Andrew Dodd
adodd30@gatech.edu
2/9/21
Machine Learning (CS 7641)

Assignment 2: Randomized Optimization

Abstract

This assignment covers using four algorithms - Random Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithms (GA) and MIMIC for a number of optimization problems including (1) Four Peaks, (2) K-Colors, and (3) Count Ones. Second, we will use these algorithms to optimize a neural network to perform classification on this dataset from Assign. 1 - and compare our results. MLRose was used extensively in all sections of this assignment.

1. Three Problems Chosen

The three problems chosen for this assignment were four peaks, K-Color, and Count Ones. Each of these problems was chosen because they would allow the researcher to highlight the strength of the algorithm in the particular problem.

1.1 Four Peaks

The Four Peaks problem is as follows - given a vector of length N, and a percentage (p), find the maximum fitness of this vector such that fitness is defined as the formula:

$$f(x) = \max(\#\text{ leading 1's}, \#\text{ trailing 0's}) + R(x, T)$$

Where R is:

$$R(x, T) = \{N \text{ if } (\#\text{ leading } 1's > T * N \text{ and } \#\text{ leading } 1's > T * N), \text{ else } 0\}$$

And T is some percentage, 0.2 was used for this problem. There are two global optima for this problem for any N or T. With N = 10, T = 0.2, we get {1, 1, 1, 1, 1, 1, 1, 0, 0, 0} or {1, 1, 1, 0, 0, 0, 0, 0, 0, 0}. Here, there is *some* structure with the input. If we set a middle value to a 1, we may expect to set other middle values to 1's in order to maximize R(x, T). Similarly, there is the opposite case if we use a 0 in the middle. The problem here is not well suited for MIMIC due to the length of time taken to approximate the distribution and the cheapness of function evaluations. We can expect the SA to find a good solution in reasonably low time, given that it explores the entire problem space with a high temperature and then becomes progressively more specific. We can expect RHC to perform okay here if there are enough restarts - there are many local optima, so the restarts will be important to avoid those.

1.2 K-Color Problem

The K-Color problem is given a connected graph with nodes N and edges E (example is Figure 1), our goal is to minimize the number of neighbors with the same color. Specifically, the fitness is calculated as follows:

$$f(x) = \sum_{i}^{N}\sum_{j}^{M} c_i = c_j, \; subject\ to\ i \neq j$$

Where M are the neighboring nodes, and c is the color of a given node. In highly connected graphs, it will be impossible to prevent neighbors from having the same color in some cases. This problem has a lot of structure, so we can assume that algorithms that consider structure might do well here.

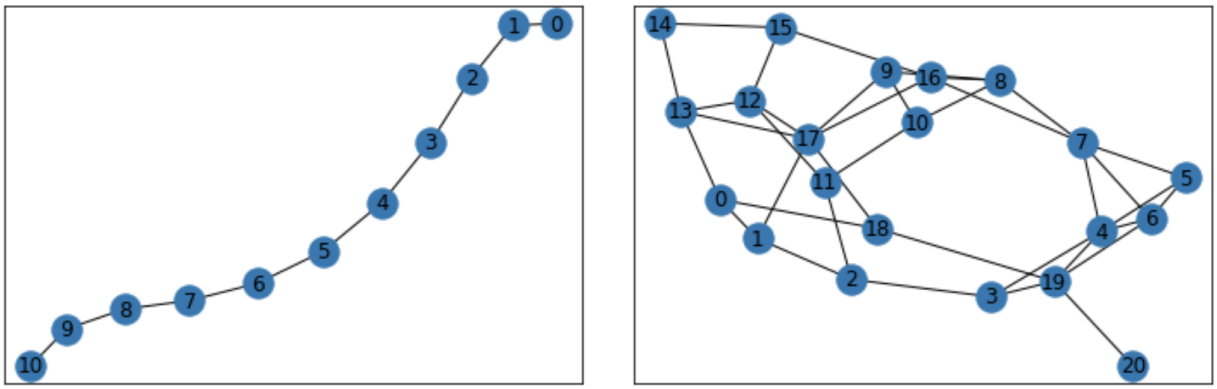

Figure 1: A few examples of undirected graphs used to formulate the K-Color problem. Later analysis that demonstrates fitness as a function of iteration is performed with the right graph, while analysis that finds time until max fitness was performed with the left graph (the researcher needs to know the ground truth optimal fitness to perform the latter test).

1.3 Traveling Salesman

The last problem explored is simply what is the optimal route to hit all nodes. It is traditionally applied on a list of (x,y) coordinate pairs. The optimal route minimizes the distance traveled. This particular problem is interesting due to its structure and how changing one node significantly changes the rest of the choices. It has applications in supply chains and logistics, among others.

2. Comparison of Performance on Three Problems

2.1 Four Peaks

Looking at the four peaks problem for a vector size of 20 (Figure 2), we can plot fitness as a function of iteration and make a number of observations[1]. The first is that MIMIC almost immediately

[1] For this problem, all three algorithms had their hyperparameters changed in different ways. Some examples - GA population size was varied to see what was most effective (did not affect much, as long as in the ballpark of 200), SA decay rates were explored and number of restarts was varied for RHC. These parameters were tested in each of the three problems. In general, the effects of tuning these were not *huge*. Generally, the ballpark of each parameter was effective in getting good results.

optimizes the problem (in terms of algorithm iterations, ***not*** fitness function evaluations). The second is that SA looks to optimize very slowly. Meanwhile GA and RHC are comparable in their fitness after fifty iterations. However, this plot does not tell the whole story; SA looks at an order of magnitude fewer neighbors when making its next move compared to GA or RHC. And we have not yet looked at larger problem spaces or the effects of this upon time.

Once we consider larger problem spaces (Figure 3), we find that MIMIC cannot solve this problem within reasonable time constraints. RHC and GA similarly become time consuming past N=35. In the context of time to find the optimal, SA performs best. It can be theorized that it performs best here due to its ability to combine exploration with exploitation. In this case, these explorative steps allow it to have a better chance of finding the global optima. It is possible that GA does not perform as well here because the population rebuild step is relatively slow and so it does not explore a wide space quickly.

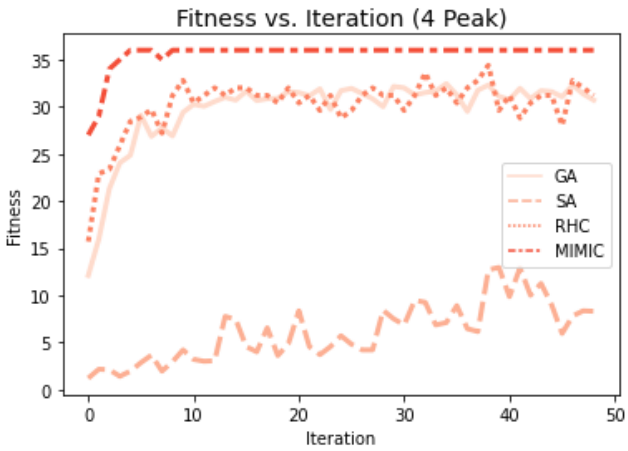

Figure 2: Fitness vs. Iteration on the Four Peak Problem.

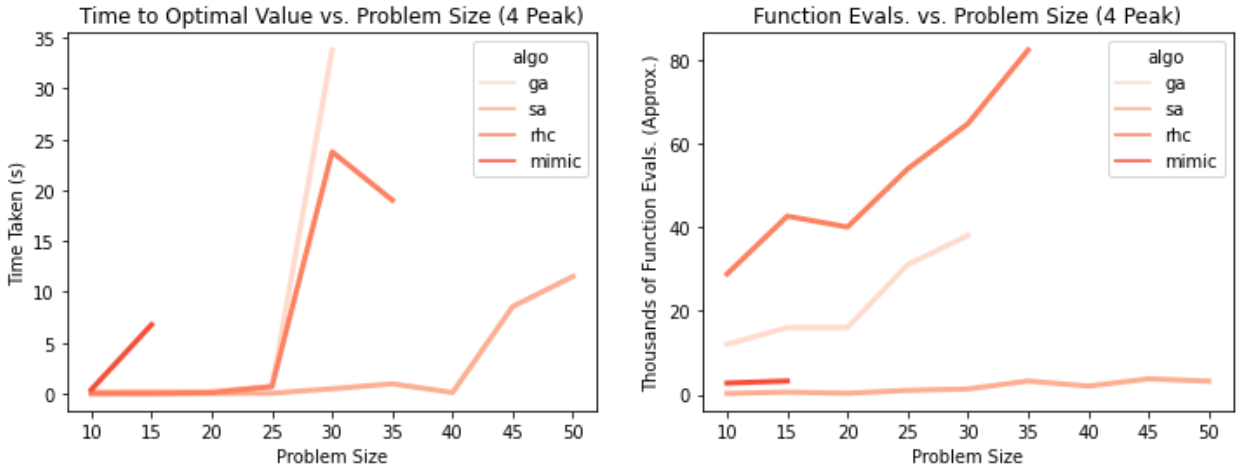

Figure 3: Time to Optimal Value vs. Problem Size (left). Fitness function evaluations vs. Problem Size (right). Both of these on the Four Peak Problem.

2.2 K-Color Problem

On the K-Color problem we find again that MIMIC optimizes the problem quickly (in terms of iterations), and in graphs with large structure, performs the fastest (Figure 4). When ordered by time, somewhat surprisingly SA performs the next best - maybe again due to its explorative and exploitative nature. These are followed by GA next, and RHC last. It seems logical that RHC performs poorly here. It quickly asymptotes to near infinite time. This is due to this problem having many local minima. For example if we change one node, we may be forced to change the nodes around it, moving towards one low energy state of the graph. However, this particular graph may have a global optimum with the original node being not changed but the instead its neighbors should have been changed. It is this interconnected effect that makes the RHC impractical here. Furthermore, it was noticed that when the population size of MIMIC was too small (< N/10), the algorithm did not perform as well (Figure 5). This is most likely because too small of a population prevented MIMIC from accurately predicting the joint population of the data.

Figure 4: Fitness vs. Iteration on the K-Color problem. Note that this is a minimization problem.

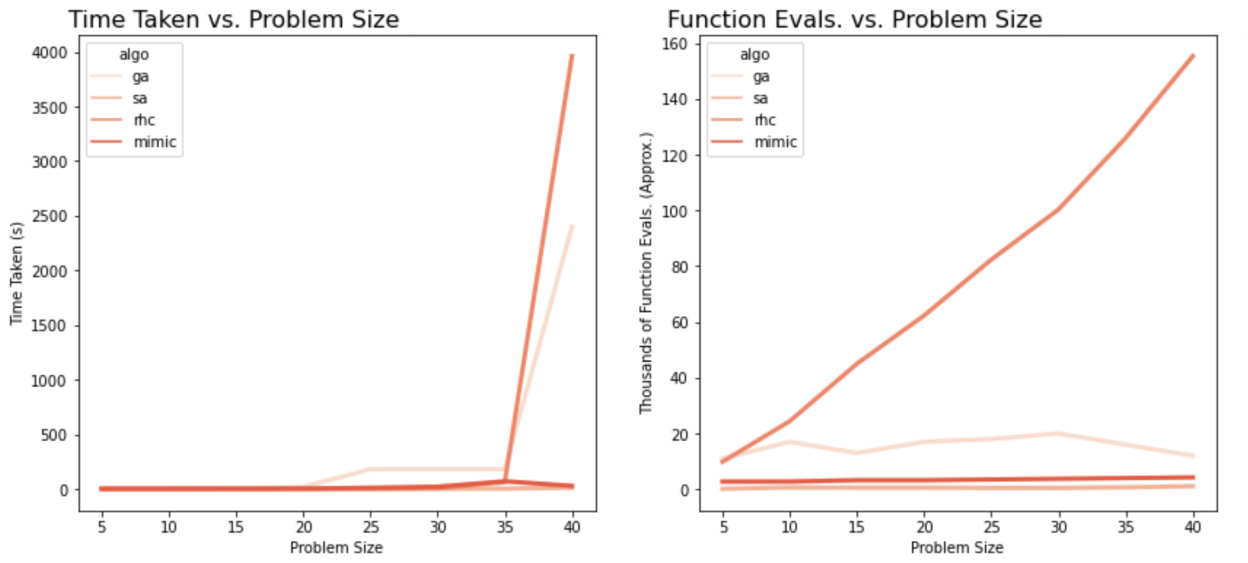

Figure 5: Time to Optimal Value vs. Problem Size (left). Fitness function evaluations vs. Problem Size (right). Both of these on the K-Color graph problem.

3.3 Traveling Salesman Problem

Here we find that GA and SA have the best performance when measured in terms of time to optimal solution (Figure 6). RHC tends to underperform here however, most likely because this problem is NP-Hard, and a brute force approach is not optimal for such a problem. Just as before, we see that MIMIC performs quite well on this problem as a function of algorithmic iterations. It also performed well in terms of function evaluations (Figure 7).

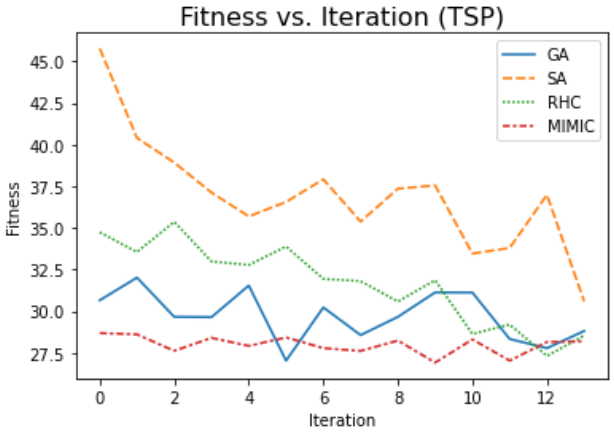

Figure 6: Fitness vs. Iteration on the K-Color problem. Note that this is a minimization problem.

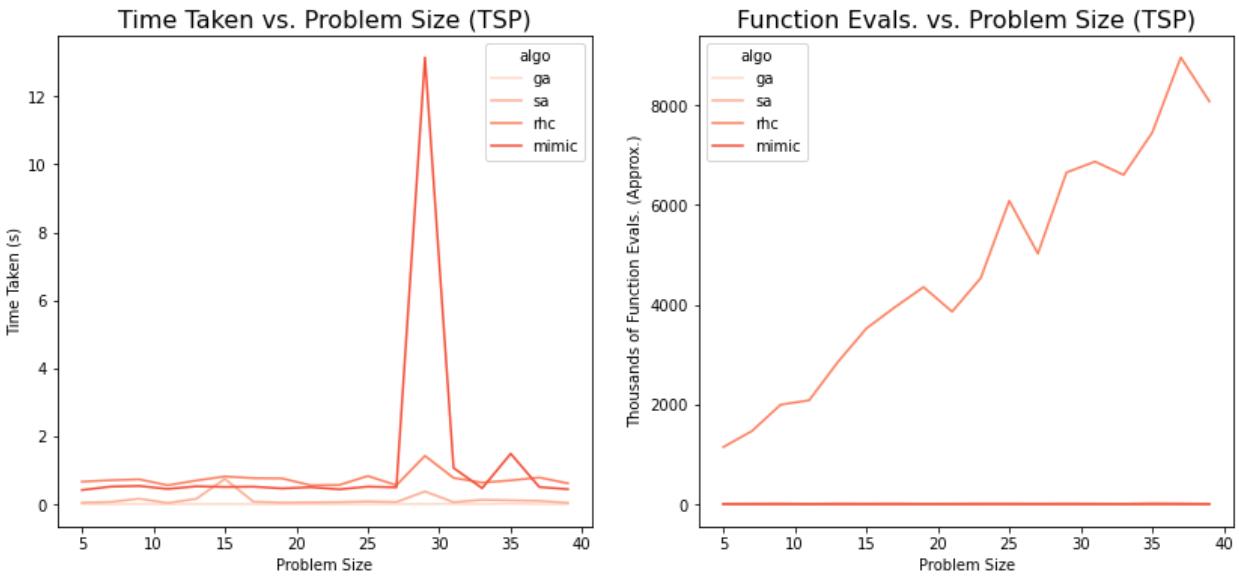

Figure 7: Time to Optimal Value vs. Problem Size (left). Fitness function evaluations vs. Problem Size (right).

3. Randomized Optimization on Neural Net Weights

Four approaches are compared in this section to optimize a single hidden layer neural network with input size of 46 elements, hidden layer size of 30 and output size of 2. The problem used is specifically the Telco Customer Dataset from Assignment 1 and is an unbalanced classification problem (approx. 5:1 for False:True ratio). The four algorithms compared are:

1. Genetic Algorithm
2. Simulated Annealing
3. Random Hill Climbing
4. Backpropagation

Here, the first three techniques for optimization are challenging to make work due to the difficulty of the problem and number of parameters (approximately 1200 parameters). In general, neural networks can be hard to optimize well, and this is with something as effective as gradient descent. In assignment 1, test AUC was 93% using the neural network described here, with a training time of 11 seconds. This will serve as the benchmark for these other optimization algorithms to compare to.

First we will go over some hyperparameter tuning for the optimization algorithms, then we will discuss the results. Starting with GAs (using the open source MLRose), we can test the effect of population size on the results (Figure 8).

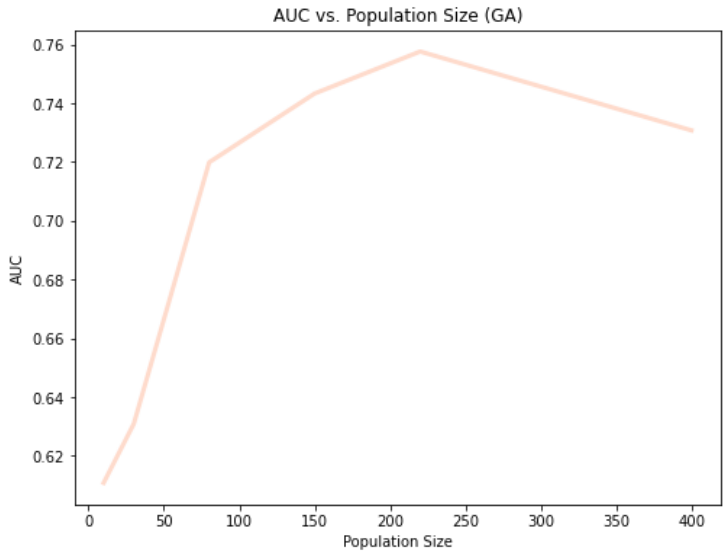

Figure 8: AUC vs. Population Size while optimizing Genetic Algorithm.

It was found that larger populations resulted in better results which makes sense - with a larger population size, we can encourage more diversity and sampling at each step. However, a population size of 200 vs. 400 has a minimal difference. Like in the last assignment, we can also compare the effect of having less data during optimization, finding that more data does help with test set performance (Figure 9).

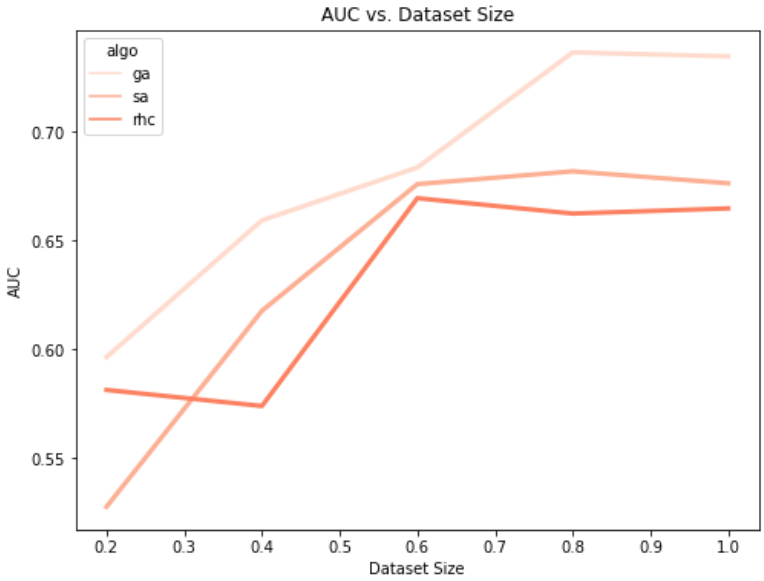

Figure 9: AUC vs. Dataset Size for all three learning algorithms.

There is a diminishing return of more data for all algorithms that must be considered. Next, we can look at how the choice of learning rate affects the optimization results (Figure 10). Though not demonstrated graphically, other parameters experimented with were decay rates for SA to see if more

exploration at the beginning would help with finding better optima, number of restarts for RHC, and the effect of hidden layer sizes on training time (though for the purpose of this assignment, it is left at 30 to compare results with backpropagation). It was found that larger hidden layer sizes massively affected training time, and over 100, they did not converge well. It seems that for a larger neural network (image or text processing) where the network is on the order of 100M+ parameters, these optimization methods would quickly break down.

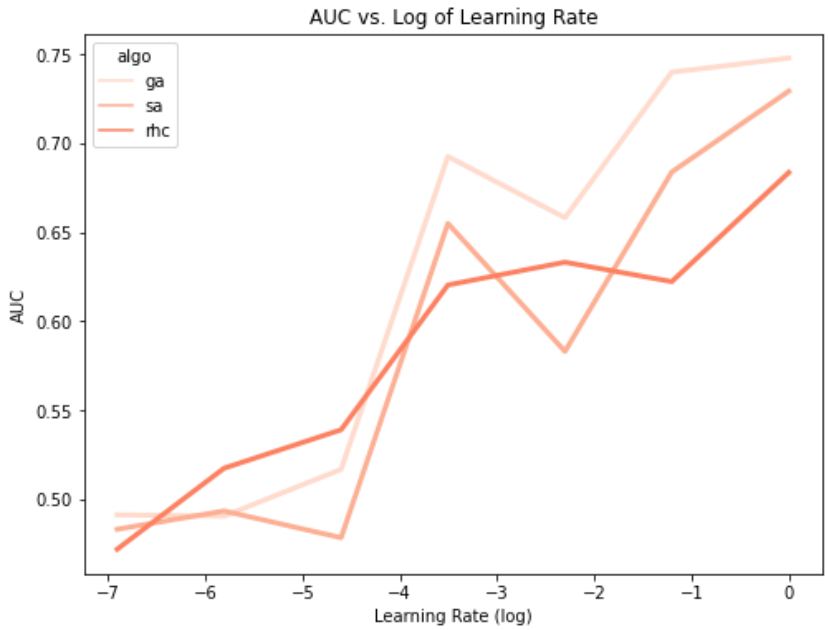

Figure 10: AUC vs. Learning Rate for all three learning algorithms.

Moving on to more tangible results, Table 1 shows each algorithm's test set AUC and training time. Backpropagation outperformed all other optimization methods in terms of both AUC and training time. The next best algorithm was the genetic algorithm, most likely due to its ability to capture some inter-relationships during the mutation and crossover step. Unfortunately, what all of the non-backpropagation algorithms fail to do is understand the mathematical relationship between the output and any given weight being optimized. They make simple checks like - if this is changed, does it improve? But they do not look at the rate of improvement and relate it to specific weights. This is what backpropagation does best. Furthermore, if a problem has 1200 weights and each weight is 32 bits, we have 1200 * 32 potential bits to change. Even with quantization of the input space, we may have 1200 * 8 potential bits. Now the solution space is $2^{(1200*8)}$ which is so large that even Wolfram Alpha refused to give a precise result. This is a huge search space for an algorithm that does not understand structure to explore.

Another drawback of the three non-backpropagation algorithms is that they are generally not as well supported in software as gradient descent is. For example the MLRose library (while wonderful) does not support heavily customizable neural networks. There is no GPU support, heavily vectorized operations, ability to integrate concepts such as the Leaky ReLU activation or BatchNorm

layer. Obviously, one cannot reasonably expect MLRose to do this, but the decision of what algorithm is "best" is often heavily weighted by a parallel question - what algorithm is best supported in software? In this case backpropagation is much better supported by tools like PyTorch, TensorFlow, etc.

In conclusion, backpropagation is probably the best choice for training time, AUC, and software support to train neural networks. It also is well backed mathematically.. However, the other three optimization algorithms are also effective at creating approximate functions given a long enough training time.

Algorithm	Train Time (s)	AUC
Backpropagation	11	93%
Genetic Algorithm	3,802	72%
Random Hill Climbing	1,940	67%
Simulated Annealing	1,292	66%

Table 1: AUC and Training Time as a function of algorithm.

References

1. SKLearn - https://scikit-learn.org/
2. Kaggle - Telco Customer Churn Dataset - https://www.kaggle.com/blastchar/telco-customer-churn
3. MLRose - https://mlrose.readthedocs.io/en/stable/source/intro.html
4. MIMIC - (Dr. Isbell) https://www.cc.gatech.edu/~isbell/tutorials/mimic-tutorial.pdf