Random Optimization

Hermann Hans

Department of Computer Science, Georgia Institute of Technology

October 17, 2016

Abstract

The purpose of this document is to explore Random Optimization algorithms, first by applying them to a neural network model using the Image Segmentation data from Assignment one, and then running them on some well-known computer science problems. The first part is done by replacing the back propagation and gradient descent we used previously with another optimization algorithm such as Random Hill Climbing, Simulated Annealing, or a Genetic Algorithm. After each training iteration, the weights are then updated using the optimization or fitness function of the respective algorithm. For the second part, the following three optimization problems were picked to highlight and further explore these algorithms: Knapsack Problem, Counting Ones, and Traveling Salesman. Each problem will show which algorithms work well for that particular domain, and analysis will be provided as to why that might be.

1 Introduction

1.1 General Approach

For each of the three algorithms, separate test classes were implemented based on the Abagail example initially, but modified to get more meaningful data out of it. The output per iteration was modified to not only output the sum of squared errors, but also the training error and testing error at each iteration.

The Image Segmentation data set was split the same way it was implemented for Assignment 1 (30% test, 70% training data). This split was static (not randomized), and for this assignment the data was separated into two different text files (segmentation_test.txt and segmentation_train.txt). Additionally, the output classes were mapped from Strings to integer values (see the *readme* for the mapping).

As these algorithms depend heavily on randomization, multiple iterations of each algorithm were run and then averaged to compensate for outlier runs. This also had the effect of getting smoother graphs as two runs would usually find local optima and then restart at a different iteration, causing the accuracy / error spikes that would be seen with a single iteration to be averaged out across all the runs.

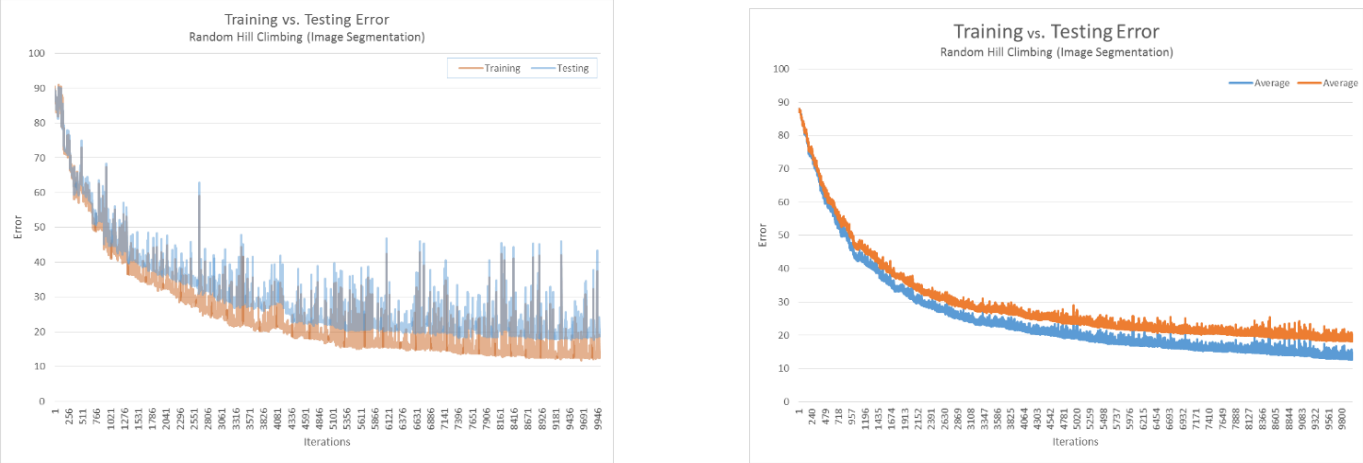


Figure 1: Single run [LEFT] vs multiple iterations averaged out [RIGHT]

2 Randomized Hill Climbing

In randomized hill climbing a random starting point is chosen and then, through application of a neighbor function, will start moving in the direction of improvement until an optimum is found. Using random restart, the algorithm will then repeat this process, using the random starting points to try to get out of local optima and find a global optimum (if there is one!). This works well for datasets where the global optimum has a broad basin of attraction, as the likelihood of picking a point in that basin is large. Of course the opposite is then that if the global optimums basin is narrow, or there are lots of local optima, then it becomes closer to doing a full search across the problem domain.

Applying RHC to the Image Segmentation dataset shows that the training error was still continuously improving (marginally), while the testing error has converged more.

An improvement to RHC might be to keep track of the points where weve already been. This would further improve the running time of the algorithm and avoid unnecessarily having to search the same space that weve already seen.

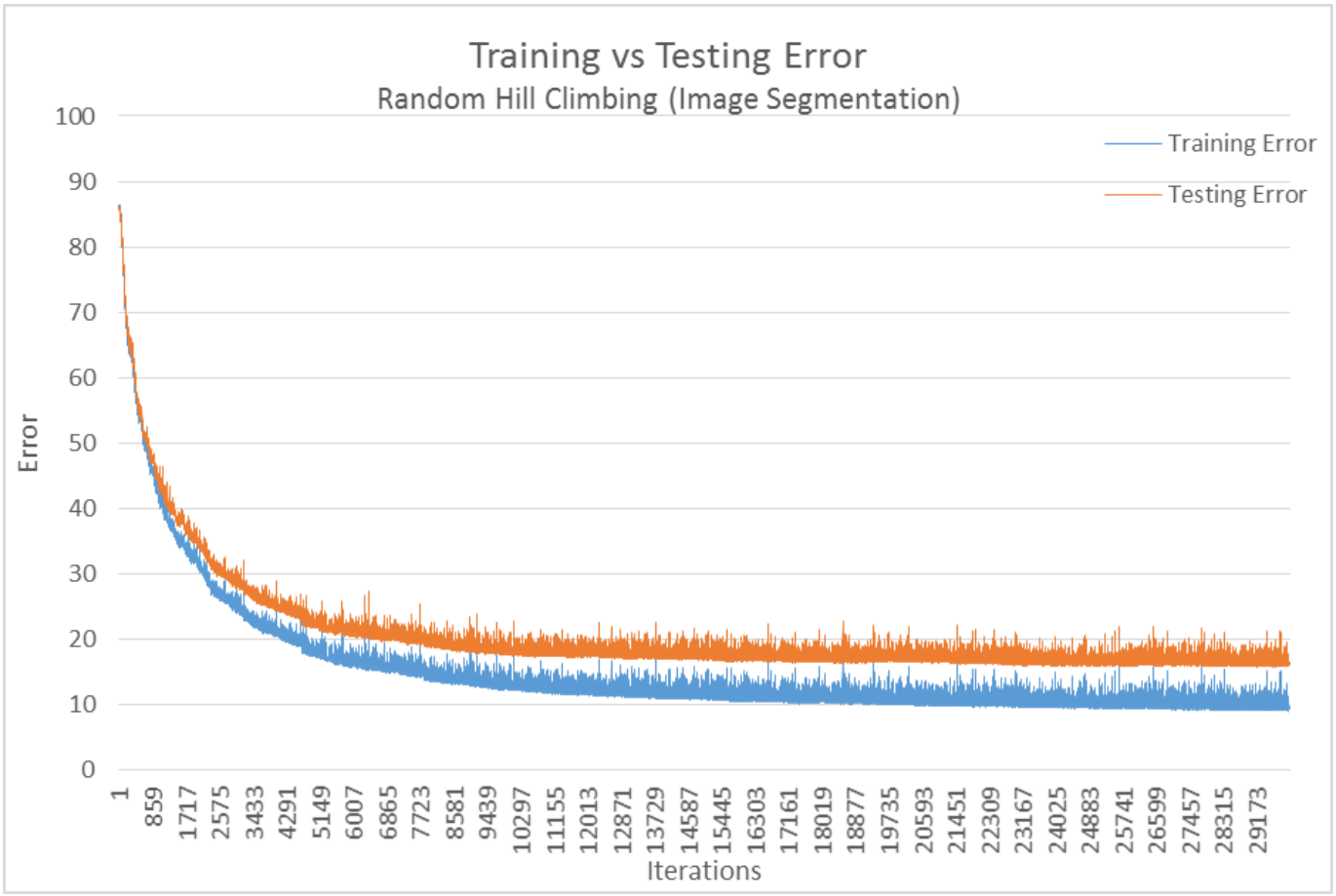


Figure 2: Randomized Hill Climbing - Training vs Testing Error

3 Simulated Annealing

Simulated Annealing is similar to RHC in that its trying to exploit by continuously moving in the direction of improvement with the added benefit of exploring the space. Coming from the analogy of heating and cooling metal to make it less brittle, the same idea is applied by starting with a high temperature T and a cooling exponent C. Initially, when the temperature is high, SA moves around more, exploring the problem space. With each iteration the step size that SA can take decreases as a function of C (cooling off). The longer we run the algorithm, the more it goes from random search to RHC. This feature of *exploring* gives SA the additional benefit of being able to break out of local optima.

SA was run with a cooling component of .7 and .95. to show the effect of the cooling function on the result. As expected, the lower cooling exponent decreases the temperature more quickly and thus converges faster, while the higher cooling exponent of .95 takes longer and even shows an increase in training error within the first 1000 iterations as its exploring more freely.

As SA starts to cool off, and the temperature becomes lower and lower it starts to look a lot more like RHC again as it goes from exploring to exploiting.

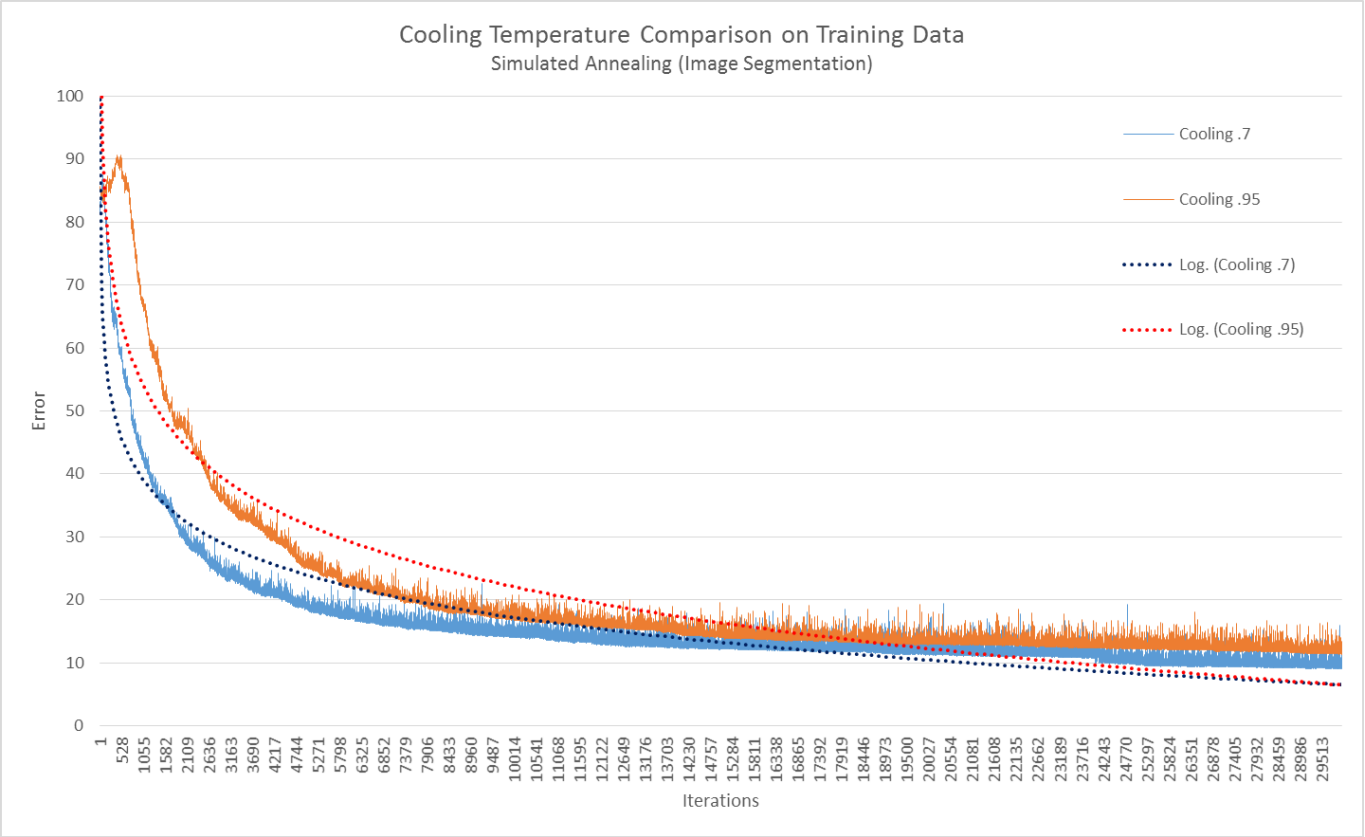


Figure 3: Simulated Annealing - Training vs Testing Error

4 Genetic Algorithm

Genetic algorithms make use mutation and crossover and the fitness function to determine which instances will be used to create the next generation. Theres several different crossover techniques which need to be thought about, such as uniform crossover vs one-point or other functions. Additionally, domain knowledge can also be applied to the mutation function to create better results with each iteration (see Traveling Salesman below).

Unfortunately the results from running GA on the Image Segmentation dataset resulted in the worst performance out of all other algorithms. I tried to do some Grid Search to get better parameters, but these seemed to have little or no effect on the outcome, however, I wasnt able to complete a fully exhaustive grid search and also didnt go the additional step of tuning the crossover function which may have brought the most improvements.

The chart shows iterations only up to 1000, as well as that GA is still improving. An additional test was run with 30k iterations and improvement was still continuing with the improvement (19% misclassification @25500 iterations). Its entirely likely that GA will eventually converge towards the same accuracy as SA and RHC, or even improve upon those, but from running the analysis it could be that this is an effect of the curse of dimensionality, where the number of features and the resulting output classes are too great of a search space for GA to work quickly enough so that better analysis

wouldve been possible.

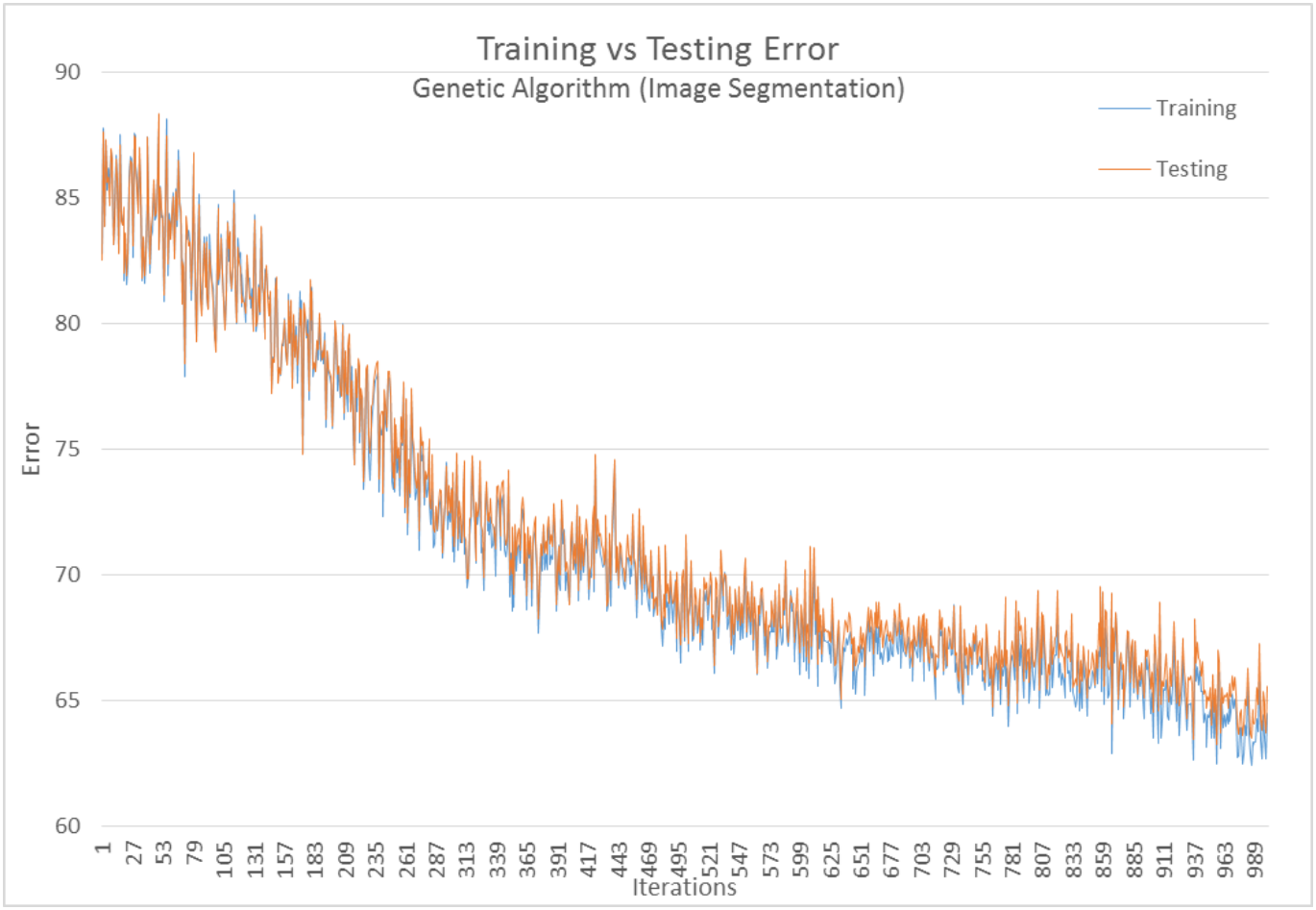


Figure 4: Genetic Algorithm - Training vs Testing Error

5 Knapsack

The knapsack problem can be defined as the problem where given a *set of items* N, where each *item* i has a corresponding *weight* w and *value* v, try to select a subset of N such that we maximize the total value V while not going over a total weight threshold W.

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

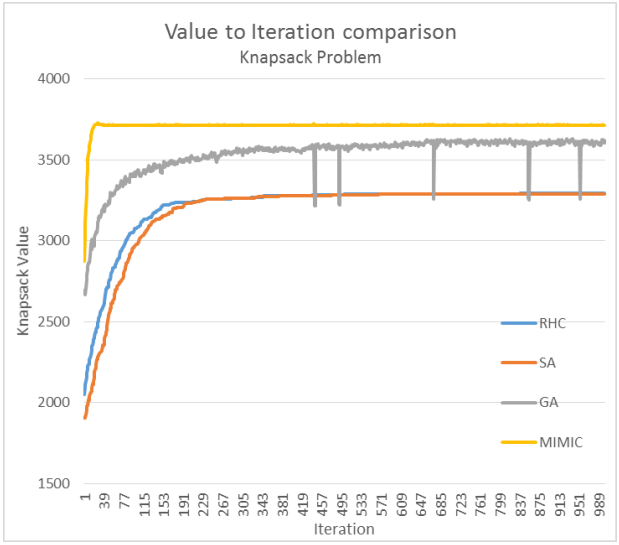
$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}$$

For analysis, the provided KnapsackTest.java file was used as a starting point. The parameters for the knapsack problem were left as is (40 items, 4 instances of each, randomized values no greater

than 50 and a weight threshold of 50). Ten iterations were done on all four algorithms, and the averages of those runs were then used to compare.

What can immediately be seen is that MIMIC and GA do much better than SA and RHC, which are both very similar in terms of what maximum value they find and how many iterations they take in doing so. Also interesting is how many fewer iterations MIMIC takes than all three algorithms. MIMIC converges within the 20th iteration, where SA and RHC are around 170 iterations (with much smaller maximum values). GA starts out higher than both SA and RHC, and levels off slower than the other 3 algorithms while still providing a maximum value thats closer to MIMIC than SA / RHC.

MIMIC does well on problems that have structure to them, where a probability distribution can be captured and then used to successfully refine the model. The reason GA does well is likely related to structure as well, and this would be interesting to pursue further sine the runtime difference for each iteration is much greater for MIMIC than GA. So much so that another interesting comparison would have been to compare the total runtime until convergence towards a maximum happens for both algorithms.



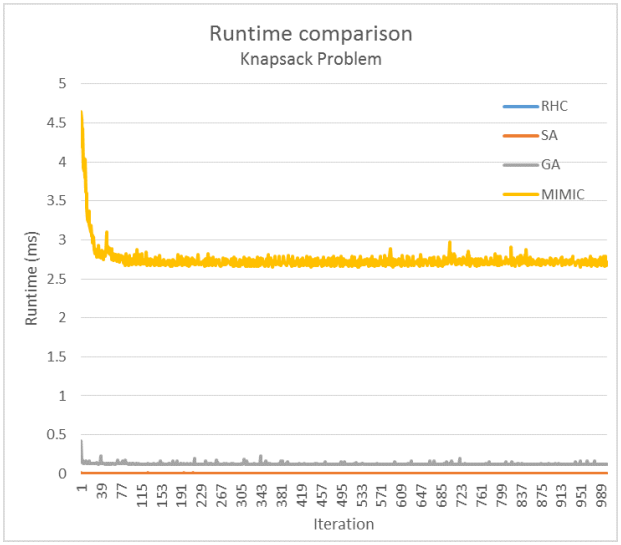


Figure 5: Knapsack - Value to iteration comparison [LEFT] and Runtime comparison [RIGHT]

6 Counting Ones

Counting ones is a very simple problem where given a *bit-string S of length n*, we are trying to maximize the number of 1s in that bit-string. Formally, we are trying to find a string $\vec{x} = \{x_1, x_2, ..., x_N\}$, where $x_i \in \{0, 1\}$ which maximizes the following equation:

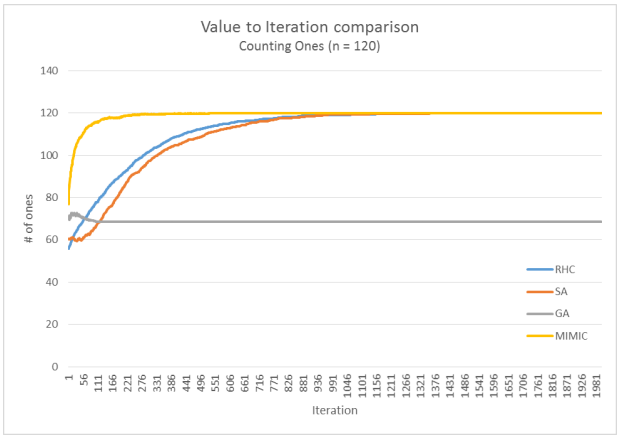
$$F(\vec{x}) = \sum_{i=1}^{N} x_i$$

Our fitness function simply counts the number of ones in the String and returns that count. What makes the problem interesting and worthwhile to use for a comparison is that each bit in the string is an independent of every other bit, and thus delivers no structure, which is where GA and MIMIC would do better on (MIMIC still does well, see below).

The algorithm was run for several sizes of n, all with similar results as the plots shown here for n=120. Again, for each value of n, ten runs were performed and then averaged. Only the number of iterations were increased, but little else parameter tuning was done which may have improved the performance of GA marginally, but probably not as the algorithm doesnt gain anything from mutating in this problem.

When strictly looking at the iterations, MIMIC takes fewer iterations again than RHC and SA and also converges on the optimal solution. However, adding up the runtime of all 2000 iterations for RHC (1.88 milliseconds) is still faster than a single iteration of MIMIC (4.19 milliseconds was the fastest averaged iteration)! This problem clearly shows the preference of RHC and SA where structure is unimportant and explore and exploit work very well.

On MIMIC doing well, the counting ones example was actually talked about towards the end of the Randomization Optimization lecture with regard to the underlying probability distribution and how MIMIC still works even if we have completely independent features.



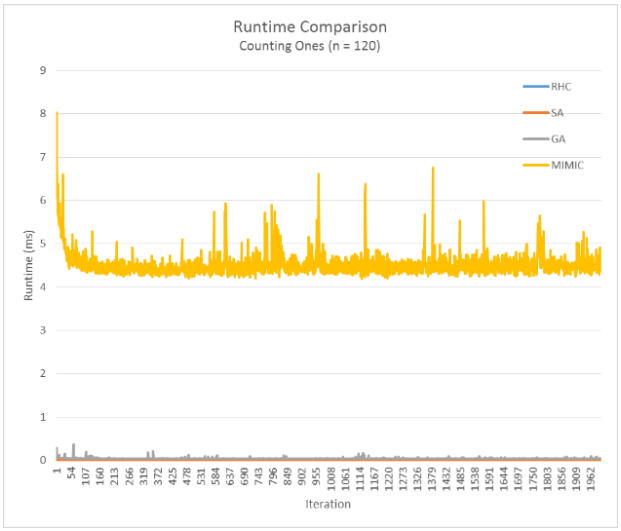


Figure 6: Counting Ones - Value to iteration comparison [LEFT] and Runtime comparison [RIGHT]

7 Traveling Salesman

Traveling Salesman, or TSP for short, is another NP-hard problem. The problem statement is the following:

> *Given a list of N cities and their distances between each other, find the shortest possible path that will visit each location exactly once and ending at the starting location.*

Since we are trying to minimize the distance, we use the inverse (1/distance) as the fitness function on which to train on.

Before going into the comparison, the code that was used for the implementation was from the example that was provided in ABAGAIL. There, the GA crossover function thats used requires some domain knowledge of the problem as when we are doing mutation and crossover we need to ensure that our instance isnt violating the constraints (each city once, and only once). The swap mutation is easy enough to understand, and does mutation by only swapping cities in a given graph instance. For crossover something similar has to be done to ensure that the values from the parents dont end up as duplicates in the offspring.

Using this domain knowledge in GA, TSP becomes a great example for an algorithm where GAs do well. In the graph below its apparent that in very few iterations GA can come close to an optimum value for 1/distance. The number of iterations were kept short here so whats not displayed here is how much more improvement is available for the other three algorithms which havent converged yet. Its also possible that with further parameter tuning (exhaustive grid search for certain values of N) of the other algorithms, they may converge quicker in the beginning. This is probably true for MIMIC as well, which couldve been explored more in the context of TSP and what parameters may provide better results.

To sum up though, MIMICs runtime for each iteration would still be far greater than that of GA which makes it the preferable algorithm here.

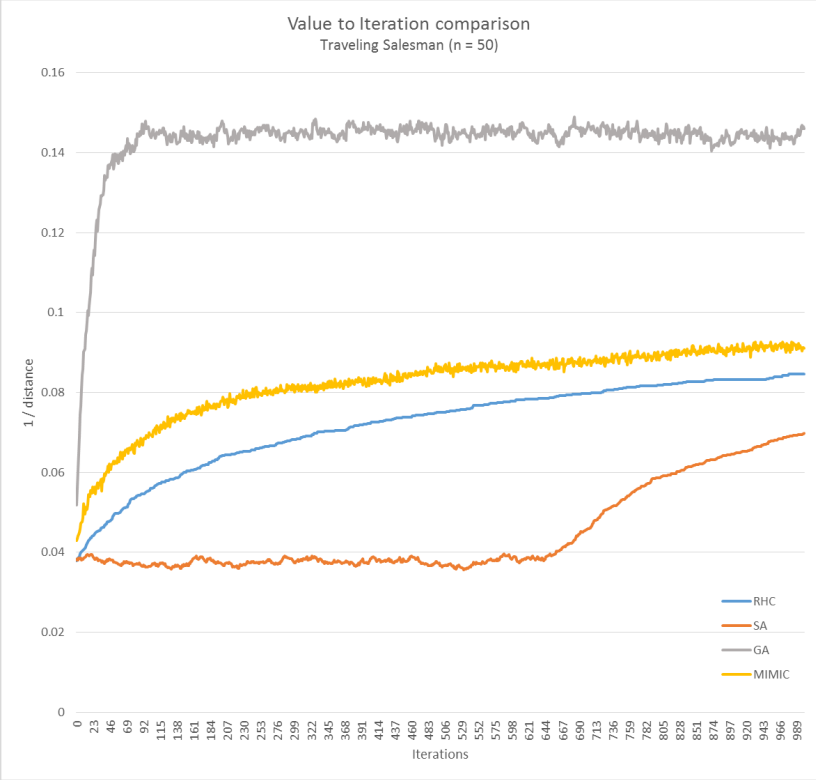


Figure 7: Traveling Salesman - Value to iteration comparison

8 Conclusion

All four algorithms were looked at in closer detail, trying to find the areas where each might excel or should not be used. In general it could probably said that the more complex a problem is (NP-complete, more structure, etc), the more likely it probably is that GA and MIMIC would do well. On the other side, the less complex problems may perform better using SA and RHC.

The exploitive nature of SA and RHC, as well as the explorative addition in SA could be observed in the experiments as well.

For the second part of the report it has to be said that while some parameter tuning was done for each algorithm, an exhaustive grid search for each problem would have made for a more accurate report. This is especially true for GA and MIMIC where not only input parameters matter. For GA more analysis couldve been done on the differences between *SingleCrossOver*, *TwoPointCrossOver* or *UniformCrossOver*. For MIMIC, while a dependency tree covers other distributions such as Uniform or a Dependency Chain, the effect the differences have on the runtime of each iteration couldve also made for an interesting side discussion.