Harsh Patel

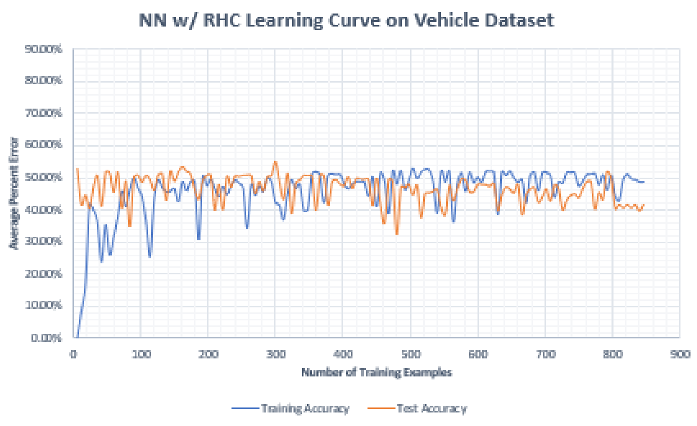
CS 4641 – Machine Learning

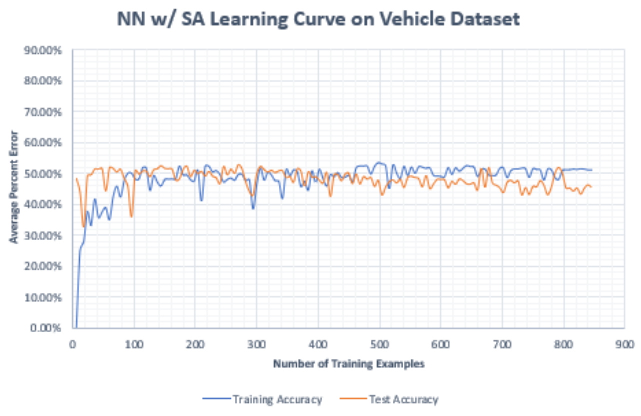
Assignment 2: Randomized Optimization

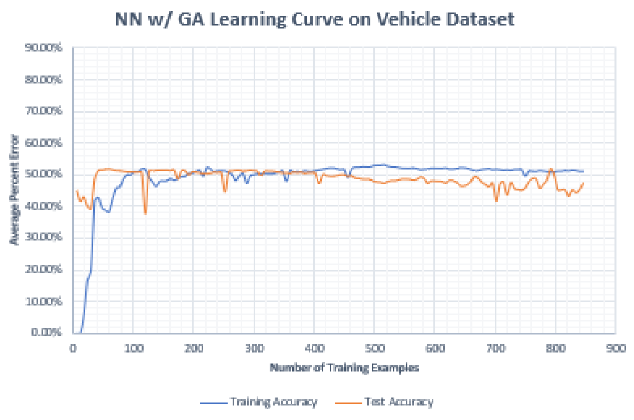
Neural Network Revisited

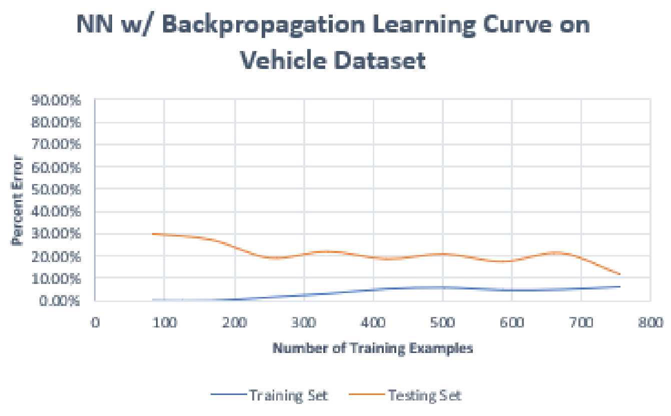
Background: In Assignment 1, we were tasked to implement (or steal) a neural network implementation and observe how changes to hyperparameters (such as learning rate and momentum) would influence the accuracy for the underlying classifier for select datasets. I utilized the Multilayer Perceptron implementation in the WEKA framework on the Vehicle Dataset (taken from the UCI Machine Learning Database) to perform the aforementioned analysis. However, WEKA utilizes backpropagation as a technique to traverse the loss function and subsequently update the network's weights. If we treat updating the weights as an optimization problem, we can utilize a different branch of algorithms (randomized optimization algorithms) in place of backpropagation and that may result in better performance. To test this, I revisited my Vehicles Dataset to see if applying these other techniques would improve training and testing accuracy. Quick refresher on the Vehicle Dataset: comprised of 846 instances (each with 18 features) that are classified into four classes ("van", "opel", "saab", and "bus"), the instances represent silhouette outlines of various vehicles that have been scanned and uploaded (different axes of the silhouette outline comprise the different features of each instance), and pictorial data is often very good to feed into a neural network as it can use feature selection to create rules that are generalizable.

Evaluation Details and Graphs: I compared the learning curve constructed from Assignment 1 with new learning curves created using the ABAGAIL Framework (specifically by modifying the AbaloneTest.java file to utilize the Vehicle Dataset; for all randomized optimization algorithms, I ran each experiment five times and report the average of those trials [this applies to all graphs in this analysis]). Below are the learning curves:





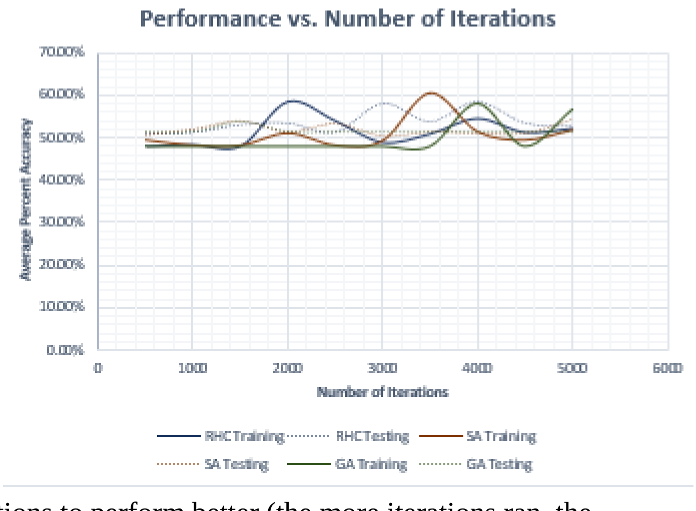




Analysis: First let's observe the learning curve generated using backpropagation on the Vehicle Dataset. We see as the number of training examples increases, the percent error gradually increases for the training set and decreases overall for the testing set. This behavior is what we would expect from a learner utilizing cross-validation as we feed more and more examples into the training set. When the number of examples is small, we have zero training error (as the learner is essentially memorizing all of the examples), but we have a high testing error (due to overfitting in the training stage and poor learner generalization). When the number of examples is large, we have a higher training error (cannot memorize all of the examples due to throttles on weight change such as learning rate and momentum), but a much lower testing error (as our learner was fed more examples and was able to generalize a lot better).

[Note: To shorted graph titles, I have utilized the following acronyms to represent the different learning algorithms – RHC (randomized hill climbing), SA (Simulated Annealing, and GA (Genetic Algorithms). These acronyms are used throughout this analysis]

Now, let's shift our attention to the learning curves generated from the ABAGAIL framework. From all three graphs, the training and testing errors oscillate greatly as the number of training examples increases but roughly stay the same after 200 instances. Near 700 instances, the training and testing errors appear to be relatively close together and we can roughly say they converge at around 45% error. Convergence is often a sign of a good learner when looking at learning curves (as it shows training error can be a good proxy for testing error), but generally, the error rate should be much lower than 50%. As all of the graphs have such a high error rate, our dataset may be at risk of having high bias amongst the examples which is preventing our learner from modeling the underlying function (compare those error convergence values to the convergence value from backpropagation [roughly 10% error]). Additionally, in each of the randomized optimization algorithms (RHC, SA, and GA), the training error appears to cross over the testing error but does not show signs of pure divergence. As each error reading was taken multiple times and averaged, we cannot attribute this pattern to random chance. From the dataset description, we knew that several instances of the "opel" and "saab" labels had high variance within their features in addition to those labels being very similar in terms of feature values. This could explain the pattern observed as higher training error is often attributed to high data variance.



From the learning curves, it was not clear which randomized optimization algorithm performed the best on the Vehicle Dataset. In hopes of further clarifying the playing field, I ran each algorithm for a various number of iterations and the results are outlined to the right.

An interesting pattern emerges. It appears that each randomized algorithm has its own "zone" of power. RHC performs the best at around 2000 iterations, SA performs the best at around 3500 iterations, and GA performs the best at around 4000 iterations. If the fitness function is quite large and has lots of local maxima, it would make sense why SA and GA take more iterations to perform better (the more iterations ran, the better SA is at finding the predicted global maxima by balancing exploration and choosing the best neighbor, and the better GA is at exploring the global function space). Near 5000 iterations, it appears as though RHC and SA converge and taper off to roughly 50% accuracy, whereas GA appears to be rising again (maybe indicative of arriving closer to the global maxima). Overall, none of the randomized optimization algorithms come close to the accuracy of using backpropagation to update the network's weights. This is most likely due to the complexity of the feature space leading to a complicated loss function that cannot be optimized using RHC, SA, and GA.

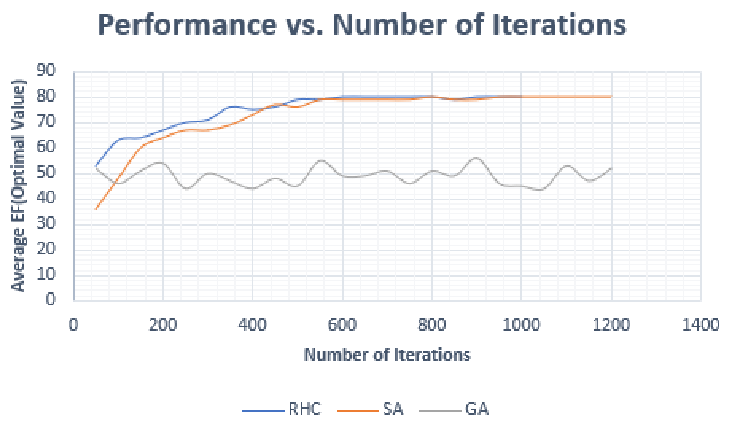
Optimization Problems

In order to further understand and break down the randomized optimization algorithms, I explored the tests (optimization problems) present in the ABAGAIL framework and selected three such tests that highlight scenarios where one randomized optimization algorithm is better than the other two for that test.

a. Randomized Hill Climbing: For RHC, I utilized the **CountOnesTest**. This optimization problem randomly creates a vector with N quantities where each quantity is assigned a value of 0 or 1. From simple mathematics, we observe that there can be 2^N such vectors to represent all of the possible combinations of 0's and 1's in an N-sized vector. The "goal" of the problem is to find the optimal vector where optimality is evaluated by summing up the number of 1's present in the vector (a vector with more 1's is more optimal than a vector with less 1's).

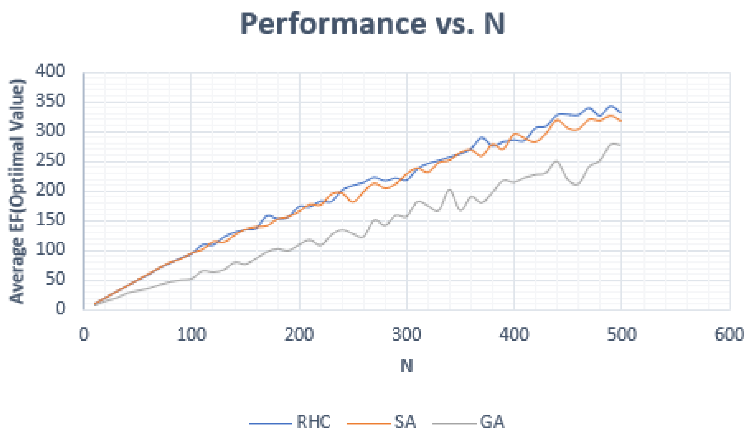
[Note: The ABAGAIL framework reports the strength of each of the learning algorithms as the outcome of passing in the optimal instance to an evaluation function (fitness function). For CountOnesTest, the evaluation function simply tallies the number of 1's. However, the evaluation function varies by the optimization problems. All graphs onwards will have "Average EF (Optimal Value)" as the performance metric where EF is the evaluation function.]

First to evaluate convergences times for each algorithm, I plotted performance versus the number of iterations and that is displayed below: (Here the N (size of vector) was held constant at 80)



It is clear to see that RHC and SA perform much better than GA as the number of iterations increases. Additionally, from 600 iterations onward, RHC and SA converge to the optimal value of 80. As number of iterations is one way to measure convergence times, it is logically sound to say RHC and SA have a convergence time of roughly 600 iterations, whereas the GA does not appear to converge in over 1200 iterations. Furthermore, we observe RHC consistently outperforming SA prior to 600 iterations. This is a byproduct of the evaluation function and the fitness landscape. As more 1's are indicative of a higher fitness and better optimality, it is clear to see how RHC would always select a vector from the neighborhood space with more 1's than the current vector. This reduces the optimization problem to simply "climbing" a smooth, linear line, which RHC excels in. SA, with enough iterations, will also excel in this setting as it will eventually converge to the predicted global maxima. However, it will struggle in the lower iterations as SA has a built-in "exploration" probability where it may sometimes select a bad successor vector in order to best explore the neighborhood. This may explain why RHC outperforms SA at lower iterations (as RHC always climbs toward the better fitness and SA may sidetrack to explore the local space), but eventually both converge to the optimal value. Furthermore, we can explain the behavior of the GA by remembering the breeding and mutation aspect of the algorithm. As we increase the number of iterations, we are normalizing the initial population variation into a somewhat constant value. From the graph, we see that the performance of the GA tends to oscillate between 40 and 50 which is roughly half of the optimized value of 80. By breeding vectors that are close to the optimized value with vectors that are really far away, we create populations with intermediate values. The background mutation rate occasionally changes some of the children vectors to deviate from the population norm which is visualized by the oscillations in the graph.

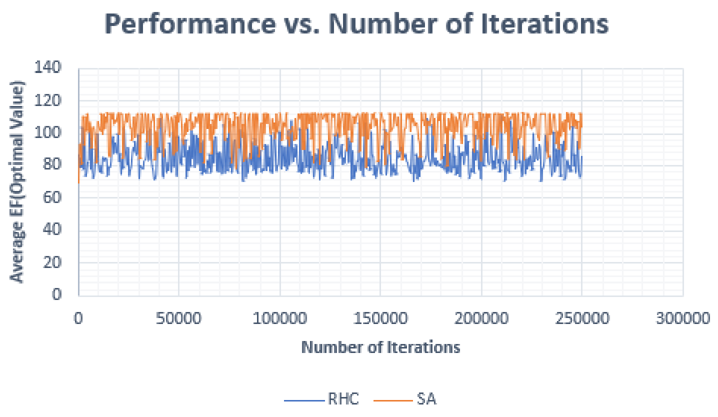
To test the impact of model complexity on each of the learning algorithms, I increased N (size of the vector) while keeping the number of iterations constant (at 600 iterations). Here are the results of that step:

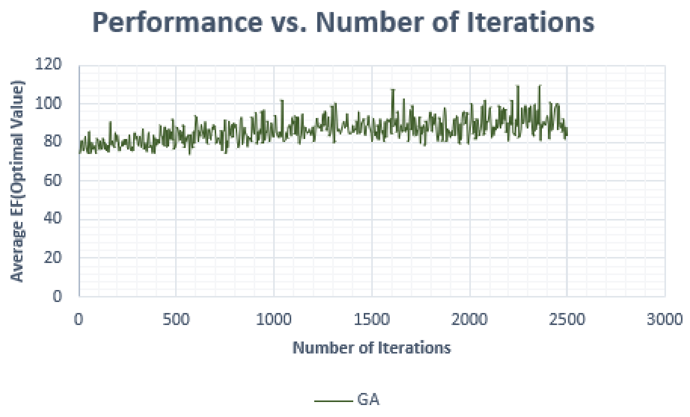


From the graph, it is apparent that RHC and SA outperform GA as N increases (basically as model complexity increases). Similar to the previous graph, RHC and SA perform quite similarly with RHC narrowly beating out SA for most values of N. Additionally, we observe that as the value of N increases, all of the algorithms do not reach the optimal. This is most likely due to the limitation put forth by restricting the number of iterations. But this does show how model complexity results in the exploration of a larger fitness function which may take longer time to fully explore and converge on the correct optimal value. Combining both of the graphs, we see that RHC outperforms the other two randomized optimization algorithms due to the linear nature of the evaluation function in addition to the lack of local maxima that could potentially trap a RHC algorithm.

b. Simulated Annealing: For SA, I utilized the **ContinuousPeaksTest**. This optimization problem is very similar to the CountOnesTest, but instead of being a linear fitness space, the ContinuousPeaks has a certain number of local maxima, any of which could contain the global maxima. The evaluation function for the ContinuousPeaks takes the maximum between the largest stretch of 0's and the largest stretch of 1's and modifies based on a T parameter (which reflects the minimum distance between evenly spaced peaks).

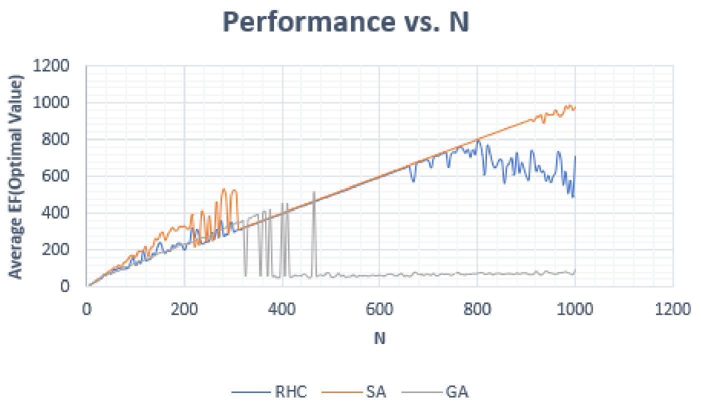
First to evaluate convergences times for each algorithm, I plotted performance versus the number of iterations and that is displayed below: (Here the N (size of vector) was held constant at 60 and T was held constant at 6; also I used less iterations for GA due to time it took to run each iteration)

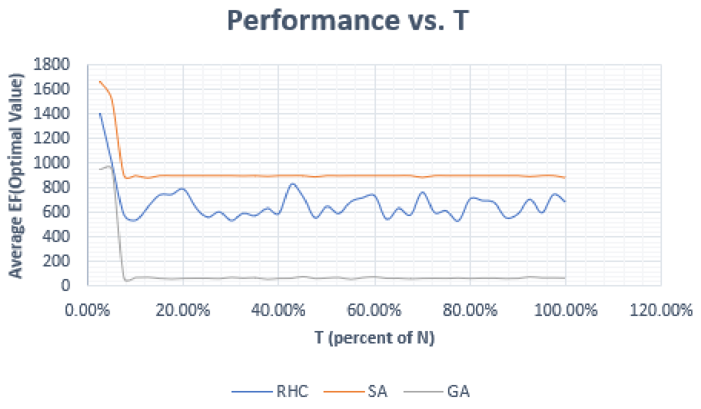




From the graphs above, it is clear to see that convergence was very difficult to establish for most of the randomized optimization algorithms. We can roughly say GA converges at around 1200 iterations, but the convergence times for RHC and SA are very difficult to determine (if I was to make an informed guess, I would say around 50000 iterations as the pattern seems to repeat after that number of iterations). One thing that is clear from the graphs is the superior performance of the SA algorithm. In terms of performance, it is distinctly above RHC and has slightly better performance than the GA. As the ContinuousPeaks problem has multiple configurations that could lead to the same optimal value (due to the symmetry present in the evaluation function), it makes sense that SA would outperform RHC as RHC may get stuck at a local maximum and not explore the rest of the fitness space. SA, on the other hand, has a probability of choosing a direction with lower immediate fitness in hopes of better exploring the fitness space. This optimization problem is perfect for that kind of exploration. GA, on the other hand, is also expected to perform well on this kind of problem due to the global nature of its search, and perhaps the number of iterations is holding it back from arriving at a better optimal value.

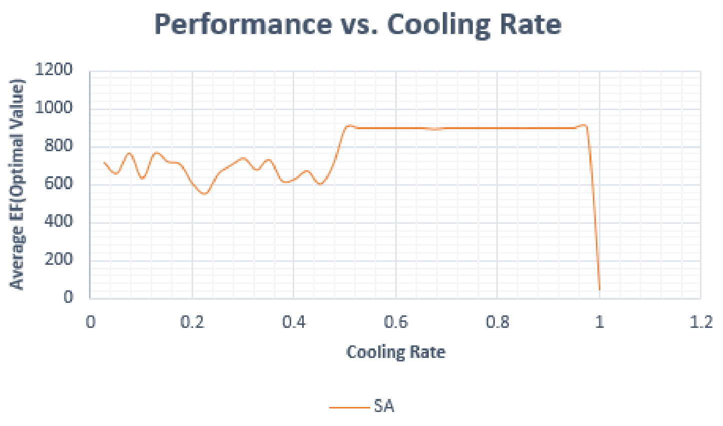
To test the impact of model complexity on each of the learning algorithms, I increased N (size of the vector; with T = N / 10) and T (the minimum separation between peaks; N = 900) while keeping the number of iterations constant (at 200000 iterations for RHC and SA and 1000 iterations for GA) in two separate experiments. Here are the results of that step:





First looking at performance versus N, we see SA outperform RHC and GA as N increases, which can be used as a proxy for model complexity. Overall, SA appears to return an optimal value that is just shy from the value of N itself, but if you observe closely, there are certain values of N that return performance values higher than N. This is not an error, but a consequence of the evaluation function which boosts performance of symmetric vectors (utilization of the T variable). RHC appears to perform similarly to SA until we reach a value of 800 for N, at which point the performance declines. It may just be at this value of N, the number of different combinations of peaks has too many local maxima for RHC to overcome. Far more interesting is that GA performance, which just rapidly drops after ~440 iterations and remains at a very low value from then on. This can mean one of many things: the GA algorithm requires more than 1000 iterations for that value of N, the distribution of the population is highly skewed right and only bad instances are mating with each other, or there are too many maxima in general from a global scope. Personally, I believe the number of iterations is what's holding back this algorithm. Shifting over to the performance versus T, we see SA outperform the other two algorithms (as expected due to the nature of the optimization problem). SA and GA appear to converge to specific values even as T increases whereas RHC continues to oscillate. This simply could mean that the number of peaks, after a certain value, does not heavily impact the optimal solution.

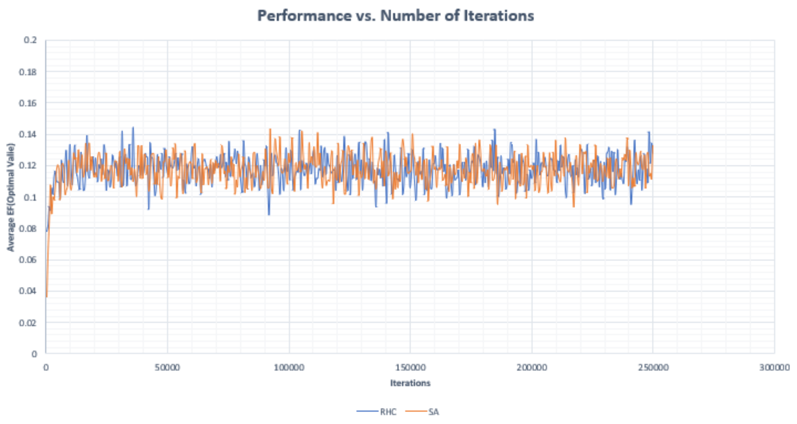
In hopes of optimizes the SA approach to the problem even more, I varied several of the hyperparameters while keeping the value of N at 900 and the value of T at N / 10. Here are the results of that experiment:

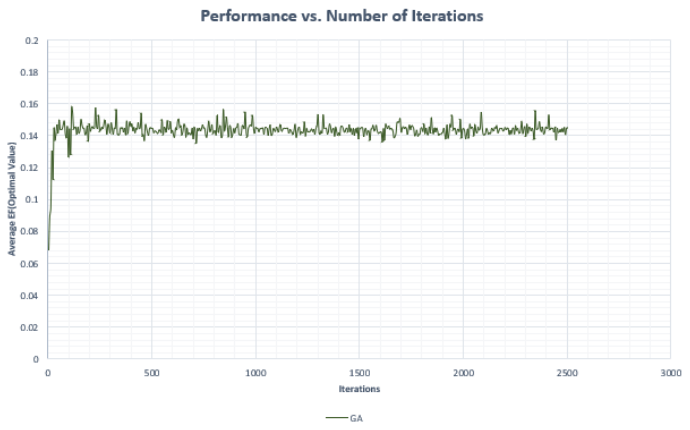


I varied both the cooling rate and the initial temperature for the SA algorithm. When I varied the initial temperature (from T = 10 to T = 1E21 where $T_{t+1} = T_t * 10$), I simply got the same answer for each run so I deemed it wasn't important to include the graph of a straight line. The average value of the optimal instance was roughly the same as N indicating initial temperature wasn't that important. What did matter was the cooling rate, which is displayed above. After cooling rate = 0.5, we see a convergence at around the predicted global maxima, but SA fails to reach that value when the cooling rate is less than 0.5. This simply means if we do not lower the temperature fast enough, the SA is still exploring the local space when it should be traveling towards the local maxima. Overall, SA performed very well on the ContinuousPeaks optimization problem. Perhaps GA could also boast similar performance values if given enough iterations. Exploring local space with many local maxima proves to be difficult for RHC, which aligns well with the theory of each randomized optimization algorithm.

c. Genetic Algorithms: For GA, I utilized the **TravelingSalesmanTest**. In this optimization problem, a salesman is tasked with the job of finding the shortest path between a set of N cities where he visits each city only one time. The distance between the cities may or may not be uniform and the evaluation function for this problem simply takes the inverse of the total distance travelled so that lower total distances are given higher values than higher total distances. As N (the number of cities) increases, the complexity of the problem increases and therefore traversing the fitness function becomes challenging.

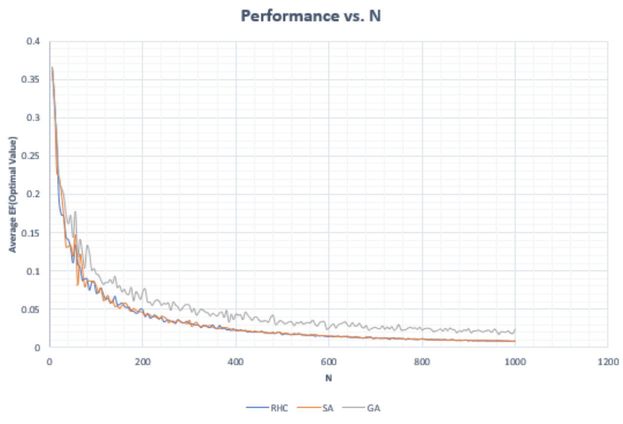
First to evaluate convergences times for each algorithm, I plotted performance versus the number of iterations and that is displayed below: (Here the N (size of vector) was held constant at 50 while the number of iterations change; also I used less iterations for GA due to time it took to run each iteration)





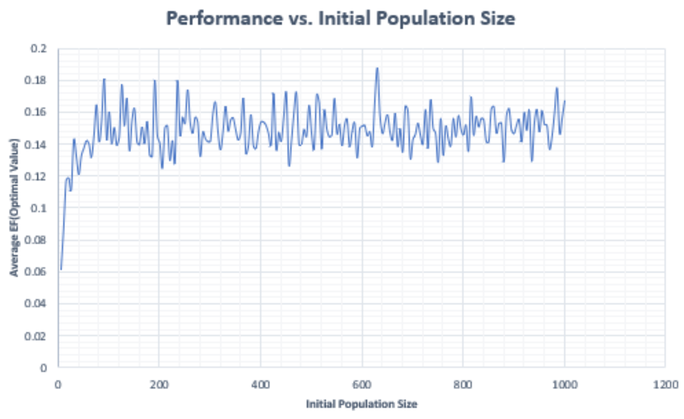
From the graphs above, the GA converges at around 200 iterations, while it is difficult to tell for SA and RHC algorithms (roughly around 10000 iterations). One thing that is clear from the graphs is the superior performance of the GA algorithm. In terms of performance, it is distinctly above both the RHC and SA algorithms (which appear to be tied to each other in terms of performance). One explanation for this phenomenon is the complexity of the fitness function. As there are so many routes possible between N cities, optimizing it via local search (which is what RHC and SA utilize) would take many, many iterations and it is also possible to be stuck in local maxima. GA, on the other hand, utilize global search which allows for faster exploration of the fitness space and attempt to improve on the best routes present in any given generation (specifically via crossing over and mutation further down the analysis). As a result, we clearly see better performance when using the GA versus the other optimization algorithms.

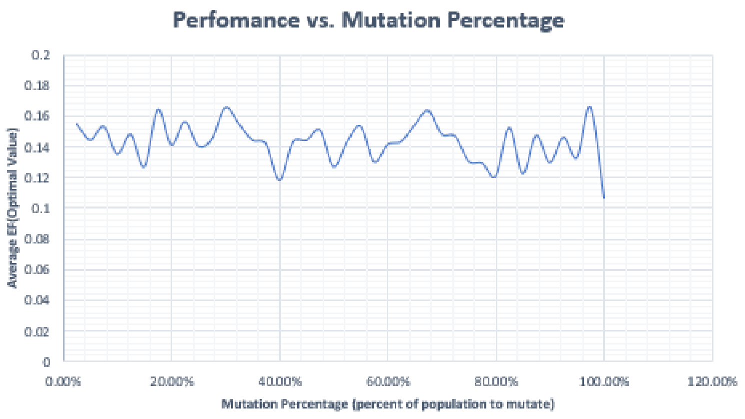
To test the impact of model complexity on each of the learning algorithms, I increased N (number of cities) while keeping the number of iterations constant (at 200000 iterations for RHC and SA and 1000 iterations for GA). Here are the results of that step:

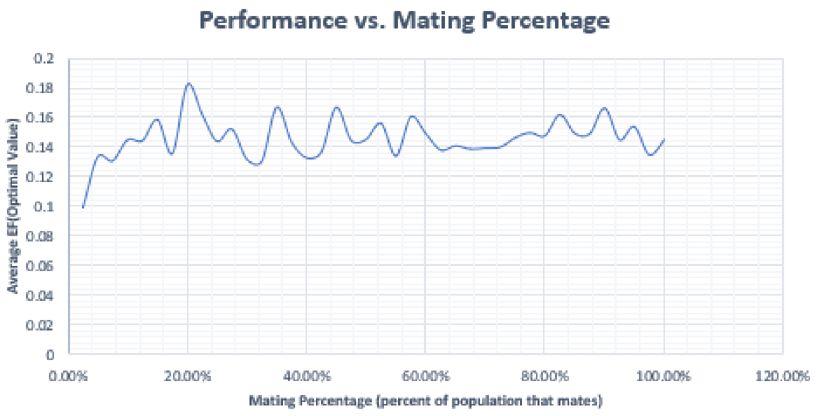


Looking at the performance as the number of cities increased, we see a similar downward trend across all three randomized optimization algorithms. However, it is still clear that GA outperform SA and RHC even as the number of cities increase. This of course is consistent with the notion that GA perform better as the fitness function grows in size and complexity due to the global nature of search used in GA's. One potential way to boost the performance of GA even as N increases is to increase the number of iterations (but the downside of doing that is the increase in processing time). However, it is not unreasonable to predict the still downward trend of performance as the problem complexity is increasing in non-polynomial time (Traveling Salesman problem is NP-Hard in theory) as the number of cities increases.

To specifically explore how changing the hyperparameters impact the GA and in hopes of boosting performance, I changed the initial size of the population, the amount of mating individuals, and the amount of mutations (N was held constant at a value of 50 and the number of iterations was held constant at 1000). Here are the outcomes of those experiments:



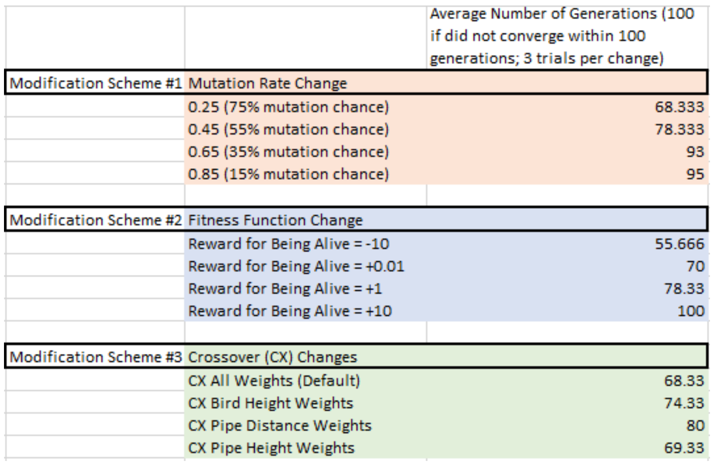




Looking first at performance versus initial population size, we see that performance appears to taper off after initial population size reaches roughly 180. This simply says that there is not a significant increase in performance after a certain number of "salesman" exploring the fitness space is reached. This could simply be due to a lack of iterations or the averaging of subsequent generations due to mating (where larger populations would have more mating pairs if the mating rate is held constant [which it was]). Then looking at the performance versus mutation percentage (percent of the population that underwent mutation), we see an oscillation of performance and no clear trend (although one can argue for a faint decrease in performance as mutation increases). This could be the result of a small population size (held at 200 for this portion) where the population is not large enough or the mutation function itself is not strong enough to impact the next generation that much. Lastly, looking at performance versus mating percentage, we see a trend of initial increase in performance, but then evening out with small oscillations (probably due to the background mutation rate [held constant at 20]). This trend can be explained by realizing what mating does to a population. As the number of mating pairs increase, the subsequent generation is more uniform than the previous one. Less mating pairs means parents with poor fitness can still pass their "genes" onto the next generation, whereas more mating pairs means offspring fitness will more or less be an averaging of the parent fitness with small changes due to mutation. From the graph and from theory, as more and more parents mate, we see a plateau in performance as the population becomes more and more uniform. One way to potentially keep increasing performance is to modify the mutation rate to scale with mating rate and population size. Overall, GA performed very well for the TravelingSalesmanTest as it was able to effectively explore the complex, global scope of the fitness function.

Flappy Bird – Neural Net + Genetic Algorithm

The final part of this analysis will focus on changes made to a Flappy Bird implementation (originally by Batchu Venkat Vishal and modified by Namkha Norsang). For a little background: Vishal's implementation uses a three layer neural net (3 inputs – 7 nodes in 1st hidden layer – 1 output) to determine when a bird should flap or not. The three inputs are the bird's current height, the distance to the next pipe, and the height of the next pipe. Additionally, instead of using gradient descent, the weights are modified using a genetic algorithm that spawns a population of birds that at aim to survive as long as possible and also going through the most pipes (defined as the bird's fitness). The next generation is determined via crossover of the first hidden layer weights, random mutation, and selective breeding. To explore the intricacies of the program and to further understand genetic algorithms, I made small adjustments to some of the hyper-parameters of the algorithm in order to see its influence on convergence time. Here convergence time was defined as the number of generations it took for the bird population to reach a score of 100 (if the population was not able to reach this score in 100 generations, then that population was given a convergence time of 100). The results of my changes are to the right:



		Average Number of Generations (100 if did not converge within 100 generations; 3 trials per change)
Modification Scheme #1	Mutation Rate Change	
	0.25 (75% mutation chance)	68.333
	0.45 (55% mutation chance)	78.333
	0.65 (35% mutation chance)	93
	0.85 (15% mutation chance)	95
Modification Scheme #2	Fitness Function Change	
	Reward for Being Alive = -10	55.666
	Reward for Being Alive = +0.01	70
	Reward for Being Alive = +1	78.33
	Reward for Being Alive = +10	100
Modification Scheme #3	Crossover (CX) Changes	
	CX All Weights (Default)	68.33
	CX Bird Height Weights	74.33
	CX Pipe Distance Weights	80
	CX Pipe Height Weights	69.33

Looking first at the change in mutation rate, I observed that the higher the chance of mutation on any given bird's neural network, the lower the number of generations the population took to converge (compare 75% mutation chance vs 15% mutation chance). [Note: I kept the fitness and crossover scheme constant at the default setup as I changed the mutation chance.] One possible explanation would be that as the number of mutations in the population increases, there is more likely for any given neural network to arrive at a set of weights that evaluate to local maxima on the fitness function. As the goal is to optimize one's value on the fitness function, more mutations allows for faster search (faster exploration) of the function and better convergence values. Lower mutation values would be indicative of slower changes to the population composition (thus impeding growth if population is poor overall). Moving on to the change in fitness function, the default implementation of the game rewards "alive" birds by adding a value of +1 to their fitness in addition to rewarding the bird with +25 for each pipe passed. I altered the amount of reward each bird would receive for simply staying alive and the results were fascinating. The smaller the reward was for staying alive, the better the population was able to perform, with a negative reward population outperforming any positive reward population. As we are only selecting birds with positive fitness values to breed, it would make sense that only birds able to pass through pipes (receiving +25 and overcoming the -10 for staying alive) would be selected for and therefore the network would optimize passing through pipes as quickly as possible. This kind of bleeds into reinforcement learning where you punish the agent for playing the game, but the reward for winning the game is so large that the agent doesn't immediately die every time and instead learns to balance negative costs with the large positive reward. Lastly, I modified by the crossover scheme to see which of the three inputs has the greatest impact on the population neural network models. The default crossover strategy performed the best, with only crossing over the weights of the "height of nearest pipe" input coming in second. Crossing over only the "distance to the pipe" weights performed the worst, suggesting perhaps that is not an impactful input to have in the network as it doesn't seem to reduce the generation time till convergence. Although not included in the chart, I did attempt to include the best changes of all the hyper-parameters into one model (mutation rate @ 75%, negative reward, and crossover of just the "height of nearest pipe") and unfortunately, that population's performance was not any better than each of the changes applied separately, which indicates maybe each modification scheme should be given weights which are also adjusted to arrive at the best linear or non-linear combination of the schemes.

Conclusion

All in all, I have a very enjoyable experience with this assignment and it definitely deepened my understanding of randomized optimization algorithms. From Part 1, I was able to see how weights to neural networks can be updated using different strategies (and not relying on backpropagation all the time). Even though, for my specific dataset, backpropagation ended up outperforming randomized hill climbing, simulated annealing, and genetic algorithms, that is not enough to say that these techniques couldn't be applied in other settings. From what I was able to gather, my specific dataset's evaluation function was far too complex and costly to explore using randomized optimization algorithms (perhaps many, many local maxima and large function domain). Moving on to Part 2, I was able to identify specific scenarios in which one randomized optimization algorithm outperformed the other two, and by doing so, I was able to gain a better understanding of the algorithm's preference biases. For RHC, the algorithm excelled when the fitness function was relatively easy to evaluate with few (or no) local maxima that were very close to the global maxima. It would always select a better neighbor until it converged so the final fitness value was the highest (CountOnesTest). For SA, the algorithm excelled when the fitness function was easy to evaluate even if it had many peaks of local maxima. As there is a built-in probability of choosing a poor neighbor ("exploration"), it was able to outperform RHC by avoiding getting stuck in local maxima (ContinuousPeaks). For GA, the algorithm excelled when the fitness function as large and a more global search was needed to quickly explore the space (TravelingSalesman) via mutation and crossover of parents. Lastly, for Flappy Bird, I used the knowledge acquired from the previous parts to further explore GA to see how tuning specific hyperparameters impacted the development of a neural network (and even diving into a bit of reinforcement learning). Overall, I learned a great amount from this assignment and have already begun to meddle with other applications of randomized optimization algorithms.