Comparison of Randomized Optimization Methods

Jenny Hung

[jehung@gatech.edu](mailto:jehung@gatech.edu) (alias: jhung34@gatech.edu)

1. Optimization Algorithms

The randomized optimization algorithms under comparison are:

* Randomized Hill-Climbing (RHC): locates local optima by moving towards more optimal neighbors until it reaches a peak. With random restarts, RHC randomizes its starting position to locate other local optima, and selects the value with the highest value as the global optimum. RHC was performed using RandomizedHillClimbing() in ABAGAIL on Java.
* Genetic Algorithm (GA): inspired by biology in which the population evolves by iteratively mating and mutating parts to crossover the best traits and to eliminate irrelevant traits. A significant disadvantage of GA is that it does not handle a large hypothesis space, which is dictated exponentially by the number of attributes. SA was performed using StandardGeneticAlgorithm() in ABAGAIL on Java
* Simulated annealing (SA): originates from metallurgy where the ductility in metals are improved by heating to a higher temperature (below melting, above the annealing temperature where residual structural stresses are relieved) and then slowly cooled to maintain its structure. The algorithm, a function of initial temperature and cooling rate, strikes a balance between exploring new points and exploiting nearby neighbors in search of local optima. Initially, at high temperatures, the algorithm explores by randomly seeking new points and as it cools, it proceeds to evaluate neighbors for local peaks. SA was performed using SimulatedAnnealing() in ABAGAIL on Java. Simulated annealing was found to be sensitive to the initial temperature and the perturbation functions used. The values were adapted by try and error. It has been found that not the absolute values for initial temperature and the amount of perturbation were significant but their ratio.
* MIMIC: MIMIC algorithm, as opposed to most optimization algorithms, “remembers” previous iterations and uses probability densities to build structure of the solution space and find optima. MIMIC was performed using MIMIC in ABAGAIL on Java.

Because all algorithms are randomly initialized, running an algorithm once on the data and try to derive conclusions about the nature of the algorithm is not suited. Instead all algorithms were run several times and the mean was used for comparison. The specific iterations are given in the results.

1. Optimization problems

I have chosen 3 diﬀerent optimization problems to demonstrate the various strengths of each algorithm. I picked the continuous peak problem, the Travel-Salesman-Problem (TSP) and the Flip Flop problem. Each of them and the motivation behind them will be explained in the following. Please note, that the first two functions are minimization problems, which can be easily transformed to maximization problems by changing the sign of the cost or the fitness function, so this is not a restriction.

2.1 Continuous Peaks

The Continuous Peaks problem is a variation based on the four-peaks problem, which was originally presented in [Baluja and Caruana, 1995][[1]](#footnote-1). The original four-peaks problem is defined as, given an input vector X, which is composed of N binary elements, maximize the following: Fitness is maximized if a string is able to get both the REWARD of 100 and if the length of one of head(1,X) or tail(0,X) is as large as possible. The four peaks problems also have two suboptimal local optima with fitnesses of N (independent of T). One of these is at tail(0,X)=N, head(1,X)=0 and the other is at tail(0,X)=0, head(1,X)=N. Hill-climbing will quickly get trapped in these local optima. For hill-climbing to work well here, it must repeatedly make “correct” decisions while searching large plateaus; this is extremely unlikely in practice. By increasing T, the basins of attraction surrounding the inferior local optima increase in size exponentially while the basins around the global optima decrease at the same rate.

In the Continuous Peaks version, rather than forcing 0’s and 1’s to be at opposite ends of the solution string, they are allowed to form anywhere in the string. For this problem, a reward is given when there are greater than T contiguous bits set to 0, and greater than T contiguous bits set to 1.

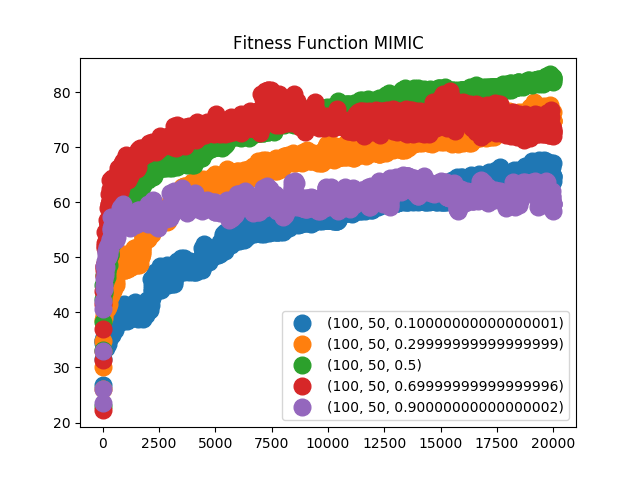
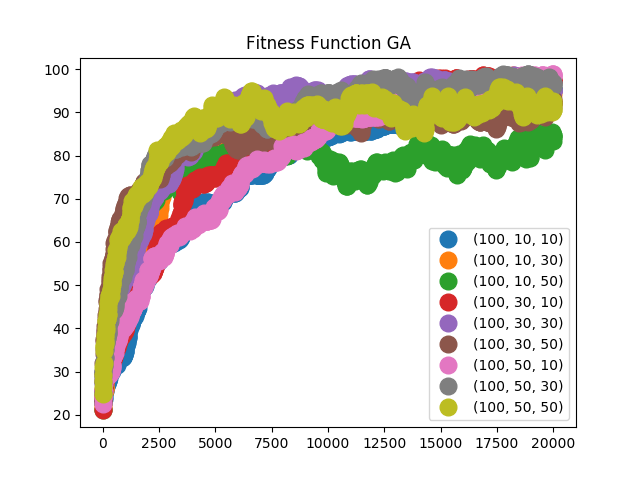
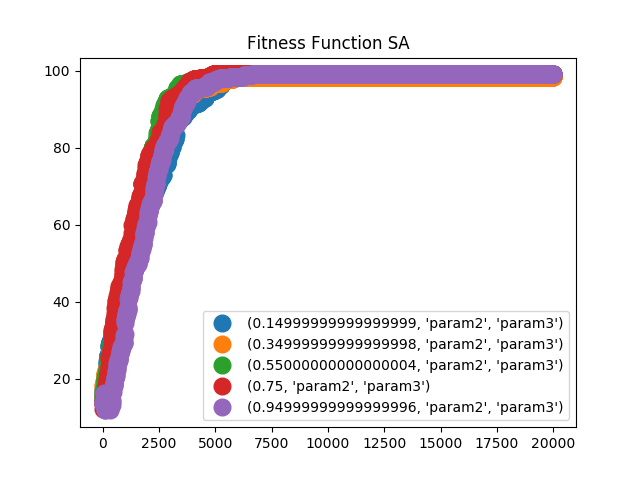
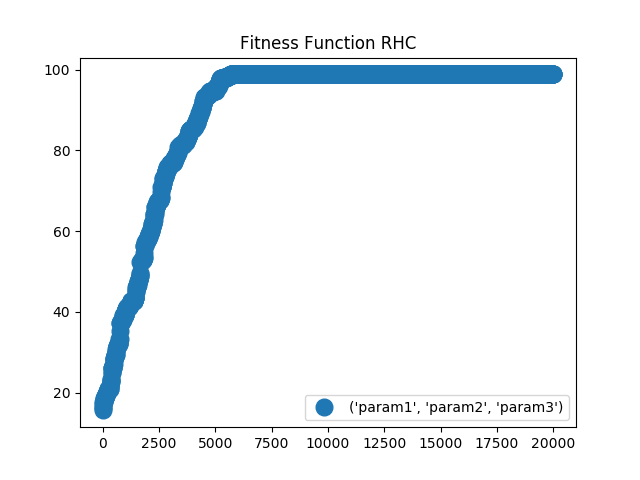
In solving for the Continuous Peaks problem, the existing cost function of the JAVA library, ABAGAIL, was used. Each algorithm was run using 20000 iterations to observe how quickly the algorithms converge on the optima. We observed that different algorithms have wildly different time complexity. For example, MIMIC is highly computationally intensive, primarily because the additional time it takes to build structure while learning enables the algorithm to learn with fewer iterations. Each of these tests were run 5 trials and the average function value is reported. The fitness score and running time were reported every 10 iterations.

The default parameters in ABAGAIL were as follows:

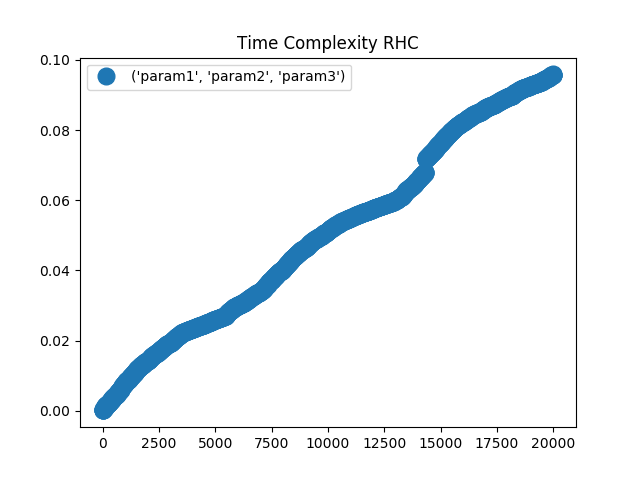
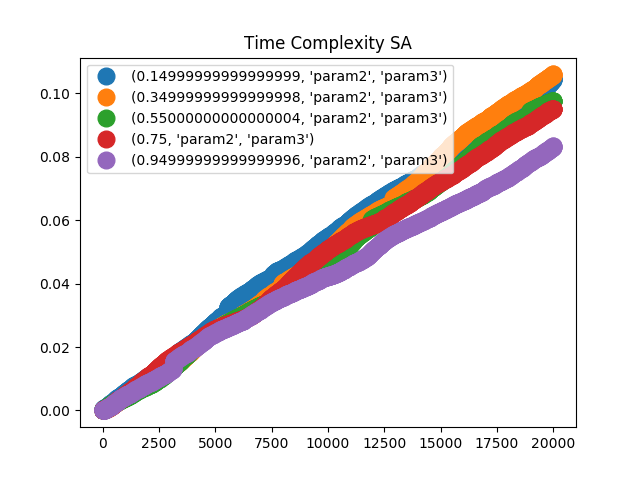
* Simulated Annealing: temperature = 1E10, cooling rate = [0.15, 0.35, 0.55, 0.75, 0.95]
* Genetic algorithm: population size = 100, toMate (varying value) = [50, 30, 10], toMutate (varying value) = [50, 30, 10]
* MIMIC: sample size = 100, keep = 50, m (varying value) = [0.1, 0.3, 0.5, 0,7, 0.9]

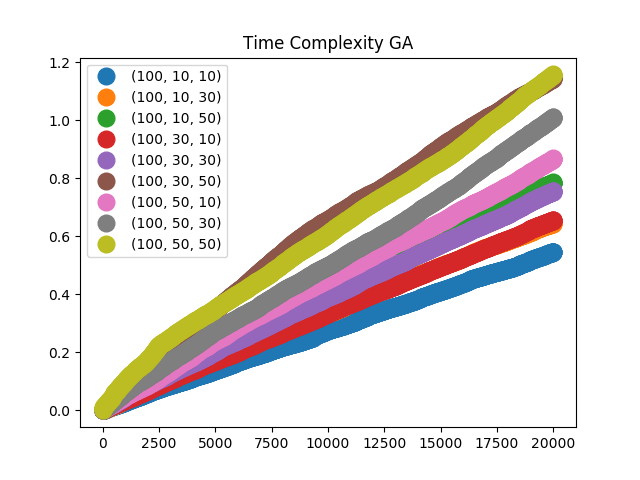
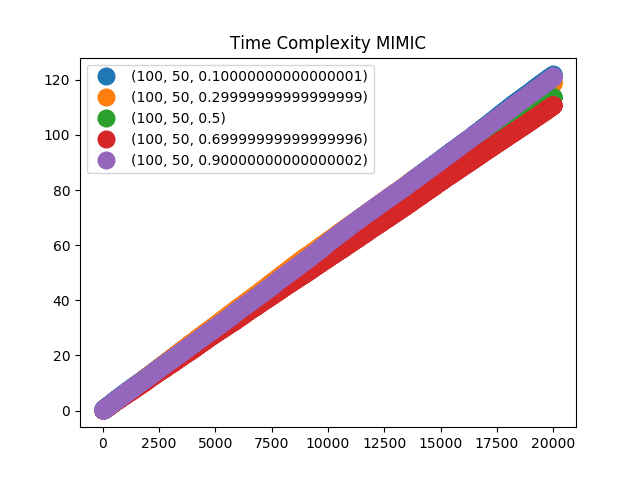
This problem set contains many local optima in a 1D space, similar to the example in lectures about determining the elevation of many peaks. Although this problem set was chosen for its simplicity, it highlights differences between random optimization algorithms well and applies to other examples like topography and optimization of surfaces.

The outcomes of fitness functions were plotted against number of iterations below.



Running time were plotted against number of iterations below.

Clearly SA and RHC outperform the other algorithms, both achieved nearly no error – SA had achieved this with the least number of iterations (approx. 3000), whereas RHC achieved this at approx. 5000 iterations. Further, GA also performs better than MIMIC, which gives the lowest fitness across all trials. MIMIC is highly computationally intensive – but what is interesting about MIMIC is that, it tends to achieve higher fitness much earlier on, compared to all other algorithms considered, but in this problem, MIMIC tends to converge after that, and only make very small improvement to the fitness after that, while all other algorithms continued to improve.

Based on the above, we clearly observed the artifact of MIMIC taking the time to build the probability structure, which enables the algorithm to learn faster, given the additional information (that does not exist in other algorithms).

Similarly, SA and RHC have very similar wall time when it comes to algorithm execution. Depending on the toMate and toMutate parameters, GA’s running time varies – it could be in the range of SA/RHC but it did progressive get worse when the number of computations gets larger given the parameters, which is consistent with our understanding. Of all the algorithms, MIMIC took the longest and its running time in terms of wall time were an order of magnitude larger compared to others. MIMIC is highly computationally intensive, primarily because the additional time it takes to build structure while learning enables the algorithm to learn with fewer iterations.

* 1. TSP

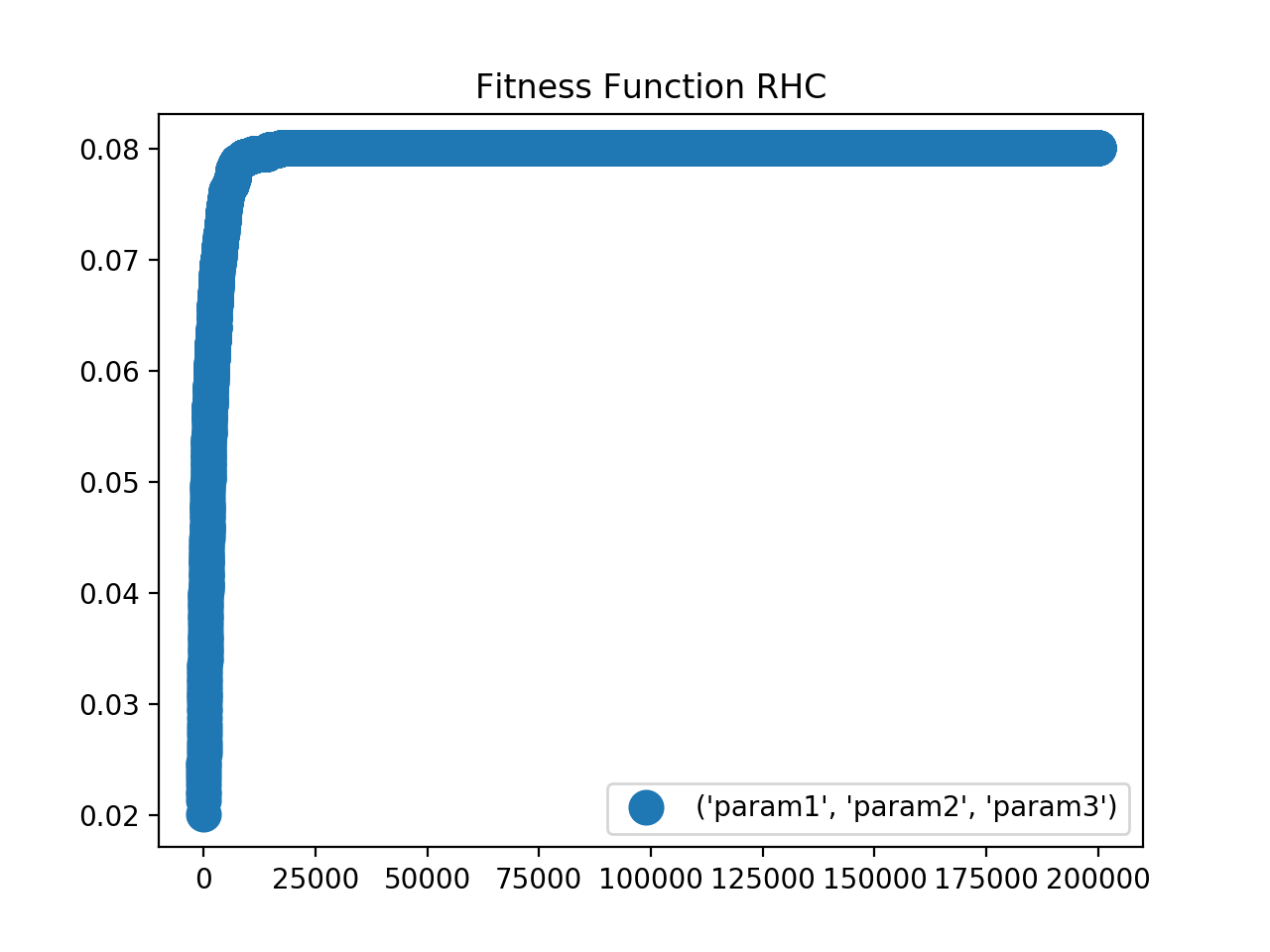
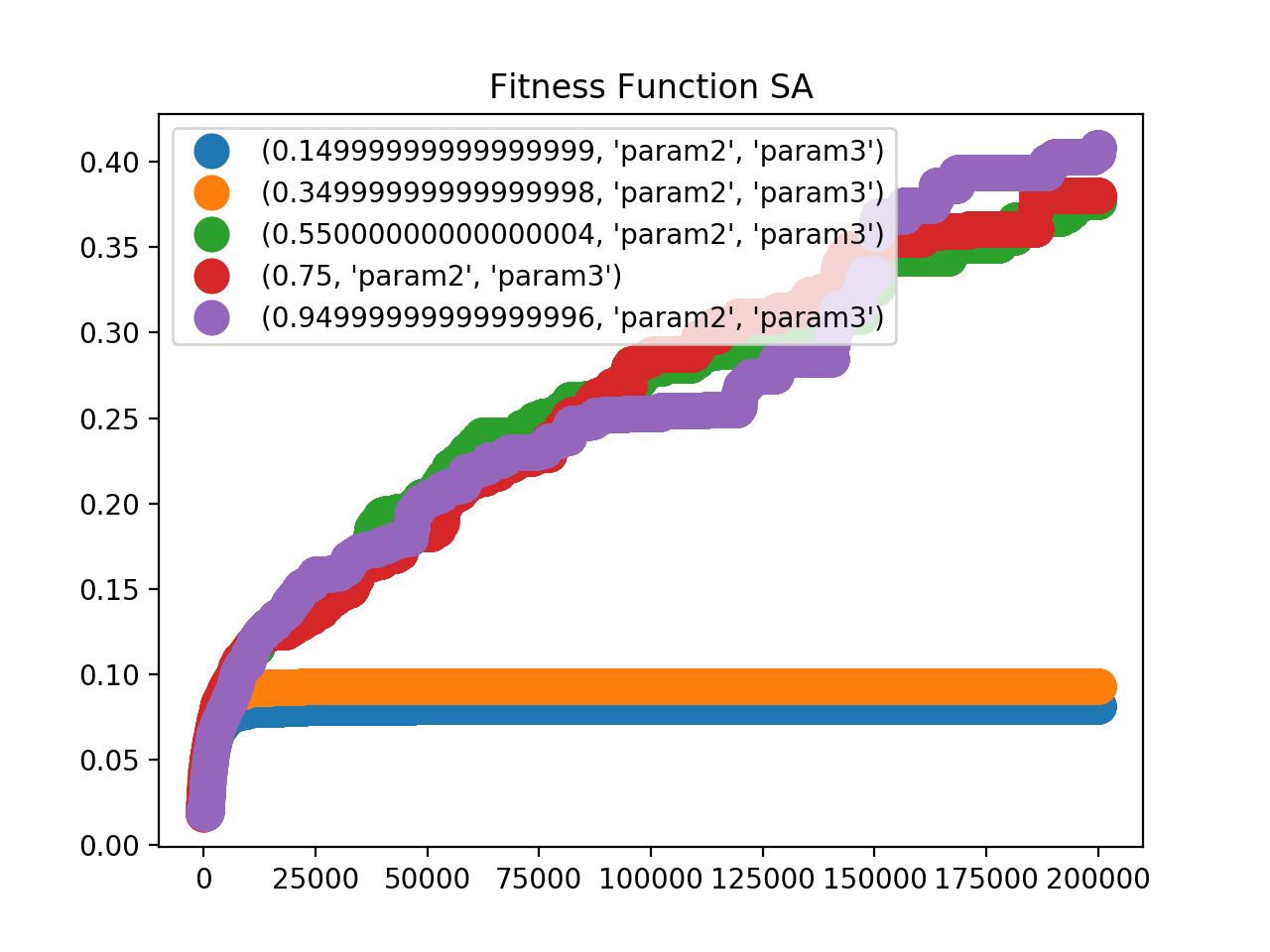
The well-known Traveling Salesman Problem is proven to be NP-hard. The goal is to find the shortest round-trip between N cities while visiting each city just once. TSP problems also occur in everyday life. In business, planning optimal routes between destinations cities is a crucial task for logistic companies such as UPS and FedEx. Other applications may include factory scheduling, wiring looms and circuit board drilling. Given n cities and picking an arbitrary city to start from, there exists possibilities for a round-trip. The factor arises from the fact, that we don’t care about the direction in which we are traveling. In general, apart from the fact that the number of possible routes grow exponentially with respect to the number of cities, the TSP problem is not ’discreet continuous’[2](#page2). That means that small changes in the configuration of the points can lead to completely diﬀerent optimal routes. That makes it hard to define a good search algorithm, and a greedy algorithm is very unlikely to find the optimal route.

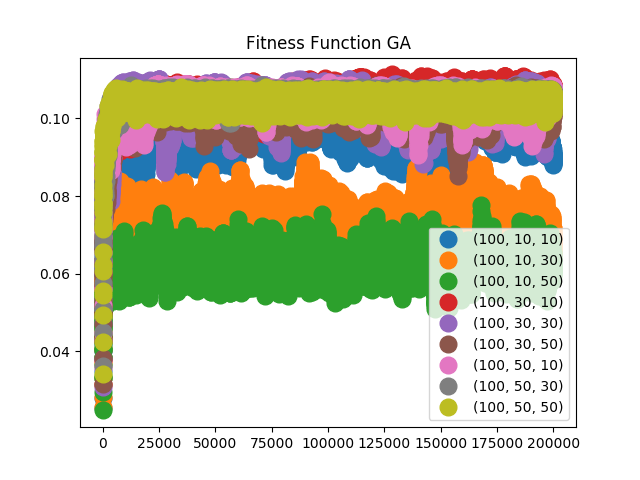
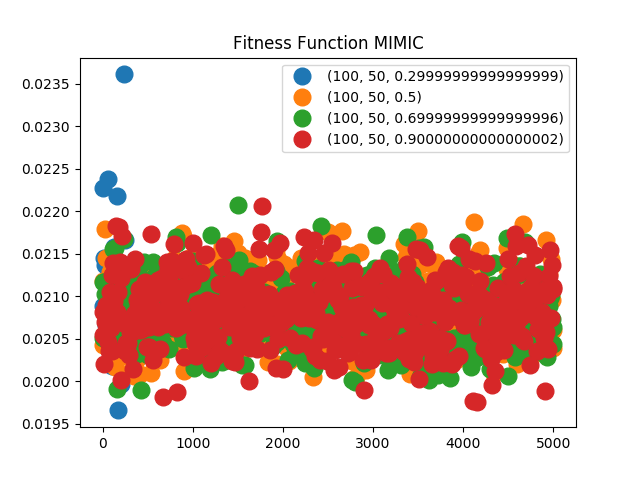
In solving the Traveling Salesman’s problem, the existing cost function of the JAVA library, ABAGAIL, was used. Each algorithm was run using 20000 iterations to observe how quickly the algorithms converge on the optima. Each of these tests were run 5 trials and the average function value is reported. The fitness score and running time were reported every 10 iterations.

The default parameters in ABAGAIL were as follows:

* Simulated Annealing: temperature = 1E10, cooling rate = [0.15, 0.35, 0.55, 0.75, 0.95]
* Genetic algorithm: population size = 100, toMate (varying value) = [50, 30, 10], toMutate (varying value) = [50, 30, 10]
* MIMIC: sample size = 100, keep = 50, m (varying value) = [0.5, 0.9]

The outcomes of fitness functions were plotted against number of iterations below.

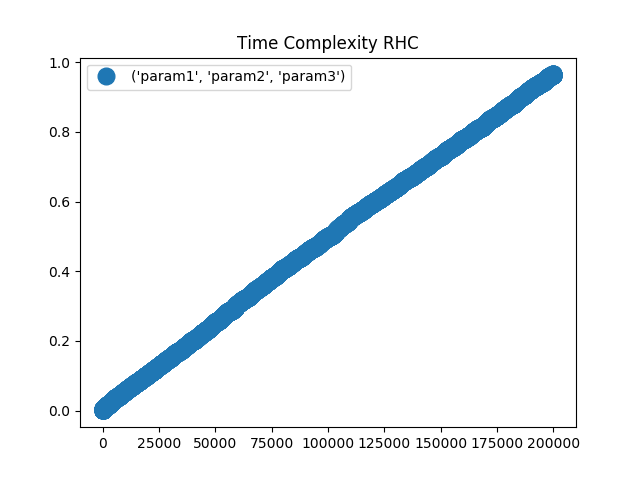
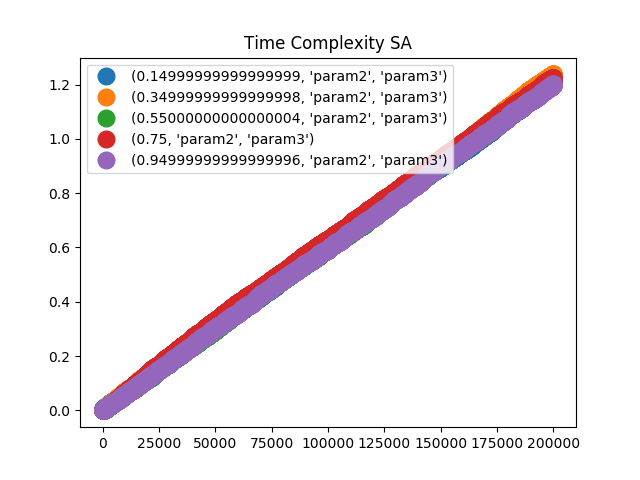
 

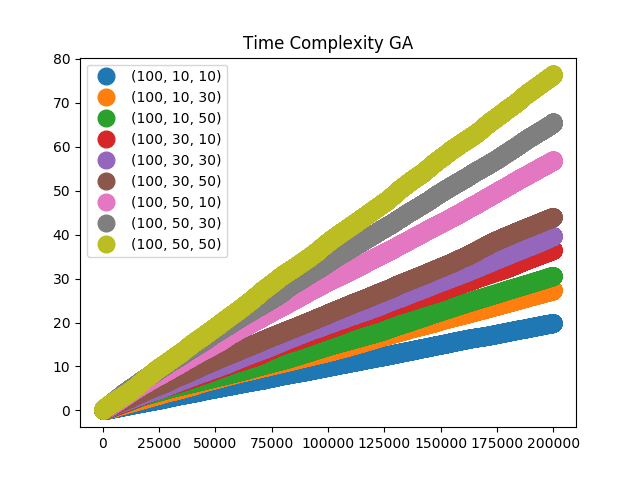
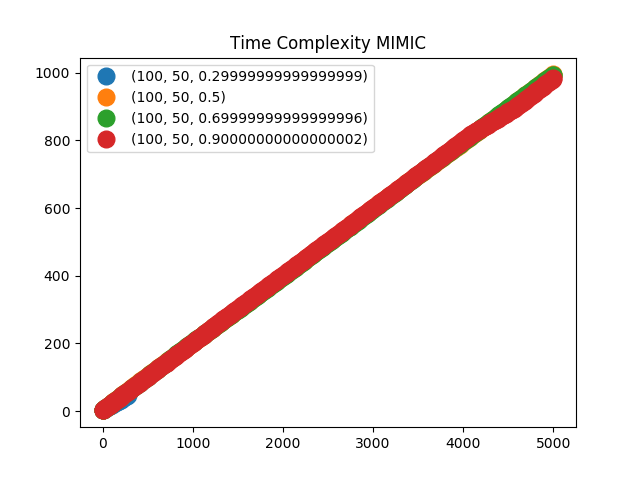
In the travelling salesperson problem, we seek to minimize the travel distance between many locations is minimized (assuming each location is visited once). This has useful applications to the travel, transportation, shipping and logistics in which destinations rarely lie along a line and some extent of detours is required to visit all locations.

We observed that interpolation among them may fool the algorithms. It should be noted, that there exists TSP-optimized variants that implement more sophisticated cross-over and mutation operations. In my implementation, a mutation might not change the route at all. RHC and SA perform equally well. Both algorithms perform pretty good as they explore a huge search space and are likely to find good routes. In my implementation, although RHC performs with roughly equal time complexity compared to SA, given the time, it has lower fitness. This is because RHC expands its search radius, it stays in its current neighborhood and searches within it. The restart from a diﬀerent random point may change this. SA evaluates not only points in the current neighborhood but also other points. That might be why it finds a “good” route faster than RHC.

Clearly SA outperforms the other algorithms with the highest fitness (maximizing the inverse of distance travelled, namely, minimizing distances travelled). SA achieved this while running at least 200000 iterations (averaged over 5 trials), and we were to let the iteration numbers increase, it is likely that the fitness will continue to increase as well. The worst performing algorithm for this problem is MIMIC. It took an extremely long time while only able to complete 5000 iterations, with very low fitness score. MIMIC does not perform well here because as a two-dimensional NP-hard problem, the hypothesis space is simply too complex. MIMIC had to go through this very complex and large hypothesis space to build the structure of joint probability distributions, and the results were in fact dismal.

Running time were plotted against number of iterations below.

In terms of time complexity for algorithms, RHC and SA have very similar time complexity per iterations. We also observed that GA’s run time were indeed more than a order of magnitude higher than both the RHC and SA. The same could be said about MIMIC. In fact, when it comes to MIMIC, the time complexity had exploded when compared to the other algorithms.

When we view the fitness score and the time complexity together, we see that the best algorithm to solve the Traveling Salesperson problem, given our specific parameters, is indeed SA (Simulated Annealing).

2.3 Flip Flops

Flip-flops can be either simple (transparent or asynchronous) or clocked (synchronous). The simple ones are commonly described as *latches*,[[1]](https://en.wikipedia.org/wiki/Flip-flop_(electronics)#cite_note-pedroni-1) while the clocked ones are described as *flip-flops*.[[2]](https://en.wikipedia.org/wiki/Flip-flop_(electronics)#cite_note-ee42-2)

Simple flip-flops can be built around a single pair of cross-coupled inverting elements: [vacuum tubes](https://en.wikipedia.org/wiki/Vacuum_tube), [bipolar transistors](https://en.wikipedia.org/wiki/Bipolar_transistor), [field effect transistors](https://en.wikipedia.org/wiki/Field_effect_transistor), [inverters](https://en.wikipedia.org/wiki/Inverter_(logic_gate)), and inverting [logic gates](https://en.wikipedia.org/wiki/Logic_gate) have all been used in practical circuits.

Clocked devices are specially designed for synchronous systems; such devices ignore their inputs except at the transition of a dedicated clock signal (known as clocking, pulsing, or strobing). Clocking causes the flip-flop either to change or to retain its output signal based upon the values of the input signals at the transition. Some flip-flops change output on the rising [edge](https://en.wikipedia.org/wiki/Signal_edge) of the clock, others on the falling edge.

Since the elementary amplifying stages are inverting, two stages can be connected in succession (as a cascade) to form the needed non-inverting amplifier. In this configuration, each amplifier may be considered as an active inverting feedback network for the other inverting amplifier. Thus the two stages are connected in a non-inverting loop although the circuit diagram is usually drawn as a symmetric cross-coupled pair (both the [drawings](https://en.wikipedia.org/wiki/File:Eccles-Jordan_trigger_circuit_flip-flip_drawings.png) are initially introduced in the Eccles–Jordan patent).

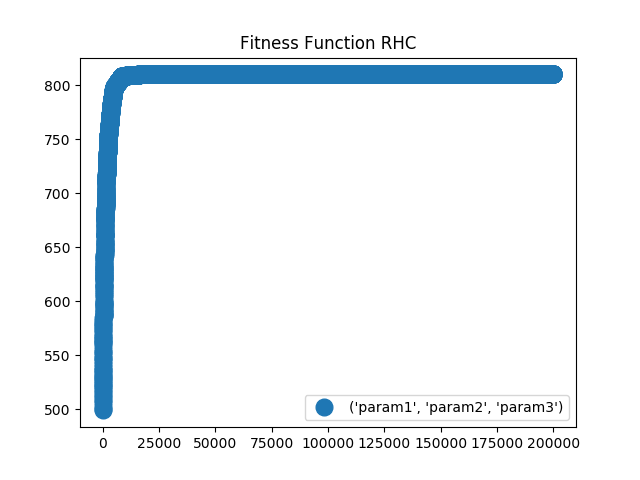
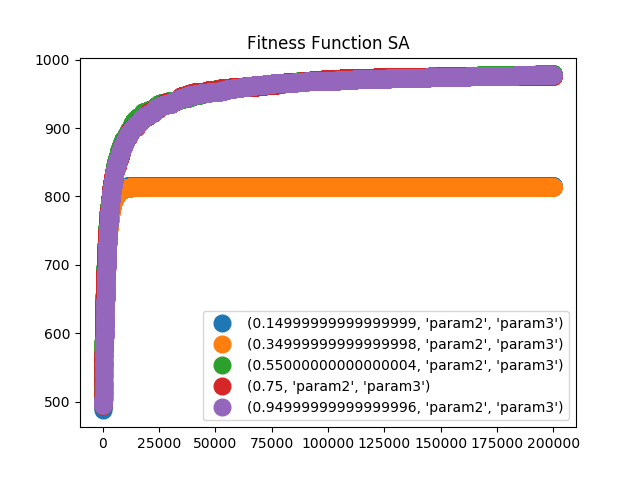
Each of these tests were run 5 trials and the average function value is reported. The fitness score and running time were reported every 10 iterations.

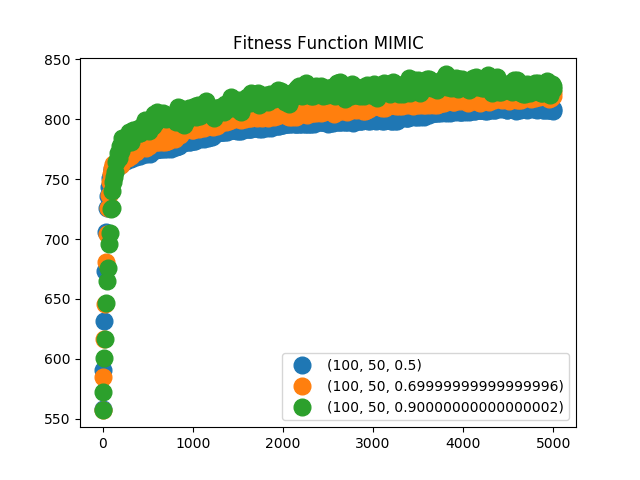
In solving the Traveling Salesman’s problem, the existing cost function of the JAVA library, ABAGAIL, was used. Each algorithm was run using 20000 iterations to observe how quickly the algorithms converge on the optima.

The default parameters in ABAGAIL were as follows:

* Simulated Annealing: temperature = 1E10, cooling rate = [0.15, 0.35, 0.55, 0.75, 0.95]
* Genetic algorithm: population size = 100, toMate (varying value) = [50, 30, 10], toMutate (varying value) = [50, 30, 10]
* MIMIC: sample size = 100, keep = 50, m (varying value) = [0.1, 0.3, 0.5, 0,7, 0.9]

The outcomes of fitness functions were plotted against number of iterations below.

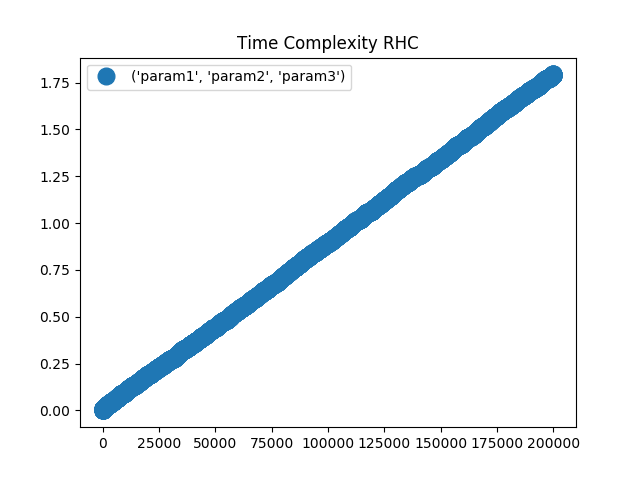
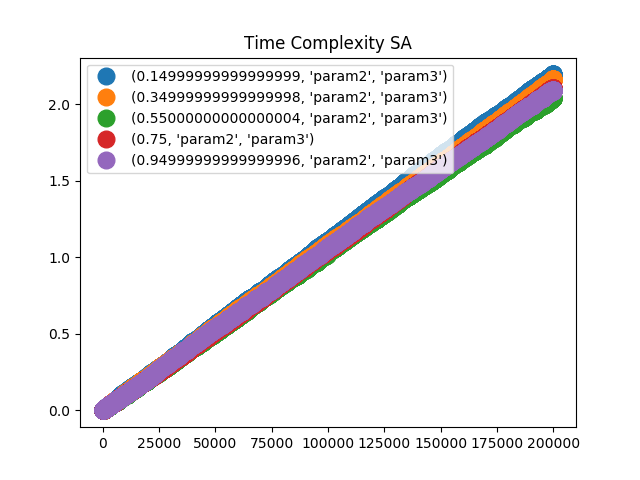
 

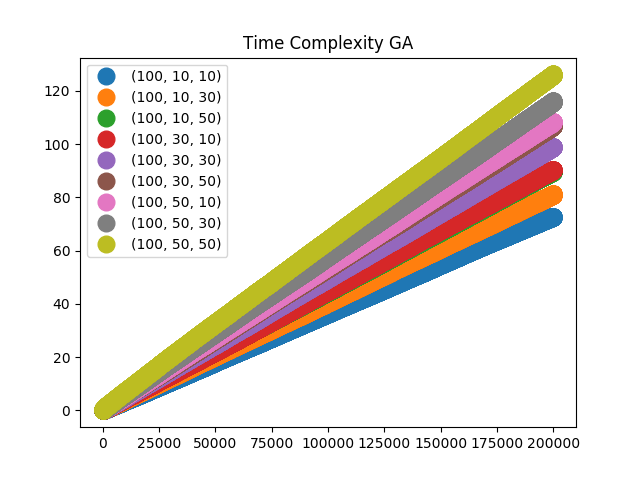
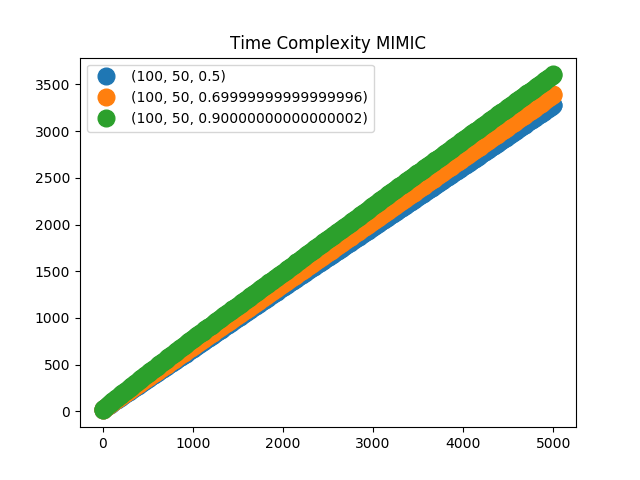
 

In the Flip Flop problem, clearly GA had the lowest fitness score – it achieved a reasonable fitness score early on (less than 10000 iterations) but has fitness score fluctuates up and down after that. Contrary to GA, RHC and MIMIC has very similar fitness scores. However, the biggest difference between RHC and MIMIC lies in the performance in terms of time complexity, which will be discussed in the following paragraph.

The star algorithm for the Flip Flop problem is SA (Simulated Annealing), which has the highest fitness scores of all algorithms considered.

Running time were plotted against number of iterations below.

In terms of time complexity for algorithms, RHC and SA have very similar time complexity per iterations – but RHC still complete 200000 iterations faster than SA. We also observed that GA’s run time were indeed a order of magnitude higher than both the RHC and SA. The same could be said about MIMIC. In fact, when it comes to MIMIC, the time complexity had exploded when compared to the other algorithms, and we could only complete 5000 iterations for this analysis. MIMIC is highly computationally intensive – but what is interesting about MIMIC is that, it tends to achieve higher fitness much earlier on, compared to all other algorithms considered, but in this problem, MIMIC tends to converge after that, and only make very small improvement to the fitness after that, while all other algorithms continued to improve.

Based on the above, we clearly observed the artifact of MIMIC taking the time to build the probability structure, which enables the algorithm to learn faster, given the additional information (that does not exist in other algorithms).

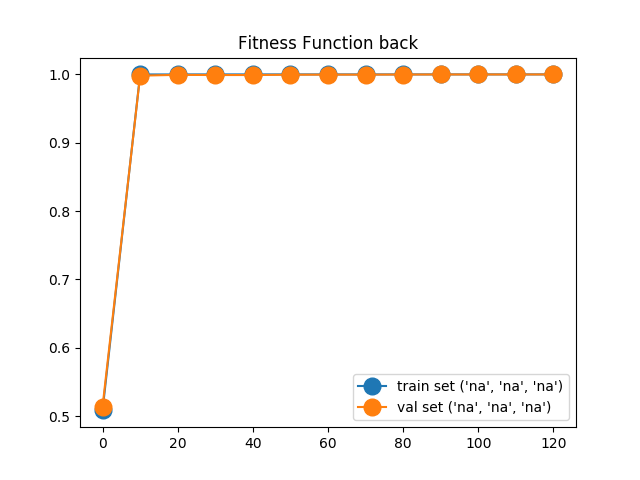
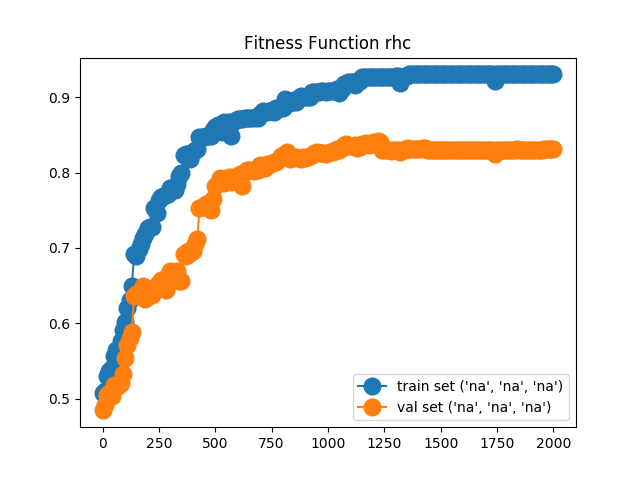
When we view the fitness score and the time complexity together, we see that the best algorithm to solve the Traveling Salesperson problem, given our specific parameters, is indeed SA (Simulated Annealing).

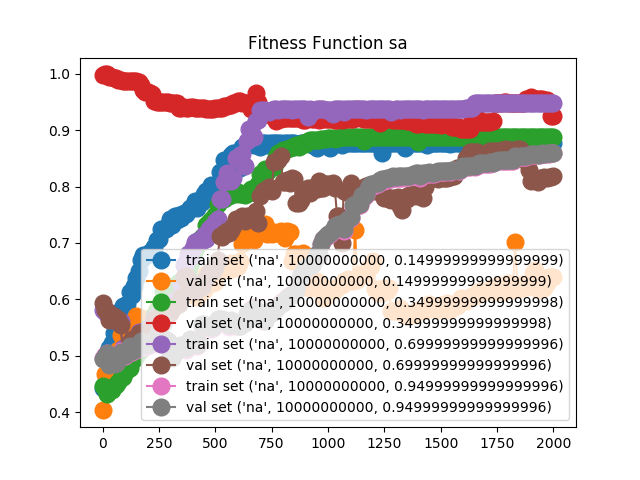
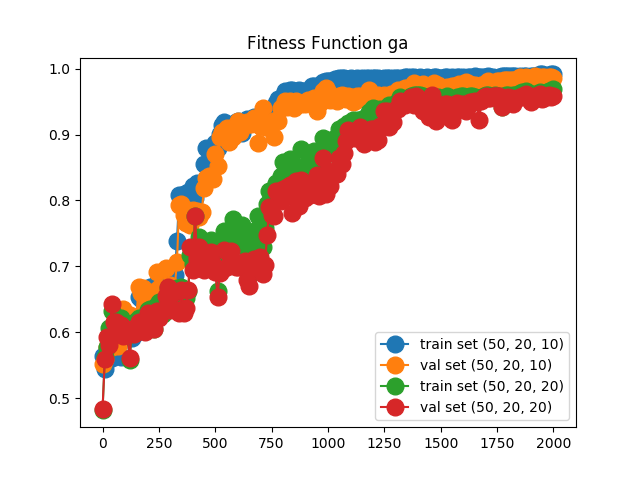
3. The neural network

In this part, the Freddie Mac dataset using neural network from assignment 1 is analyzed using randomized optimization algorithms. Based on Assignment 1, the best neural network had an input layer equal to the number of attributes, an output layer, and five hidden layers with 109, 76, 76, 76, 76 units for each respective hidden layer. The weights within the network were determined using back propagation, and in Assignment 1, achieved a classification rates of 92%.

In this Assignment, I have performed the same analysis using the back-propagation implementation in the JAVA library ABAGAIL to establish a baseline, again which the randomized algorithms will be compared to. I used the standard minimization of the sum of squared distances (SSD) of the error. The weights were initialized randomly and each optimization algorithm was run 5 times. The mean results are given in the following table. Note that in this assignment, a split of 0.1, 0.1 was used to split the data into training and test set, and cross-validation has not be been implemented.

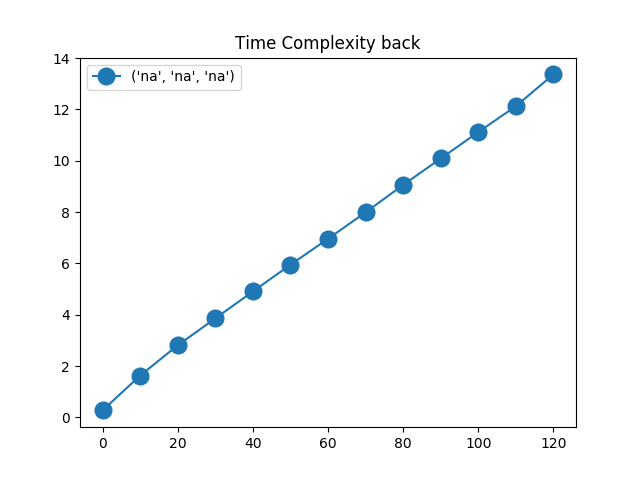
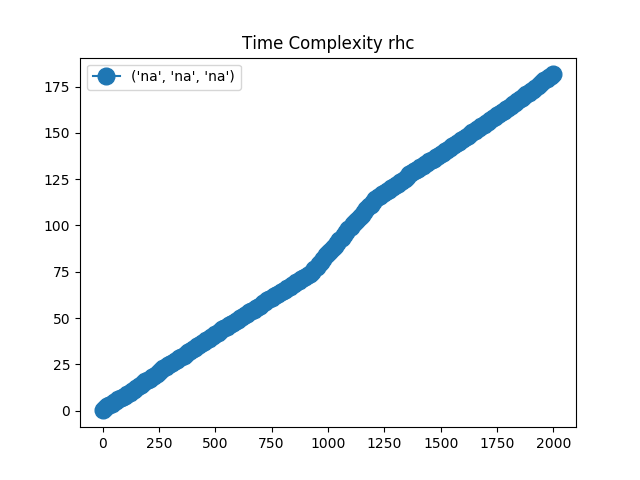
Running time were plotted against number of iterations below.

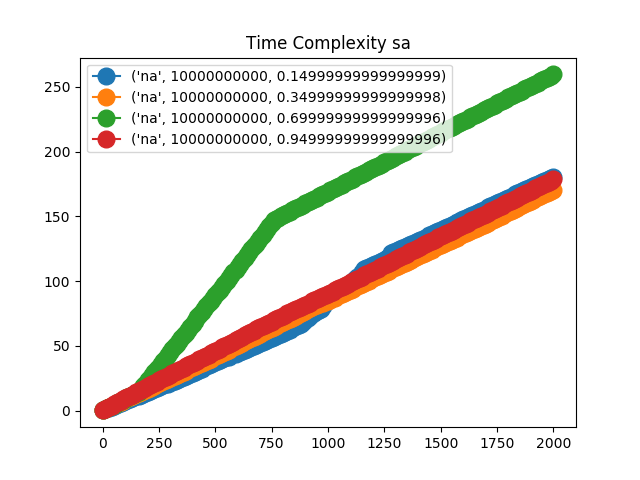
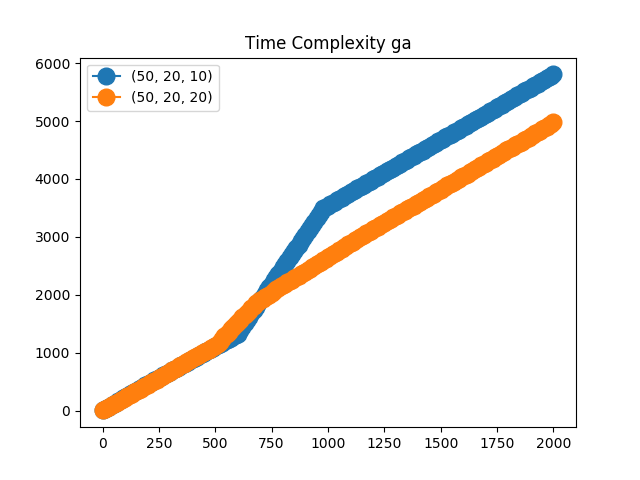
 

Not surprisingly, back propagation has the highest accuracy rate when compared to randomized algorithms. However, among the randomized algorithms, GA performed the best, defined as having the lowest sums of squares errors, in approx. 700 iterations, depending on the parameters chosen. More specifically, when we used a lower number of mutation points in GA, the accuracy rate is observed to be quite good at a low number of iterations. RHC performed second best, but it does take more iterations to converge – but there has been a larger difference between the training errors and test errors (called validation errors in the graph), something that GA does not show.

Running time were plotted against number of iterations below.

Again, not surprisingly, back-propagation converged has the lowest time complexity, followed by RHC and SA. We see that GA took a long time for each iteration. This observation is very much consistent with our observations so far.

4. Conclusion

It can be shown that smooth problems can be solved best with SA or GA (such as neural network training), while MIMIC tend to perform well in Flip Flop problems, which can be described as having discrete smooth cost functions. RHC is an exhaustive search which performs very well on smooth problems, but ran into diﬃculties with problems with discontinuities. All four algorithms perform poorly on problems like TSPs, where special variants of them are needed to obtain good results. If we have a smooth problem with a global optimum, such as in the case of a neural network with a convex cost function, then none of the algorithms can outperform an ordinary Gradient Descent method used in the back-propagation.

1. Using Optimal Dependency-Trees for Combinatorial Optimization: Learning the Structure of the Search Space [↑](#footnote-ref-1)