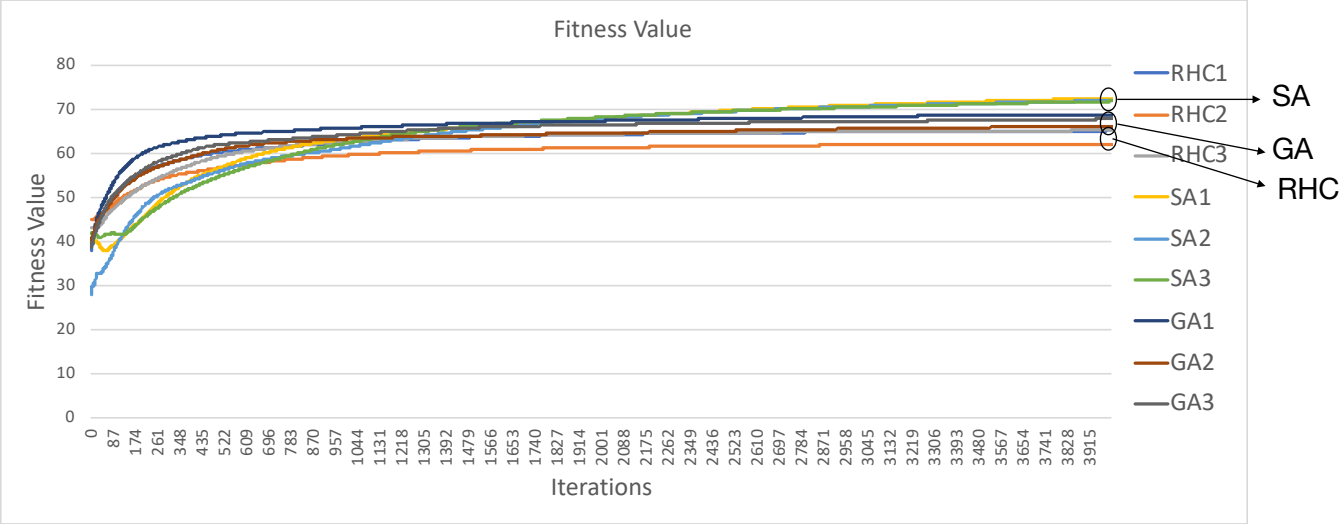
Description: Given a binary string, try to find the ordering that has the most alternating bits. A very simple problem.

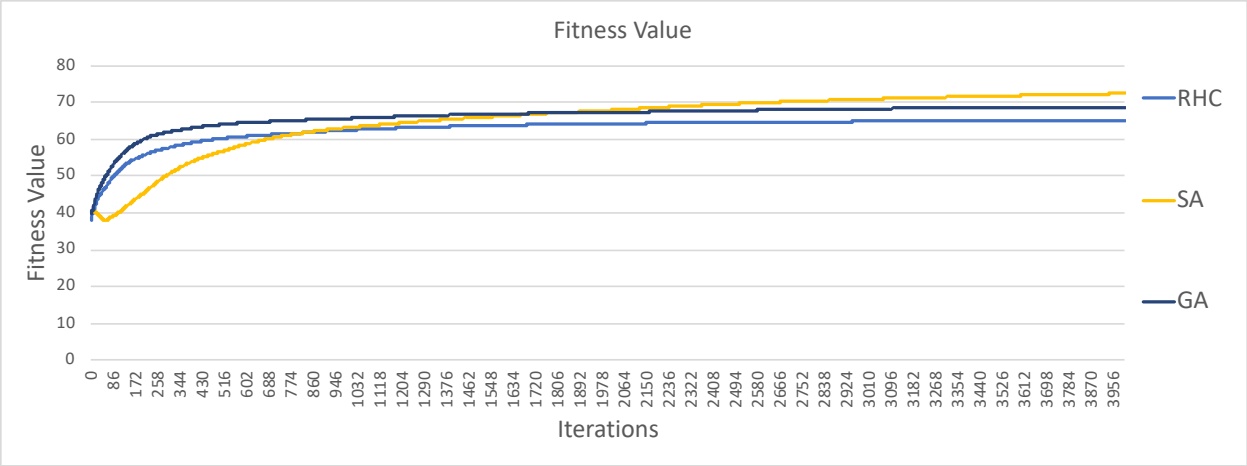
Problem Definition:

- Fitness Function: Given a binary string, whenever the bit alternates, the fitness value increments by 1. For example, when a 0 is followed by a 1 or a 1 is followed by a 0, fitness value increments by 1. When a 0 is followed by a 0 or a 1 is followed by a 1, fitness value doesn't change.
- Neighbor: replace a random index in this bit string with 0 or 1 randomly
- Cross-over: Choose a random index in the string, for indices less than this index, fill with one of the parents' corresponding bits. Fill the rest with the other parent's corresponding bits.
- Mutation: Randomly pick a neighbor

This problem is a very simple optimization problem, but effectively highlights the SA algorithm. The problem space is very simple with two global optima: alternating bit strings starting with 1 or 0. However, in this problem space, there is no local optimum because for any bit string ordering, there is a way to get the bit string to a fully alternating bit string through iterations of toggling one bit. For example, 1111 -> 1110 -> 1010. RHC would work well on this because this problem space has a massive amount of shoulders. For example, at 1011 and the neighbor 1001 would function as a shoulder to RHC. Actually at a bit string that looks like 1011, among all the neighbors {0011, 1111, 1001, 1010}, only one of the neighbors has a more optimal fitness value. Image a bit string that is 80+ bits in the form of 1011011..011, an optimal solution would be very hard for RHC to find. In fact, RHC would have a harder time finding more optimal neighbors as iterations get more. However, SA is very good at overcoming shoulders, thus has a high performance on this problem space. GA is always known for

finding good-enough results for difficult problems, but for a problem like this, a good-enough result is actually not good enough anymore. Sometimes, GA might still stumble





It might be a little messy in the first graph, but the second graph clearly shows that SA outperforms other algorithms. Even though the result doesn't differ by as much as the previous problem, it still shows that SA is doing well on this problem space. RHC has an expected behavior of getting stuck and slow improvement at around 400 iterations for the reasons discussed above. GA is also doing well at finding solutions that are good but just not as optimal as SA. Even at the end of 4000 iterations. SA is the only curve that still doesn't seem to converge.

RHC			
	63.333333		
SA			
	cooling rate		
temperature	0.95	0.3	
1.00E+02	76.666667	70	
1.00E+01	75.666667	72.666667	
GA			
	mutate		
mate	10	20	30
150	72.666667	73	74.666667
100	72.666667	74.666667	73.333333
50	72	74	71.666667

Similar to the previous problem, more tests are done to confirm the performances. RHC, as expected, has a poor performance. SA when at cooling rate of 0.95 gives a considerable better optimization than GA. Since in the graphs above, SA doesn't seem to converge given 4000 iterations, one more test is done with 8000 iterations. The fitness actually got to 79, which is the highest possible given a string of 80 bits. It is reasonable to conclude that SA does the best on the flip flop problem.