CS7641 – Machine Learning

Assignment 2 – Randomized Optimization

Mason J. Kelchner  
mason.kelchner@gatech.edu

# 

# INTRODUCTION

The purpose of this report and the analyses presented is to explore random search optimization algorithms and their application to supervised machine learning techniques. The random optimization algorithms investigated include random hill climbing (RHC), simulated annealing (SA), genetic algorithms (GA), and MIMIC [1]. Two experiments are performed for these random optimization methods. The first experiment used each algorithm to find solutions to three varying complexity discrete combinatorial problems including: Travelling Salesman (TSP), Random Bit Matching (RBM), and Flip Flop (FF). These optimization algorithms and problems were implemented using custom functions and modules as well as the *mlrose-hiive* implementations [2] [3]. Other optimization problems were investigated including the Knapsack optimization problem and One Peak optimization; however, for the purpose of highlighting the advantages of simulated annealing, genetic algorithms, and MIMIC these were not included. The second experiment compared the performance of random hill climbing, simulated annealing, and genetic algorithms with varying hyperparameters on the optimization of the weights in a neural network in place of backpropagation. Performance of these various algorithms was determined by looking at a combination of factors including the accuracy of predictions from the neural network formed from the weights set by each algorithm, the F1 score (utilizing the python module sklearn.f1\_score), and the wall clock time required to solve the optimization of weights [4]. Various activation functions, hidden nodes, and other hyperparameters were tuned for each algorithm to explore the effects of each on the performance of the optimization algorithms. For this second experiment, a mushroom classification data set was used with 20 features and 61,069 instances classifying mushrooms either as edible (e) or poisonous (p). Each experiment and all data preparation will be explained in the following sections.

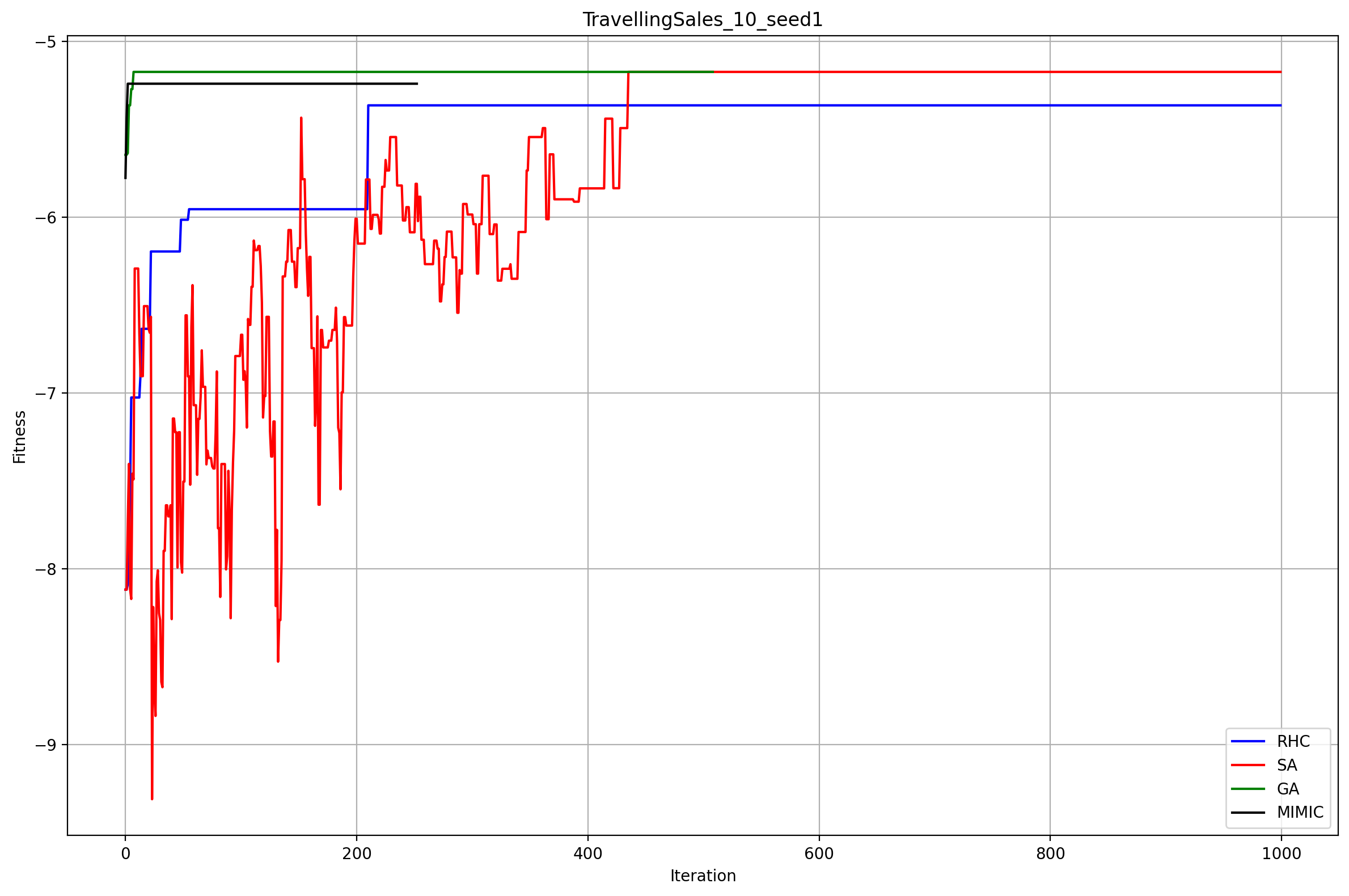
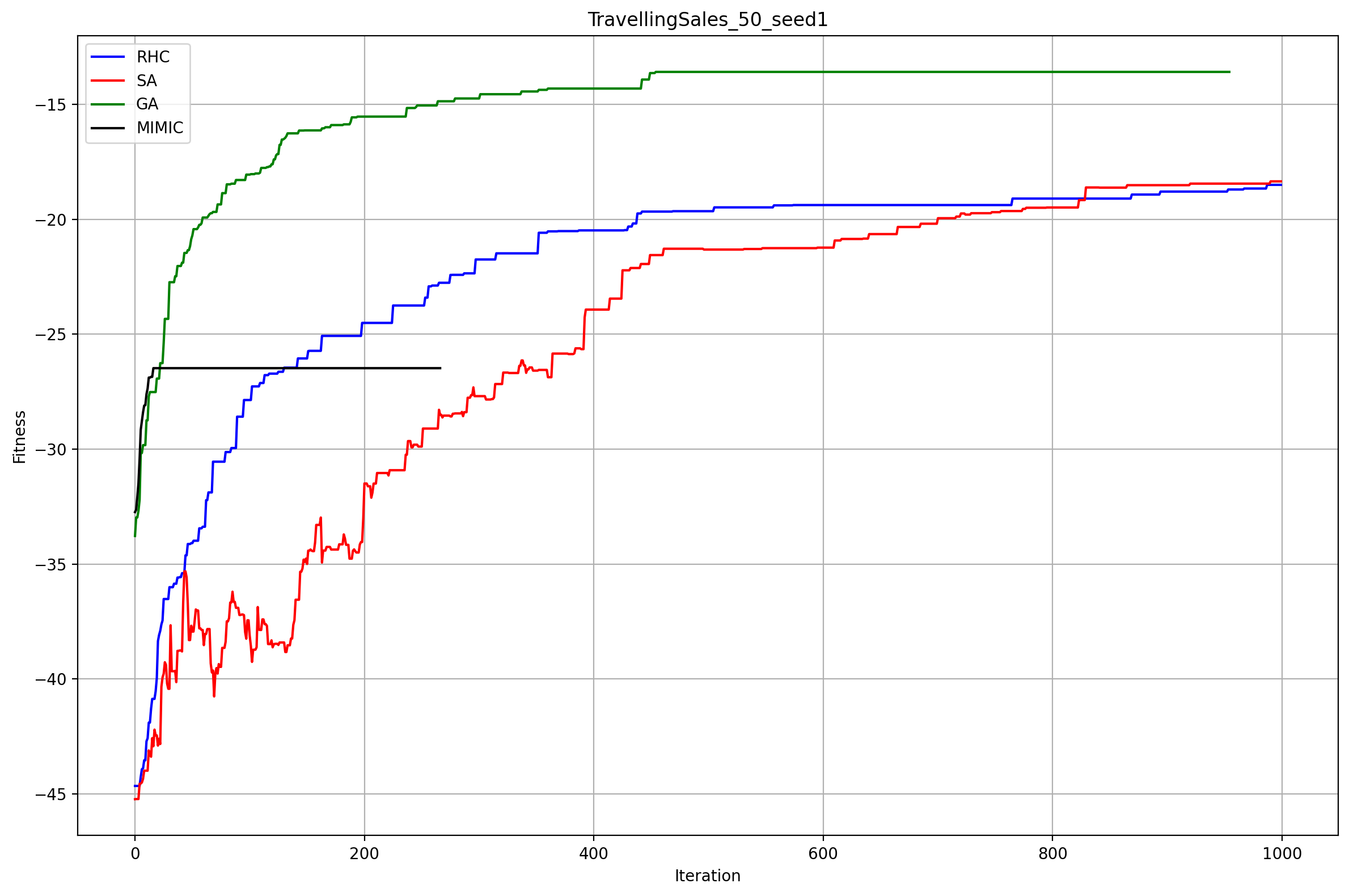
# Randomized Optimization Experiments

Each of the following optimization problems were selected to highlight advantages and disadvantages of each of the random optimization algorithms introduced earlier. It is important to note that each problem is a discrete value optimization problem as the state features are discrete integer values. Additionally, all problems are constructed as maximization optimization problems for consistency. Each problem has real world relevance and give a more concrete relationship between the importance of these random optimization algorithms and machine learning techniques and their applications to real problems. The format of the following problems are simplified from their real world applications though they contain enough complexity for discussion on the advantages and disadvantages of each algorithm. Finally, the performance of the algorithms on each problem is evaluated as the space complexity of the problem varies between small, medium, and larger parameter spaces. This is meant to illustrate how each algorithm performs as the problem grows in complexity since the performance of one algorithm could be significantly affected by the space complexity depending on the problem.

## Travelling Salesman Problem (GA is best)

The travelling salesman problem (TSP) is a famously difficult problem to solve optimally as it is NP-Hard and there is no polynomial time solution. The optimal solution for the TSP produces the shortest length (optionally weighted) traversing all cities. Some formulations of the problem include the stipulation that the final city must be the initial city as well. This problem is well suited for representation as a graph structure with cities represented as nodes and the routes between them as weighted edges. It has extremely important real world applications most notably in package delivery and vehicle route planning. The brute force method for this problem grows as a factorial of the number of cities and their connections (edges) i.e. the solution space is O(n!) for the TSP. Randomized optimization techniques find generally good solutions to TSPs; although, they are all subject to failing to converge at the global optimal solution by converging to locally optimal solutions especially in strongly connected graphs of TSPs.

A problem was created by randomly generating a connected graph utilizing the python *networkx* library and using the erdos\_renyi\_graph function with a 50% probability of connecting each node to another for three size solutions: 10,50, and 100 city solutions [5]. The weights of the graph were set as 1.0 for a simplification of the problem. This generated a graph with 2,396 edges. Each of the previously mentioned randomized optimization algorithms were then formulated using the open source python module *mlrose-hiive* and evaluated on their performance determining a minimizing path for the problem. Minimization optimization was done by negating the fitness function used and thus maximizing the fitness function i.e. maximizing a negative number. The performance of each randomized optimization algorithm including the fitness generated by each algorithm at every iteration is shown in Figure 2‑1. Additionally, the run time for the 100 city problem is shown in Figure 2‑2. The relative run time for each algorithm remains the same between problem sizes, the magnitude scales with the complexity of the problem.

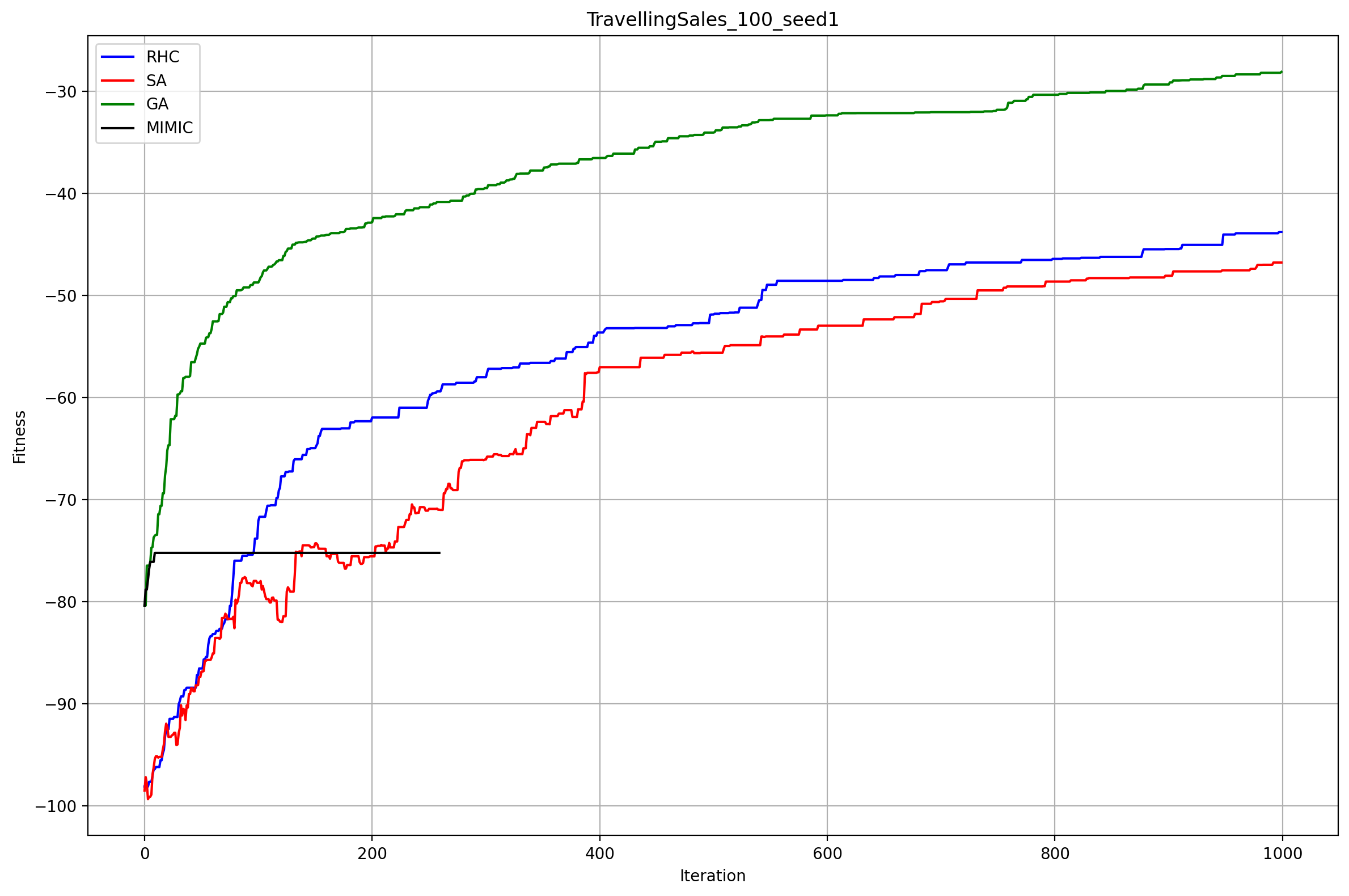


Figure 2‑1 Performance comparison for the 10, 50, and 100 city Travelling Salesman Problem between each random optimization algorithm

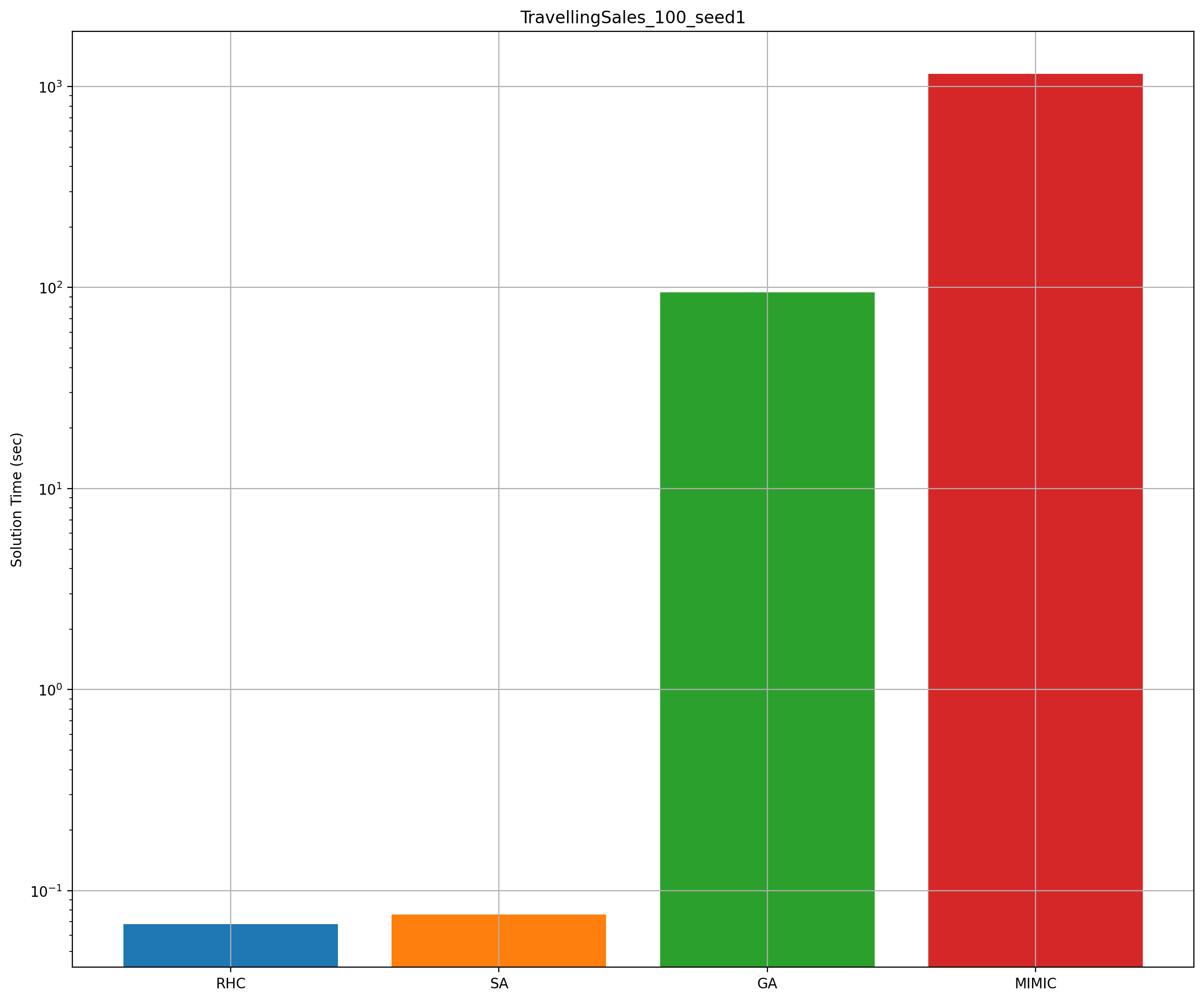


Figure 2‑2 Wall clock run time comparison for the 10, 50, and 100 city Travelling Salesman Problem between each random optimization algorithm on a log-scale in the y-axis

## Random Bit Matching Problem (Simulated Annealing is best, beats GA on time)

The second optimization problem formulated was a random bit matching problem. For this problem, a randomly generated bit string is generated. For this problem, a bit string length of 1,000 was selected. Each bit was an integer 0, 1, 2, or 3 not just 0 or 1 to increase the problem complexity and more accurately replicate the real world application of this problem described later. Various smaller bit string lengths were tested to determine if the length of the bit string significantly affected the relative performance of each algorithm in finding an optimal solution; however, it was determined that the length of the bit string did not change the relative performance of each algorithm. For this problem, the fitness function was a custom function which rewarded successively matching bits more than matching individual bits for each algorithm. For example, if the bit string to match called the *mask* is 01001, a 5 bit length string, the maximum fitness is 1+2+3+4+5 = 15. If an algorithm produces a state of 01101, the fitness of this would be 1+2+0+1+2 = 6. Thus, each success bit that is correctly matched rewards increasingly more fitness. Incorrectly matching a single bit, resets the cumulative fitness back to 0. Thus, the fitness function has a single optimal peak, but can entangle the randomized hill climbing and simulated annealing algorithms due to the importance of order whereas the genetic and MIMIC algorithms perform better in this problem due to the structure of the problem.

This problem is interesting as it could be applied to genetic testing. Imagine a scenario where a geneticist is trying to align and match specific gene sequences within a larger genomic dataset. These gene sequences can be very long, many orders of magnitude larger than in this example. Each gene can be represented as a bit string, where each bit represents a nucleotide (e.g., adenine, thymine, cytosine, guanine). The task is to find the best alignment of a target gene sequence (the "mask" in this problem) within a pool of genomic sequences. The fitness function in this case could evaluate how well a given sequence aligns with the target gene sequence. Successively matching bits (representing nucleotides) in the sequence to be aligned with the target gene would receive higher fitness scores, reflecting the biological significance of consecutively matching nucleotides. Different algorithms, such as randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC, can be applied to optimize the alignment process and find the best match for the target gene sequence within the pool of genomic data. Each algorithm's performance can be evaluated based on how well it aligns the sequences and maximizes the fitness function, similar to the described random bit matching problem. The goal is to effectively and accurately align genes and understand their structures and functions within the context of the larger genomic dataset.

As with the previous problem in section 2.1, each random optimization problem was tuned iteratively and applied to the stated problem and evaluated according to the fitness function previously described. The algorithms’ fitness for each iteration is shown in

Figure 2‑3 Performance comparison for the 1000 bit length Random Bit Matching problem between each random optimization algorithm

## Flip Flop Problem (MIMIC performs best)

# Randomized Optimization For Neural Network

The second experiment, utilized three of the random optimization algorithms: random hill climbing, simulated annealing, and genetic algorithm. For this experiment, a mushroom classification data set was used to fit a neural network for prediction based on 20 features whether an instance of a mushroom was poisonous (p) or edible (e). However, in place of backpropagation to update the network weights during each iteration of training, the random optimization algorithms will be used to iteratively determine an optimal weight state to maximize the neural network performance. Here the optimization problem is created as a continuous optimization problem by representing all the nodes including the input, output, and hidden nodes as an array forming the state in the optimization problem. Each of the optimization algorithms are then used to determine an optimal weight matrix for the neural network for a variety of hyperparameter configurations. Hyperparameters that were varied include the number of nodes in each hidden layer, the number of hidden layers, the activation functions, the population size for the genetic algorithm, the learning rate for the random algorithms, the number of iterations to run, the mutation probability for the genetic algorithm, and the loss functions when determining the fitness of the predicted values for each iteration for each algorithm. The loss functions evaluated include the mean squared error, the log loss error (or cross entropy error), and the mean squared log error. The various activation functions used include the *identity, relu, sigmoid*, and *tanh* functions.

In the realm of optimizing neural network weights, a critical endeavor in modern machine learning, the choice of optimization algorithm is paramount. When dealing with continuous and real-valued weights, as opposed to discrete ones, algorithmic adaptations become necessary. Classical optimization methods like simulated annealing, random hill climbing, and genetic algorithms require appropriate representation schemes for continuous weights, utilizing floating-point encoding. The vast continuous search space necessitates careful parameter tuning to facilitate effective exploration and exploitation. However, these algorithms lack the advantage of gradient information utilization, a strength of backpropagation, and may converge slower or suboptimally. Additionally, they often face challenges in handling the high-dimensional continuous space efficiently. In contrast, gradient-based optimization methods, specifically tailored for continuous optimization, such as backpropagation, generally exhibit faster convergence and efficiency in optimizing neural network weights. Thus, selecting the appropriate optimization approach necessitates a careful consideration of the problem's characteristics, including the nature of weights, the dimensionality of the search space, and computational efficiency requirements.

The mushroom classification data set was preprocessed to encode the labels for each feature using integers using the sklearn preprocessing class *LabelEncoder*, these integers were then scaled across the data set to values between 0 and 1 using the sklearn preprocessing class *MinMaxScaler.* This preprocessing improves the performance and convergence of the optimization algorithms. No additional data reduction or feature selection was performed on the data set as the data set was relatively balanced between instances of poisonous and edible mushrooms samples and each feature was found to contribute in some meaningful way to the classification of each. This was shown by evaluating the information gain by training a simple decision tree on the data set which found a relatively uniform importance weight between each feature when classifying a the test data set.

The performance of each algorithm was evaluated as a combined quantification of the resulting neural network accuracy score against a training set of the data, set aside prior to training, the F1 score, and the wall clock fit time of the model. The accuracy score was adjusted to weight correctly identifying poisonous mushrooms as poisonous (i.e. true negatives) by penalizing falsely identifying poisonous mushrooms as edible (i.e. false positives). The training data set used when iteratively fitting the network weights was created from a randomly chosen subset of the whole data set containing 80% of the instances while the test data set contained the remaining 20% of the data.

# Summary

# References

|  |  |
| --- | --- |
| [1] | J. De Bonet, C. Isbell and P. Viola, "MIMIC: Finding Optima by Estimating Probability Densities," *NIPS,* pp. 424-430, 1997. |
| [2] | G. Hayes, "mlrose: Machine Learning, Randomized Optimization and SEarch package for Python," https://github.com/gkhayes/mlrose. |
| [3] | A. Rollings, "). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python, hiive extended remix," https://github.com/hiive/mlrose. |
| [4] | F. Pedregosa, Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011. |
| [5] | A. A. Hagberg, D. A. Schult and a. P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," *Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds),,* pp. 11-15, 2008. |