Analysis of Randomized Optimization Techniques Applied to Neural Networks

In order to determine the effectiveness of a few randomized optimization techniques – Randomized Hill Climbing (RHC), Simulated Annealing (SA), and Genetic Algorithms (GA) – I used the Phishing Websites dataset from my analysis of supervised learning algorithms. Instead of K-fold cross validation, I used the holdout method – splitting my dataset into a training (70%) and a testing (30%) set – to evaluate the models built using randomized optimization. The dataset was randomized prior to splitting in order to avoid a loss of generality and lower the variance of the evaluation. While K-fold cross validation helps reduce the variance even more since every instance gets to be in a testing set once, it requires significantly more computation to make an evaluation. Furthermore, the parameters of the classifier – number of hidden layers, nodes per layer, training iterations, etc. – were chosen using 10-fold cross-validation and remained unchanged, so these factors are consistent across all models.

The results in this analysis will be compared to the results from the supervised learning analysis, specifically the artificial neural network (ANN) models, as well as the ZeroR classifier to provide a benchmark for performance. The ANN has 2 hidden layers with 16 nodes each, a learning rate of 0.7, and a momentum of 0.3, which I found to be the best hyper-parameters from the supervised learning analysis. The ZeroR classifier disregards all the attributes, looks at the target output that occurs the most frequently, and always uses that as its prediction. I expect that ZeroR will be the least accurate for this reason. I expect randomized optimization to be less accurate for this dataset, but train faster than backpropagation with stochastic gradient descent.

Randomized optimization tends to work well on non-differentiable functions, by going to "better" positions in the hypothesis space, chosen by some probability distribution. Phishing Websites has 30 attributes and is a binary classification problem where, given features of a website, we must determine whether it is legitimate or suspicious. Some attributes (URLs that redirect, abnormal URLs/scripts, etc.) may be much more significant than others (shortened URLs, existence in Google's Index, etc.). Backpropagation is likely to identify relevant attributes, and assign greater weights to them, tuning out less relevant attributes. Randomized optimization, on the other hand, will attempt to randomly find a good solution, which is likely to be less accurate. However, randomized optimization is less likely to be trapped in local minima than stochastic gradient descent backpropagation.

Since randomized optimization takes random starting points and jumps to random neighbors in the search space, the results are likely to have some variance. Therefore, all tests were run ten times, and the error values and training values were averaged to minimize the variance and its impact on evaluation.

Randomized Hill Climbing

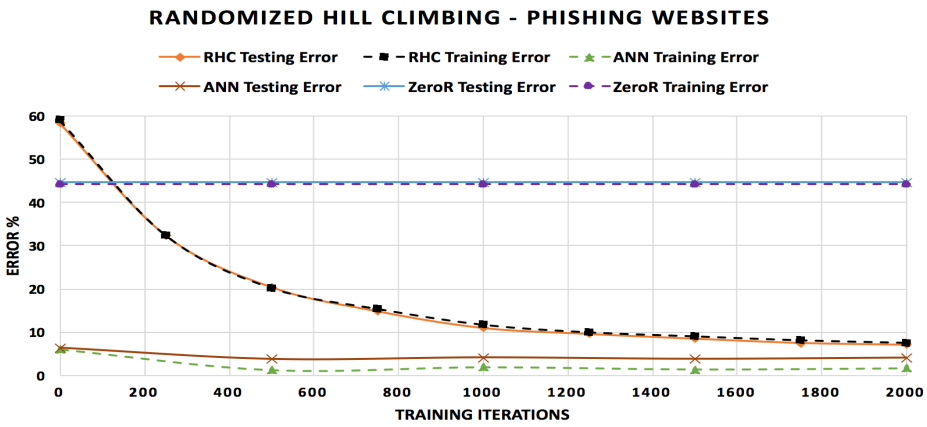

Figure 1: RHC vs. ANN vs. ZeroR

RHC doesn't take in any parameters, so it was trained for 2000 iterations and tested every 250 iterations. This was repeated 3 times, and the values were averaged to minimize the variance of results. The error continues to decrease as the number of iterations increases. At each iteration, we pick a random neighbor to evaluate, and move to if it performs better. As iterations increase, we are likely to explore more data, increasing the probability that we find the optimal value. The training and testing error was almost identical for RHC. One explanation could be simply that I didn't perform enough iterations to overfit the model to my training set. Another explanation is that, unlike backpropagation where we send the error back through the ANN and adjust the weights, with RHC, we only check to see if we reached a better point, we don't use the error to make adjustments. As a result, because they are trying to optimize, RHC does not search across the entire dataset, thus preventing it from overfitting to the training data.

Clearly, it performs more than twice as better than ZeroR in just 500 iterations. However, in as few as 500 iterations, ANN is able to achieve a much lower training and testing error than RHC can with as high as 2000 iterations. Backpropagation allows the ANN to learn from bad classifications and adjust its weights with each iteration, thus optimizing in fewer iterations, at the cost of increased training time, as we will see later on.

Simulated Annealing

In order to find the best hyper-parameters (temperature and cooling rate) for SA, I ran 10 trials each with all combinations of 4 temperatures and 4 cooling rates and averaged them. A low temperature (100) with quick cooling rate (0.02) performed best. High temperatures increase the probability of jumping to a locally "worse" neighbor (which may or may not be globally "better"). So, a low temperature with a quick cooling rate indicates that we trust the algorithm to find the answer without getting stuck in local minima. Consequently, we trust our data less, because we will converge to an answer sooner, without jumping around too much.

Temperature	Cooling Rate	Avg. Testing Error (%)	Avg. Training Error (%)	Avg. Training Time (s)
100	**0.02**	**14.6218**	**14.7816**	**26.4529**
100	0.2	16.2165	16.2316	33.1737
100	0.5	16.3159	16.4333	32.7776
100	0.95	28.4173	28.4349	32.1258
1.00E+05	0.02	17.2778	17.4206	30.0054
1.00E+05	0.2	16.129	16.5728	30.9862
1.00E+05	0.5	16.8827	17.2501	25.9948
1.00E+05	0.95	40.3827	40.526	28.1843
1.00E+11	0.02	15.8216	15.7431	26.2348
1.00E+11	0.2	15.6346	15.9176	25.8513
1.00E+11	0.5	17.6485	18.2992	26.2179
1.00E+11	0.95	51.7033	51.6141	29.2314
1.00E+13	0.02	17.2688	17.5395	26.9514
1.00E+13	0.2	17.332	17.3676	24.7399
1.00E+13	0.5	18.3208	18.5785	25.6581
1.00E+13	0.95	52.448	52.6687	35.1482

Table 1: Simulated Annealing Parameter Tuning (2 hidden layers, 16 nodes each)

After setting the optimal temperature and cooling rate, SA was trained for 2000 iterations and tested every 250 iterations. This was repeated 3 times, and the values were averaged to minimize the variance of results. The error falls as iterations increase. Similar to RHC, with more iterations, we take more of our data into consideration, so we are more likely to find the optimal value. We see that both training and testing error consistently fall, indicating that we are not overfitting. Once again, this can be explained by the fact that we are not performing backpropagation to adjust weights based on the calculated error, preventing the ANN from overfitting. It is noteworthy that a low temperature and fast cooling rate performed best, because this means SA behaved similar to RHC. It means we trust our data less, since we converge to an answer without considering too much training data.

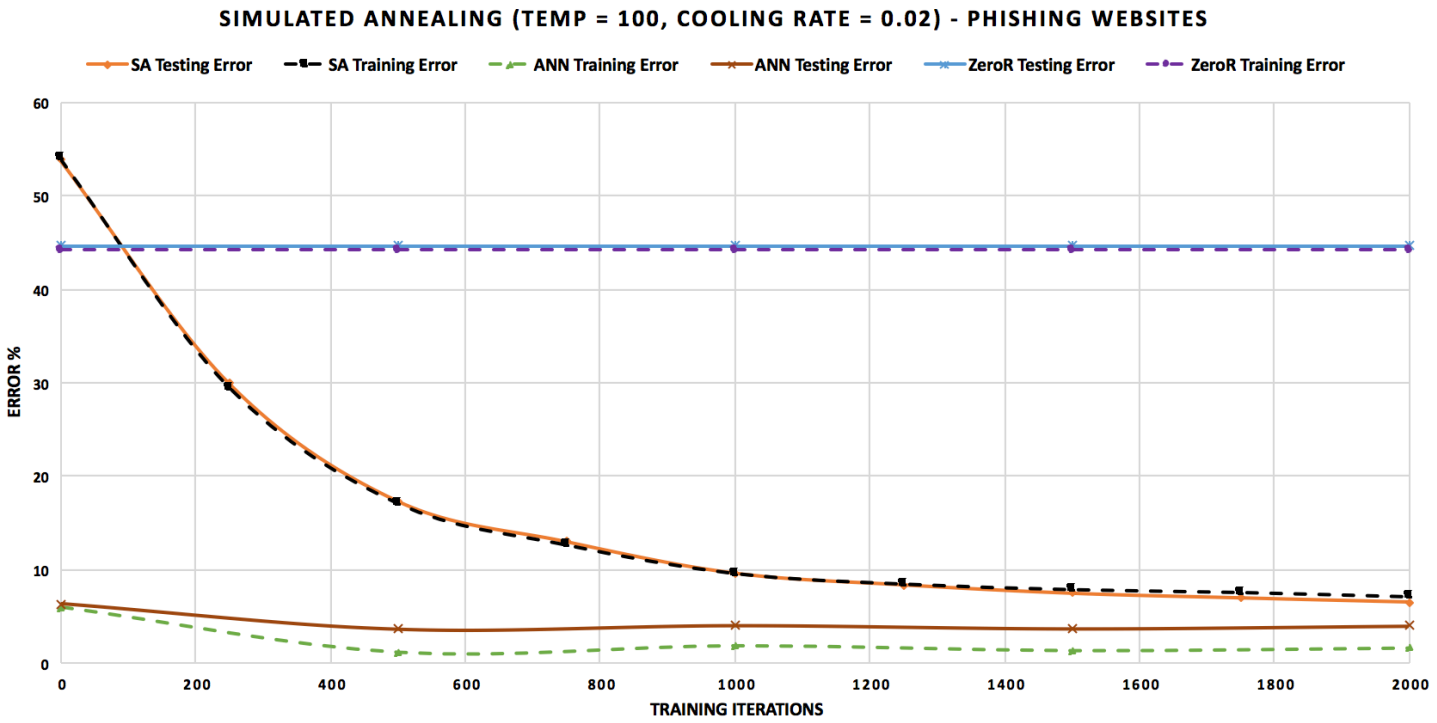

Figure 2: SA vs. ANN vs. ZeroR

SA performed a lot better than ZeroR, getting higher accuracy in as few as 250 iterations, and converging in roughly 1250 iterations. The ANN using backpropagation achieved a much lower error rate than SA because it learned from its calculated error at every iteration and adjusted its weights to improve its performance for the next iteration. In just 500 iterations, ANN was able to outperform SA even after 4 times as many iterations.

Genetic Algorithms

In order to find the best hyper-parameters (population size, # to mate, # to mutate), I ran 10 trials each with a few different combinations and averaged them. A low population size (100) with 50% mating rate and 10% mutation rate gave the best results. Although higher population sizes did lower the error, the time increased notably, so I did not consider them to be the best parameters.

Pop. Size	To Mate	To Mutate	Avg. Testing Error (%)	Avg. Training Error (%)	Avg. Training Time (s)
100	**50**	**10**	**11.031**	**10.9823**	**552.985**
100	5	1	25.2457	25.203	133.1951
100	75	10	11.3988	11.636	810.2219
200	100	20	8.5078	8.4763	1142.3519
500	200	30	7.7599	7.9337	1882.461

Table 2: Genetic Algorithm Parameter Tuning

After setting the optimal hyper-parameters, GA was trained for 2000 iterations and tested every 250 iterations, repeated 3 times and averaged to reduce variance.

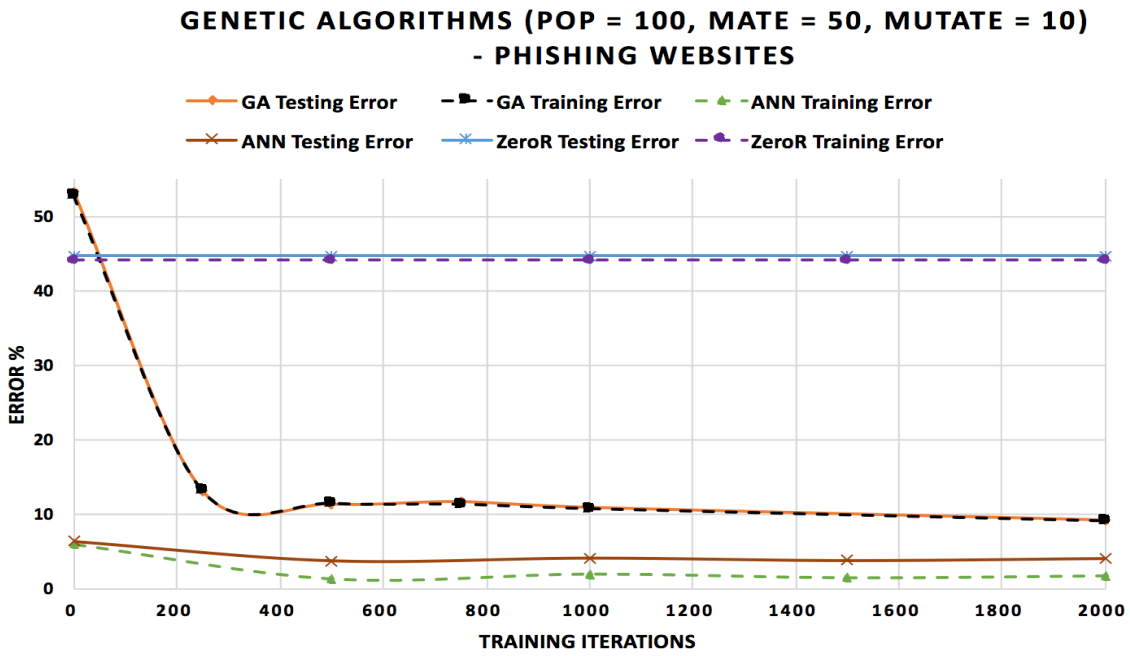

Figure 3: GA vs. ANN vs. ZeroR

The error falls rapidly going from to 250 training iterations, but then plateaus. Once again, there is no evidence of overfitting, likely due to the lack of backpropagation as explained earlier. GA performed marginally better than RHC and SA. Picking a small population size means that we are restricting ourselves to a smaller subset of our data. A high mating rate means we deviate quickly from the original population, allowing us to explore a wider range of populations. Finally, a high mutation rate would help us find global optima, since it reduces the likelihood of settling in a local optimum if we mutate our population. Picking too high of a mutation rate, however, might cause us to miss our global optima by changing our population too much. The manner in which the error plateaued may be due to settling in a local optimum. The high crossover rate allowed GA to swiftly tune out irrelevant attributes (Phishing Websites has 30 total attributes) and find an optimal point.

GA performed significantly better than ZeroR, almost 4 times better in just 250 iterations. Once again, our ANN with backpropagation outperformed GA, because of its ability to adjust weights at each iteration. The ANN converged to a very low error in just a few iterations, much like GA, and it was able to perform roughly twice as better.

Time Requirements

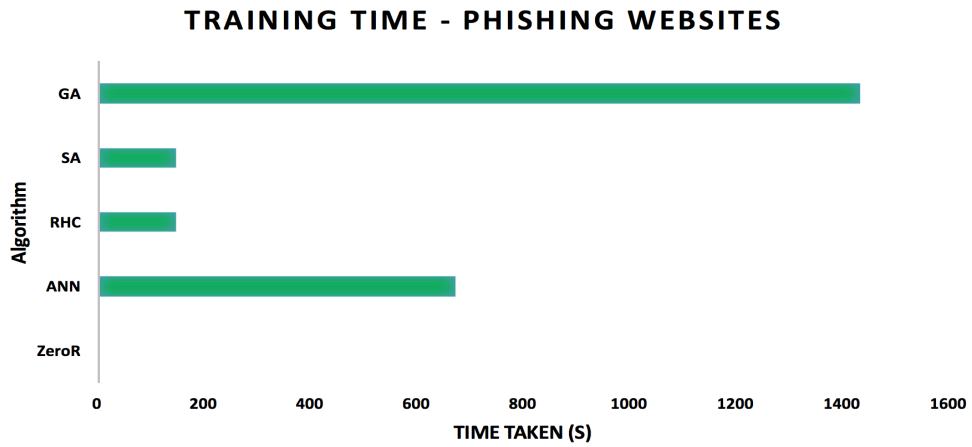

Figure 4: Total Training Time For 2000 Iterations

ZeroR took no time to train, since all it needs to do is identify which output appears more frequently, and use that for its prediction. However, it was extremely inaccurate, so it is not a good choice. RHC and SA took a little longer (~145 seconds). The only difference between these algorithms is that SA may move to a locally "bad" neighbor, while RHC is greedy, which is why they took roughly the same time to train. They both achieved over 90% accuracy with 2000 iterations, so they worked quite well for my dataset in a reasonable amount of time. ANN with backpropagation took nearly 700 seconds to complete 2000 iterations. It requires more time, because after each iteration, it calculates the error in its classifications and goes through the whole network, adjusting weights to improve for the next iteration. ANN achieved over 95% accuracy in just 250 iterations, but it took longer to train. GA took the longest by far, nearly 1500 seconds to complete 2000 iterations. GA is a computationally complex algorithm, because it needs to evaluate the fitness function for every member of the population at each algorithm. This explains why a larger population significantly increased the training time for GA.

The best choices for Phishing Websites would be SA or ANN. SA is preferred over RHC because of its ability to avoid local optima. ANN, however, had the highest accuracy. A final test of SA with a higher temperature (1E20) and slower cooling (0.2) and 5000 training iterations yielded just over 95% accuracy, while taking ~270 seconds. Based on this, it seems that SA would be the best algorithm to model this dataset. Even though it required more iterations to match the accuracy of the ANN with backpropagation, it did so in less wall-clock time.

Four-Peaks Problem

The Four-Peaks problem takes in an N-dimensional input vector and some value T. The global maxima for this function is achieved when there are T+1 leading 1's followed by all 0's or T+1 trailing 0's followed by all 1's, i.e., there are two global maxima. Two suboptimal local maxima occur when there are N 1's or 0's.

For all trials, N was set to be 200, and T was set to be 40. In order to find the best hyper-parameters, I did 5 trials of 5000 iterations each for SA, GA, and Mutual-Information-Maximizing Input Clustering (MIMIC), and averaged the values to minimize the variance. The highlighted and emboldened values were the ones chosen for each algorithm.

Simulated Annealing (Temp. = 1E13)			**Genetic Algorithms (Pop = 300, ToMutate = 50)**			**MIMIC (Sample size = 200)**		
Cooling Rate	Averages		ToMate	Averages		To Keep	Averages	
	Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)
0.999	2	24	200	80.8	2613	100	24.4	234900.4
0.95	26.6	24	150	84	2306.4	75	40.4	201461.6
0.9	24.2	23.6	**100**	**84**	**2183.4**	50	61.8	187197.4
0.5	**34**	**33.2**	50	76	2229	**20**	**197.6**	**133362.8**
0.2	26.6	30.8	20	70.4	1982.2	5	194.4	90448.2
0.02	21.8	27.6	5	72.6	1009.4			

Table 3: Tuning Parameters for Four-Peaks Problem

With the best hyper-parameters, each algorithm was run three times, and the values for fitness function and training time were averaged to minimize the variance and enable a more accurate evaluation. Since the time values were heavily skewed by MIMIC's long training times, the natural log of these values is presented for clarity and effective comparison.

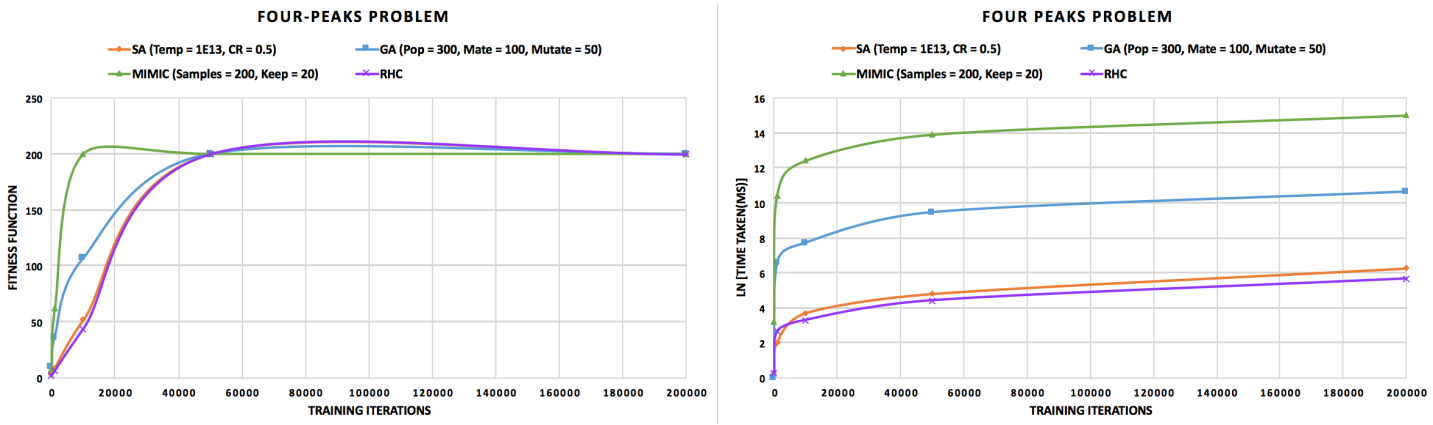

Figure 5: Four Peaks Problem - LEFT: Fitness Function, RIGHT: Training Time

Since all algorithms converged to a fitness function value of 200, we will assume this is a global maximum. It is clear to see in Figure 5 (LEFT), that MIMIC reaches the global maximum in the fewest number of iterations (10000). GA performs the next best for fewer iterations, while RHC and SA perform roughly the same, converging at about 50000 iterations.

However, when we look at Figure 5 (RIGHT), we see that MIMIC takes exponentially much more time than the other algorithms. While GA takes less time than MIMIC, it still takes considerably longer than both RHC and SA. This can be explained by the inner workings of each algorithm. At each iteration, RHC and SA only check if the randomly chosen neighbor gives a higher fitness function value. In order to perform crossover, GA picks two "parent solution" through a fitness-based process, i.e., solutions with a higher fitness function value have a higher probability of being selected. Clearly, if the fitness function is expensive to evaluate, the training time

for GA is notably higher. Similarly, MIMIC works by choosing the fittest samples at each iteration and generating a new probability distribution from them, so it needs to evaluate all the samples, making each iteration much more time intensive.

The Four-Peaks problem highlights the strengths of simulated annealing. Given the proper temperature and cooling rate, SA is able to search through a discrete space, avoid local maxima and settle in a global maximum. For a large N, our search space is quite large, since each neighbor is essentially reached by flipping a bit. In these tests, N = 200, which means we have a total of 2^{200} bit strings to search through. Therefore, SA requires a high number of iterations to find the optimal solution in the large search-space, but since each iteration takes less wall-clock time, SA is able to find the global maximum significantly faster than MIMIC, which needs fewer iterations. RHC also performs well in our tests, however, because it is a greedy algorithm, it is liable to get stuck in local maxima for larger search-spaces.

Traveling Salesman Problem

The Traveling Salesman problem is an NP-Hard problem, where, given a graph with N-vertices, we must find the shortest path that visits all nodes. All tests were run on a graph with 50 vertices. In order to change Traveling Salesman problem to a maximization problem, the evaluation function returns the inverse of the distance, so that shorter distances give a higher fitness value.

In order to find the best hyper-parameters, I did 5 trials of 5000 iterations each for SA, GA, and MIMIC, and averaged the values to minimize the variance. The highlighted and emboldened values were the ones chosen for each algorithm.

Simulated Annealing (Temp. = 1E11)			**Genetic Algorithms (Pop = 200, ToMutate = 10)**			**MIMIC (Samples = 200)**		
Cooling Rate	Averages		ToMate	Averages		To Keep	Averages	
	Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)
0.999	0.040514943	12.8	200	0.143719376	4349.6	100	0.110783472	209526.6
0.95	0.107880112	13.4	150	0.148733684	4894.4	75	0.117199888	*204205.8*
0.9	0.10949094	12.6	100	0.151422763	3484.6	50	0.097550712	201866.8
0.5	0.110149109	21	50	0.160523086	2165.4	20	0.118020395	192892.6
0.2	**0.122141319**	**17.8**	**20**	**0.158435734**	**1416.8**	**5**	**0.134124248**	**189443.6**
0.02	0.105777492	21.8	5	0.132873453	880			

Table 4: Tuning parameters for Traveling Salesman Problem

With the best hyper-parameters, each algorithm was run three times, and the values for fitness function and training time were averaged to minimize the variance and enable a more accurate evaluation. Since the time values were heavily skewed by MIMIC's long training times, the natural log of these values is presented for clarity and effective comparison.

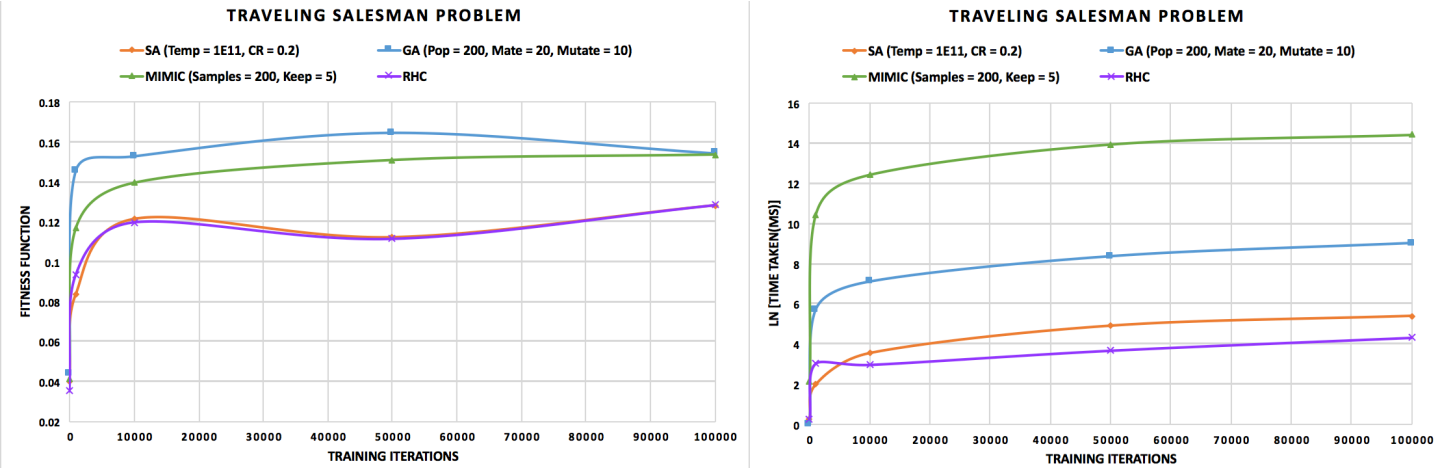

Figure 6: Traveling Salesman Problem - LEFT: Fitness Function, RIGHT: Training Time

Since the Traveling Salesman problem is NP-hard, that means that we do not (at least till now) have a polynomial time algorithm to solve it. Therefore, the algorithm with the highest fitness value will be considered to be the best, since it gets closer to the global maximum (which is unknown) than the rest.

Looking at Figure 6 (LEFT), we see that GA has the highest fitness value, and it gets there with the fewest iterations. MIMIC is the next "fittest" algorithm, followed by SA and finally RHC. Even with a very high number of iterations (100000), RHC and SA do not come close to the fitness values obtained by GA or even MIMIC. The number of paths in a graph with 50 vertices is extremely large, which gives an extensive search-space, with several local maxima. As a result, RHC is likely to get stuck in a local maximum. Even though SA may jump out of a local maximum, because there are so many of them, it is unlikely to find the global maximum. To test this more extensively, it would help to try it with very high temperatures for a large number of iterations.

Looking at Figure 6 (RIGHT), we see that, once again, MIMIC takes exponentially more time to train as compared to the other algorithms, and it does not have the highest fitness value even with a large number of training iterations, so MIMIC is not suited for this problem. GA takes more wall-clock time (~e^9ms) to train than RHC (~e^4 ms) and SA (~e^5ms). However, even with a high number of iterations, RHC and SA are less fit than GA.

The Traveling Salesman problem highlights the strengths of genetic algorithms. Unlike RHC and SA, GA starts with a set of entire solutions and, at each generation, selects a certain number of "parent" solutions to crossover and create a "child" solution for each pair of parents. The parent solutions are chosen by virtue of their fitness, i.e., a solution with a higher fitness value is more likely to be chosen for crossover. As a result, GA allows us to search through a varied population. Comparatively, RHC and SA only consider one solution at each iteration, evaluating whether or not it is better. Even though SA may move to a locally "bad" point in the hope of reaching a globally "good" point, the search space of all possible paths for a graph of 50 vertices is simply too large.

MIMIC generates a set of samples from a probability distribution and tries to assess the likelihood of the global maximum being in that set. At each iteration, MIMIC chooses the best samples, creates a new probability distribution, and repeats. For my tests, the number of best samples retained was very low (5), which might have caused it to settle in a local maximum. For the Traveling Salesman problem, MIMIC might perform better if we have a larger set of samples at each iteration, and we also retain a higher number of best samples. This, however, would also result in much longer training time, because MIMIC needs to calculate the fitness of each sample at each iteration.

Knapsack Problem

The Knapsack problem is an optimization problem where, given a set of items with some weight and value each, we must find how many of each item to include so the total weight is at most a given limit, and the total value is as high as possible. Specifically, I tested the bounded knapsack problem, where the maximum copies of each item is restricted. The bounds of the problem are:

- Number of items = 40
- Maximum number of copies for each item = 4
- Maximum value for a single item = 50
- Maximum weight for a single item = 50
- Weight limit = 3200

In order to find the best hyper-parameters, I did 5 trials of 5000 iterations each for SA, GA, and MIMIC, and averaged the values to minimize the variance. The highlighted and emboldened values were the ones chosen for each algorithm.

Simulated Annealing (Temp. = 1E7)			**Genetic Algorithms (Pop = 200, ToMutate = 25)**			**MIMIC (Samples = 200)**		
Cooling Rate	Averages		ToMate	Averages		To Keep	Averages	
	Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)		Fitness Function	Time Taken (ms)
0.999	1613.642052	1.6	200	3825.302284	1837.2	100	4047.428666	10773.6
0.95	3006.448637	5	150	3798.71969	1458.8	75	4078.940673	9814.6
0.9	**3421.08607**	**5.6**	**100**	**3809.810405**	**1078.2**	50	4049.505379	8111.2
0.5	2940.303629	7.8	50	3799.645933	738.6	20	4120.259435	7271.2
0.2	2829.062795	5.8	20	3759.269422	530.8	**5**	**4147.448174**	**6567**
0.02	2980.981747	14.2	5	3679.217685	426			

Table 5: Tuning Parameters for Knapsack Problem

With the best hyper-parameters, each algorithm was run three times, and the values for fitness function and training time were averaged to minimize the variance and enable a more accurate evaluation. Since the time values were heavily skewed by MIMIC's long training times, the natural log of these values is presented for clarity and effective comparison.

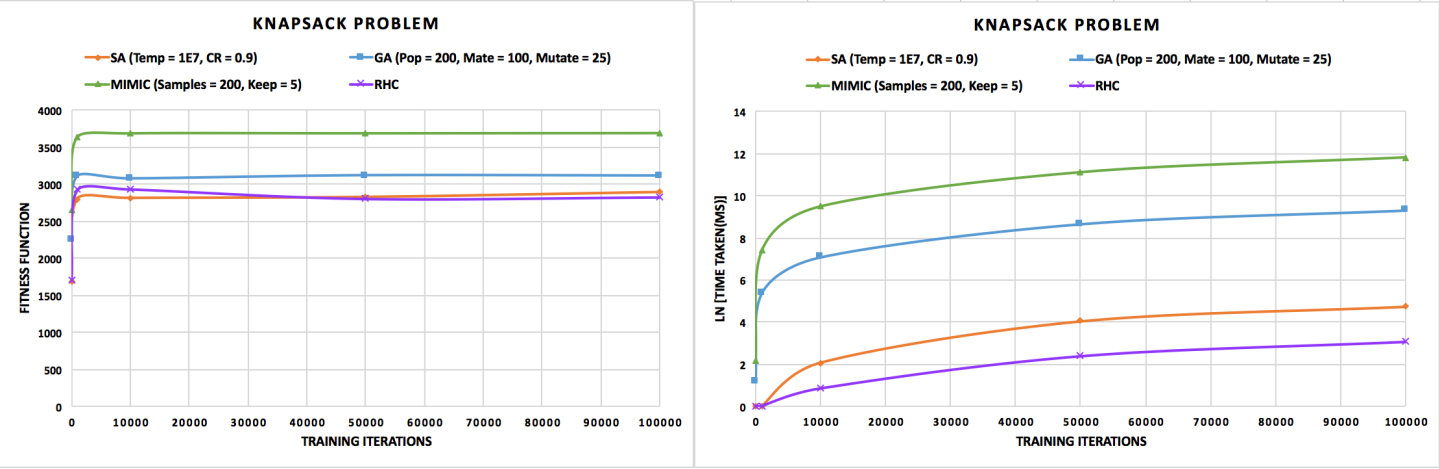

Figure 7: Knapsack Problem - LEFT: Fitness Function, RIGHT: Training Time

Since the Knapsack problem is NP-complete, that means that we do not (at least till now) have a polynomial time algorithm to solve it. We do have a pseudo-polynomial time algorithm, $O(nW)$, using dynamic programming, but it is not truly polynomial, because it depends on the length of the input W. Therefore, the algorithm with the highest fitness value will be considered to be the best, since it gets closer to the global maximum (which is unknown) than the rest.

Looking at Figure 7 (LEFT), we see that MIMIC converges to the maximum fitness value very fast, in as few as 1000 training iterations. As we increase the number of iterations, MIMIC does not drastically improve its fitness. In fact, looking at the values for MIMIC in Table 5, it appears that the maximum might lie somewhere between 1000 and 10000 iterations, the 2nd and 3rd test points used in Figure 7 (LEFT), so, redoing this experiment with iterations between these numbers may improve MIMIC's performance. Similarly, for GA, SA, and RHC, the maximum for these algorithms may lie between 1000 and 10000 iterations. However, with the test data we have, it is clear to see that no other algorithm obtains the high fitness value that MIMIC does.

Looking at Figure 7 (RIGHT), we see that MIMIC takes, by far, the longest time to train. RHC and SA take the least time to train, however, they also have the lowest fitness values, so they are not a good choice for the Knapsack Problem. RHC is expected to settle in a local maximum due to its greedy nature. SA may break out of local maxima because of its ability to take locally "bad" steps, however, for a large number of items, and with

sufficiently high limits on their weights and values, our search-space is extremely large. Since SA can only consider one neighbor at a time, even a very high number of iterations (100000) is not enough to completely search through all the data. GA takes much longer than RHC and SA to train, but does not perform much better. GA may be able to perform better if we give it larger populations, and a higher mutation rate, since this would prevent it from settling in local maxima.

In order to improve the analysis of the randomized optimization algorithms' performance on the Knapsack problem, I would choose a set of iterations between 1000 and 10000 because, from Table 5, it seems like all the algorithms performed better at 5000 iterations. This strengthens the argument for MIMIC, because, if we lower the number of iterations, the wall-clock time MIMIC requires to train is reduced.

The Knapsack problem highlights the strengths of MIMIC. MIMIC works by creating a probability distribution that describe where the global maxima are likely to occur. It uses this distribution to generate samples. It retains a certain number of best samples, and uses them to create a new probability distribution. This allows it to, at each iteration, weed out local maxima, and focus on the globally best samples. Even though MIMIC requires a long time to run, since none of the other algorithms can maximize the fitness function like MIMIC does, it is clearly the best choice for this problem.