Zach Arnold
CS 7641
Assignment 2

## 1 Introduction

In this paper I will be exploring the performance and trade-offs of four randomized optimization algorithms: randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC against three problems. Later in the paper I will reexamine work I did previously in Assigment 1 and attempt to optimize the weights of a multi-layered perceptron neural network using these same four algorithms. For the purposes of this assignment I used the mlrose library [3] which was publicly available. Note that I made some modifications to the library on my computer to be able to generate timing curves (and I do not know how to publish a Python package so I will include instructions for how to get my code in the Readme.)

## 2 Description of Randomized Optimization Algorithms

This section is dedicated to exploring a comparison of four randomized optimization algorithms against example problems which should highlight their relative strenghts and weaknesses. I will briefly introduce parameters I will be varying for each algorithm here and then we will compare their performance in each problems subsection below.

### Randomized Hill Climbing

Randomized hill climbing is perhaps one of the more straightforward randomized optimization algorithms to understand. It begins in a random location in the search space and looks for random neighbors around that point for an optimal value defined by some fitness function. If it succeeds in finding such a point the algorithm terminates unless it is configured to restart from a random place elsewhere in the search space. The hope is that by randomly restarting we can prevent the algorithm from getting "stuck" in a local maximum and find a global maximum. The parameters I am varying are:

- `max_attempts` The number of attempts to find a neighbor with better fitness
- `max_iterations` The number of times to iterations to search (this will usually be lower than `max_attempts`)
- `restarts` The number of times to randomly restart the above search in a new location in a random location in the search space

### Simulated Annealing

Simulated Annealing involves a randomized searching technique that involves dynamically constraining and relaxing the values that the algorithm will consider to be acceptable as "better fit" by using a schedule to simulate "heating and cooling" or "better or worse fit."

The theory is that by varying this schedule of heating and cooling that the algorithm will be able to avoid local maxima by eventually considering worse values as better outside of the space where the local maximum was in search of a possibly better global maximum.

- `max_attempts` The number of attempts to find a neighbor with better fitness
- `max_iterations` The number of times to iterations to search (this will usually be lower than `max_attempts`)
- `schedule` The decay schedule controls the temperature chance for each iteration of the SA algorithm. It can take on three values: Arithmetic, Geometric, and Exponential decay.
  - Arithmetic decay is modeled as $Temp(t) = T_0 - rt$ where $T_0$ is the initial temp and $r$ is some value in (0,1) and represents the rate of arithmetic decay.
  - Geometric decay is modeled as $Temp(t) = T_0 - r^t$. We expect in this scenario the temperature to decrease much more rapidly.
  - Exponential decay is modeled as $Temp(t) = T_0 e^{-rt}$ This may look familiar as it is modeled similar to the continuously compounding interest rate formula and should decay the most quickly.

**Genetic Algorithm**

A genetic algorithm takes the concept biological evolution (particularly the concept of "survival of the fittest") and applies it to randomizes search. A grouping of randomly selected guesses is selected and the best ones are kept, and the worst fitting elements are replaced by random recombination of the elements of of the best performing guesses. In this way the algorithm hopes to find a global maximum or "most fit" population by always searching for better and better random recombinations of the the population. The parameters I am varying for this algorithm are:

- `max_attempts` The number of attempts to find a neighbor with better fitness
- `max_iterations` The number of times to iterations to search (this will usually be lower than `max_attempts`)
- `pop_size` The number of guesses (or population count) to retain at each iteration of the search
- `mutation_prob` The likelihood of an element in a vector to be randomly recombined if it is in a bad population. For example, if this value were 0.1 and the population is a grouping of 10 element vectors, then most likely 1 of the elements will be recombined if it is in the "worse performing" group at the end of a "generation" (iteration of the algorithm).

**MIMIC**

MIMIC expands on the concept of Genetic Algorithms combining the best of each generation and replacing the worst in the search with the concept of "memory" of previous generations through the use of probability distributions and information theory as modeled by Kullback-Libeler divergence. [4] Thus it should be able to converge much faster to an optimal solution than a genetic algorithm for the same problem. The parameters I will be varying for this algorithm are:

- `max_attempts` The number of attempts to find a neighbor with better fitness
- `max_iterations` The number of times to iterations to search (this will usually be lower than `max_attempts`)
- `pop_size` The number of guesses (or population count) to retain at each iteration of the search
- `keep_pct` The percentage of guesses to retain between each iteration of the algorithm.

## 2.1 One Max Problem; highlights advantages of MIMIC

The One Max Problemis a fitness function that seeks to maximize a vector $v$ such that:

$$\sum_{i=0}^{n-1} v_i \tag{1}$$

is a maximum, where $v_i$ is the $i$th component of vector $v$ with length $n$. This is a relatively easy optimization problem to understand and it illustrates the effectiveness of MIMIC quite nicely from an efficiency standpoint. The best values found for each algorithm were as follows:

| Algorithm | Parameters | Value |
|---|---|---|
| RHC | Max Iterations, Max Attempts, Restarts | 40, 40, 0 |
| SA | Max Iterations, Max Attempts, Decay Schedule | 180, 180, Exponential |
| GA | Max Iterations, Max Attempts, Pop Size, Mutation Prob | 10,10,100,0.1 |
| MIMIC | Max Iterations, Max Attempts, Pop Size, Keep % | 10,10,0.1,300 |

As you can see, MIMIC converged to the correct optimal value within 10 iterations of the function making it a very powerful algorithm for this particular problem type.

Figures 1 and 2 illustrate how quickly MIMIC converged and that it took less wall clock time to do so than Genetic Algorithms (a similar class of algorithm.) This result is most likely due to the fact that while MIMIC requires time to calculate probability density functions, by using this to improve its guess each time it saw the most dramatic improvement

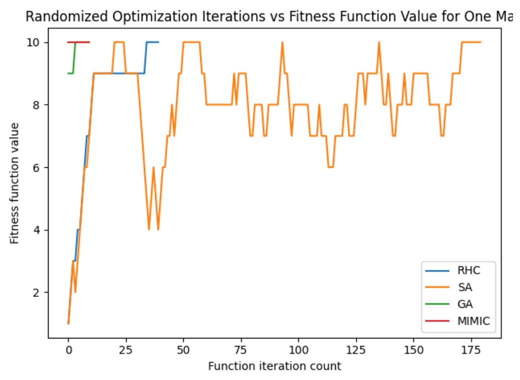

Figure 1: One Max Iterations vs Function Value

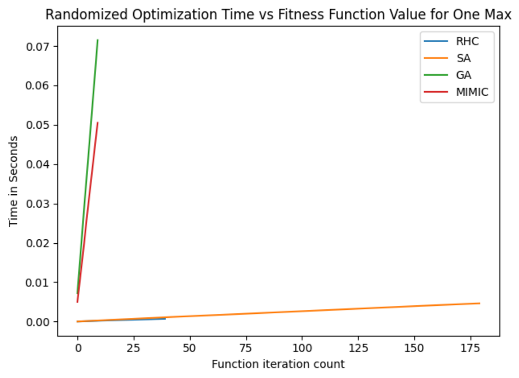

Figure 2: One Max Iterations vs Time

between iterations. By contrast Simulated Annealing took over 175 iterations to converge to the the correct value. The reason for this is due to the schedule of decay of SA. Since this problem has only one best value for any n dimensional input (a vector containing all 1's,) by considering worse values as potentially better (in order to to avoid local maxima). The same could be said for randomized hill climbing. Given that it will randomly restart in a different location it could potentially sabotage its own search for a maximum because the solution has only one max. The best performing class of algorithms were GA's and MIMIC for this problem, with MIMIC being a clear winner in terms of number of iterations required to find the optimal value. Because GAs/MIMIC are predisposed to favor well fitting values from previous generations to create a new generation of values this problem highlights this strength. Since there is only one max and by flipping one bit each time from a 0 to a 1 increases fitness, each algorithm would do successively better with each iteration. MIMIC did especially well because it could sample from a probability distribution of good values to create the next generation increasing the speed (in terms of iterations) of converging to the optimal value. It is worth noting however from a wall-clock time perspective, in general MIMIC performs worse than Genetic Algorithms because of the time required to recalculate $p^{\theta_i}(x)$ for the next generation as given by the definition of $p^{\theta_i}(x)$. [4]

## 2.2 8 Queens Problem; highlights advantages of Simulated Annealing

The eight queens problem is a specific implementation of the n-queens optimization problem. [5] It poses an $nxn$ board like in chess where $n$ queens need to be placed such that a minimum number of queens could "attack" each other (diagonally, horizontally, or vertically.) So the space to search is a collection of vectors of 8 elements where each element is an integer between 0 and 7 and represents where the queen is on that column of the chess board. As illustrated below, this optimization problem favors the simulated anneal-

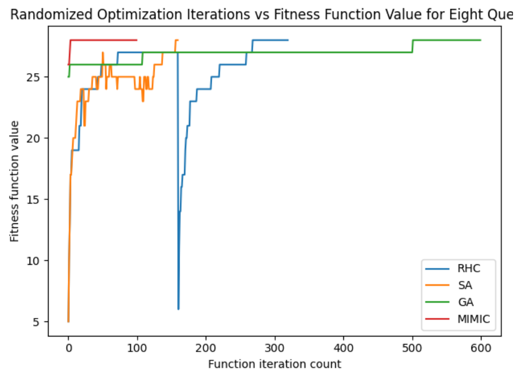

Figure 3: Eight Queens Iterations vs Fitness Function Value

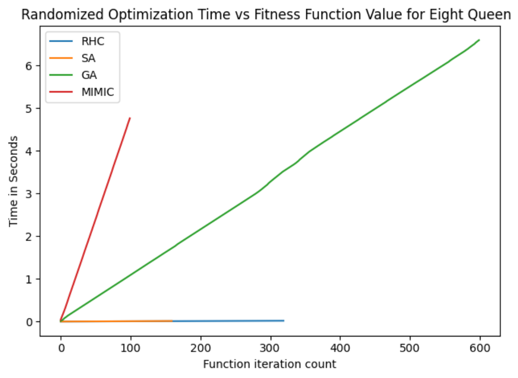

Figure 4: Eight Queens Iterations vs Time

ing algorithm implementation approach I have chosen in terms of function iteration count required to achieve an optimal placement. The best values found for each algorithm were as follows:

| Algorithm | Parameters | Value |
|---|---|---|
| RHC | Max Iterations, Max Attempts, Restarts | 160, 160, 1 |
| SA | Max Iterations, Max Attempts, Decay Schedule | 160, 160, Exponential |
| GA | Max Iterations, Max Attempts, Pop Size, Mutation Prob | 600,600,100,0.1 |
| MIMIC | Max Iterations, Max Attempts, Pop Size, Keep % | 100,100,800,0.1 |

The 8-Queens problem favors the Simulated Annealing algorithm I used (and somewhat the RHC algorithm as well) due to the nature of the problem. Because there are at maximum 92 solutions that are valid (derived from 12 fundamental solutions,) [1] and those maximal fitness values are dispersed throughout the space of all possible board states, algorithms like GA and MIMIC are at a disadvantage, because permuting random sections of a partially-fit vector do not necessarily equate to a strictly better fit vector. However in the case of SA and RHC which search for any maximal value through neighbor searching, these algorithms can directly compute an answer quickly often with an aggressive decay schedule since there are 92 best values each of which have the same fitness function value. In terms of wall clock time, Simulated Annealing did the best with a near zero wall clock time and only 160 function iterations to achieve the best result. While MIMIC did have a lower function iteration count required to get the best value, it used a population size of 800 to achieve this and at a very steep cost in terms of computation time. Looking at figure 4, you can see that while Genetic Algorithms took the longest, MIMIC grew the fastest with respect to function iteration count, meaning that if we were to optimize an n-queens problem ($n > 8$), the lower iteration count would stop being a benefit in terms of wall-clock time quickly.

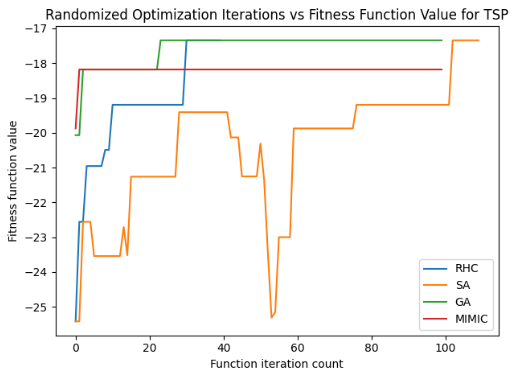

Figure 5: TSP Iterations vs Fitness Function Value

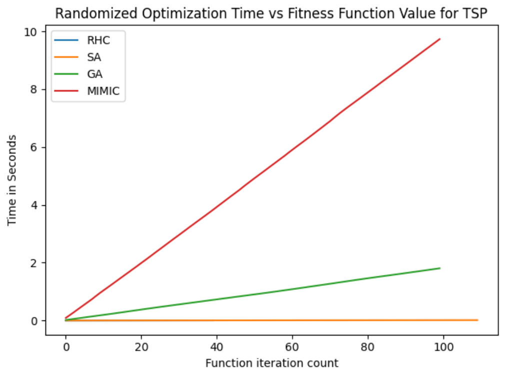

Figure 6: TSP Iterations vs Time

## 2.3 Traveling Salesman; highlights advantages of Genetic Algorithms

The Traveling Salesmanis defined by mlrose's documentation in the following way: "The travelling salesperson problem (TSP) is a classic optimization problem where the goal is to determine the shortest tour of a collection of n 'cities' (i.e. nodes), starting and ending in the same city and visiting all of the other cities exactly once. In such a situation, a solution can be represented by a vector of n integers, each in the range 0 to n-1, specifying the order in which the cities should be visited." [3] This problem is similar to the 8 Queens Problemin that the vector represents an ordering of positions. In the 8 Queens Problemthey are board postions but in Traveling Salesmanthey represent an order of cities to visit. Below is the table of optimal values for each function and the iteration and timing curves.

| Algorithm | Parameters | Value |
|---|---|---|
| RHC | Max Iterations, Max Attempts, Restarts | 40, 40, 0 |
| SA | Max Iterations, Max Attempts, Decay Schedule | 110, 110, Exponential |
| GA | Max Iterations, Max Attempts, Pop Size, Mutation Prob | 100,100,100,0.1 |
| MIMIC | Max Iterations, Max Attempts, Pop Size, Keep % | 100,100,200,0.1 |

As you can see, the best all around algorithm in terms of speed and iteration count is Genetic Algorithms. While Simulated Annealing was faster in terms of wall clock time, Genetic algorithms got to the convergence value in a much shorter ($> 80$) number of iterations. Simulated Annealing and RHC would have the same advantages over GA and MIMIC in terms of their ability to search neighboring solutions for a max value easily as they did in 8-queens, the difference here is in the problem statement. Since the Traveling Salesmanis about minimizing the number of revisits to any point, it can be thought of as a constrained One Max problem (in that there is a best solution with the fewest number of revisits and there is only one global min.) In this way, the problem is like a combination of One Max and 8 queens. Given that, Genetic Algorithms ability to rapidly recombine better and better solutions allowed it to converge quickly to the correct value. The difference here is, whereas MIMIC is a variant of GA and in general performs better in the number of function iterations required to solve the same problem, in this instance MIMIC never converged to the globally minimal solution and even the sub optimal path ordering it proposed took 5x the amount of time as the nearest algorithm.

## 3 Neural Network Weight Optimization

In this section I will be applying the algorithms from above on my original assignment 1 dataset 1 (Classifying whether or not someone makes less than $50k in a year in America) and attempt to optimize a Neural Network's weights. The goal is to replace backpropagation as the method of NN training with RO. As was the case with the initial training I will be doing the following:

- Splitting the data into training and validation sets
- Optimizing for Jaccard Accuracy Score [2]

If you'll recall from the previous assigment, the NN did the worst out of all of the supervised learning algorithms from the previous section. This was due to the networks hidden layer construction and the fact that this problem did not lend itself well to any of the activation functions supplied to backpropagation. The best overall accuracy achieved was 75.14% in total.

## References

[1] Eight queens puzzle. `https://en.wikipedia.org/wiki/Eight_queens_puzzle`, Aug 2020.

[2] Jaccard index. `https://en.wikipedia.org/wiki/Jaccard_index`, Aug 2020.

[3] G Hayes. mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. `https://github.com/gkhayes/mlrose`, 2019. Accessed: 20 September 2020.

[4] Paul Viola Jeremy De Bonet, Charles Isbell Jr. Mimic: Finding optima by estimating probability densities. `https://www.cc.gatech.edu/~isbell/papers/isbell-mimic-nips-1997.pdf`, 1997.

[5] S. Russell and P Norvig. Artificial intelligence: A modern approach, 3rd edition. Prentice Hall, New Jersey, USA., 2010.