CS7641 – Machine Learning

Assignment 2 – Randomized Optimization

Mason J. Kelchner
mason.kelchner@gatech.edu

1 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.

2 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.

2.1 Travelling Salesman Problem

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. For scale, one of the large graphs generated contained 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 averaged across 3 runs with different random seeds. This shows the stability in the solutions generated by the algorithms. The relative run time for each algorithm remains the same between problem sizes, the magnitude scales with the complexity of the problem.

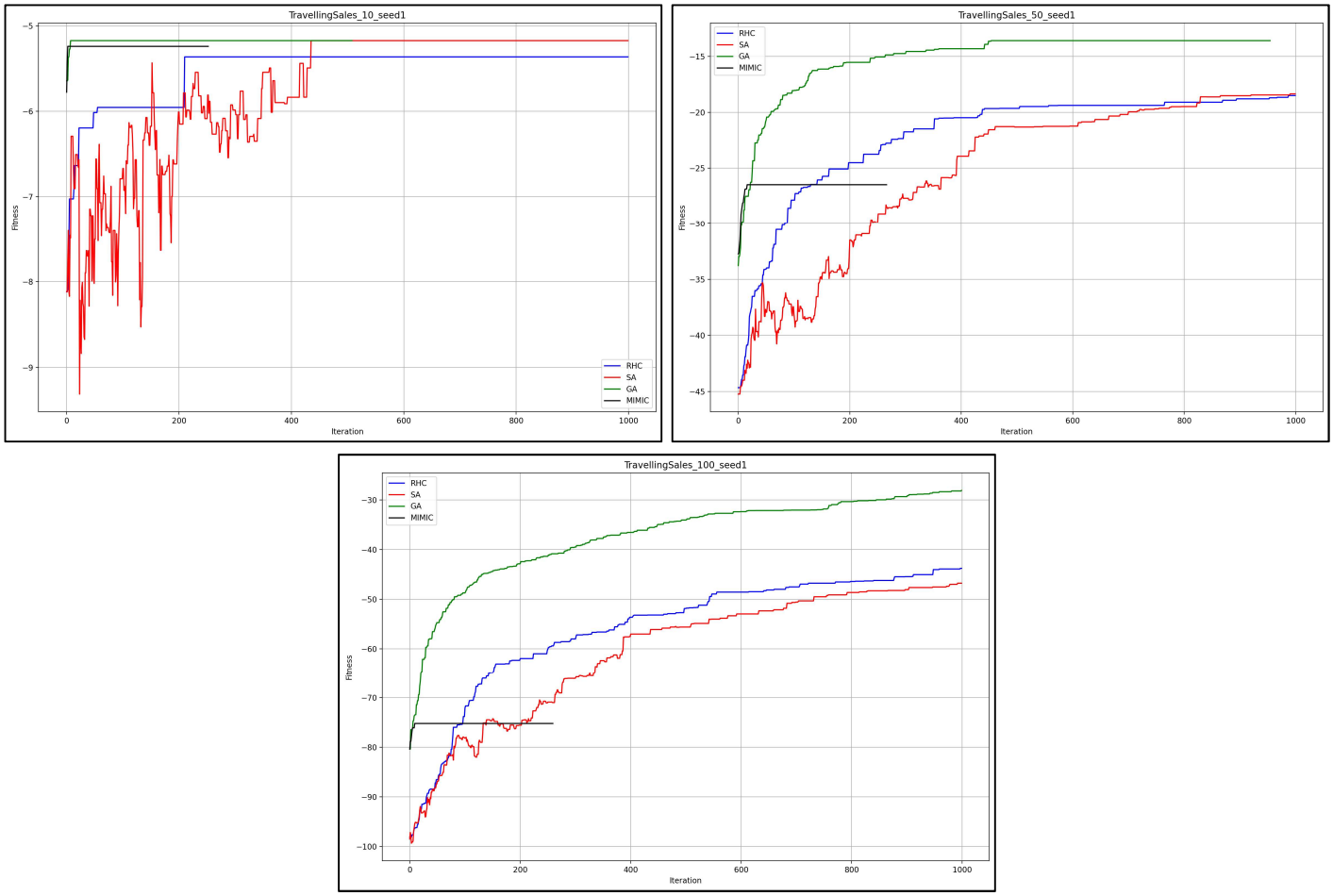

Figure 2-1 Performance comparison for the 10, 50, and 100 city Travelling Salesman Problem between each random optimization algorithm

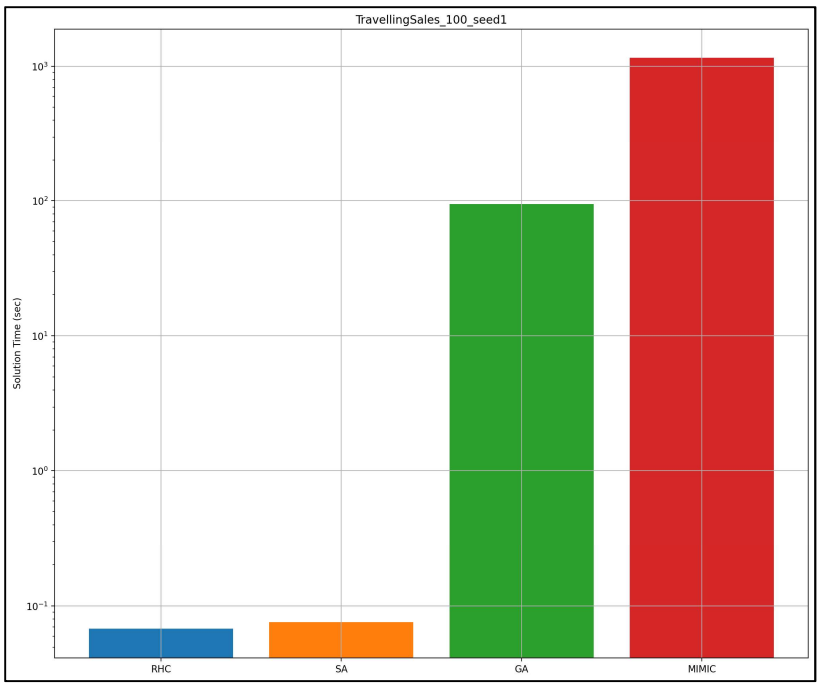

Figure 2-2 Wall clock run time comparison for the 100 city Travelling Salesman Problem between each random optimization algorithm on a log-scale in the y-axis averaged across 3 random seed runs

It is evident from Figure 2-1 that for more complex (larger) TS problems, the genetic algorithm is best suited for this problem. The algorithm consistently performs the best relative to the other algorithms. The performance of the genetic algorithm was found to be dependent on the population size, mutation probability, and the parameters controlling the probability of breeding and crossover. While simulated annealing and random hill climbing perform well for smaller problems, they lack were found to find local non-global optimal solutions to the problem. On the other hand, MIMIC performed the worst among the algorithms both in terms of the performance of the solution found and the run time. MIMIC requires significant fine tuning of the hyper parameters used in a particular problem; however, despite improving state estimation rapidly in the first few iterations regardless of problem complexity, MIMIC would often converge to a local optimum. The genetic algorithm performed well in this problem likely due to the relationships between similar paths where parts of each were in the optimal path while neither optimal themselves. As the iterations continued the genetic algorithm was able to breed new paths formed from smaller overlapping sets of paths which are part of the optimal path. While the genetic algorithm performed better than RHC, SA, and MIMIC for fitness, it was on the order of MIMIC for run time. The genetic algorithm would benefit from any performance improvement for speed including importance sampling when mutating and parallelization.

2.2 Random Bit Matching Problem

The second optimization problem formulated was a random bit matching problem. For this problem, a randomly generated bit string is generated with a length of 50, 100, and 200. 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 and larger 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 this was not the case though the problem sizes are compared and contrasted for completeness. For this problem, the fitness function was a custom function which rewarded successively matching bits for both directions 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 going from left to right: (1+2+3+4+5) + going from right to left: (1+2+3+4+5) = 30. If an algorithm produces a state of 01101, the fitness of this

would be going from left to right: (1+2+0+1+2) + going from right to left: (1+2+0+1+2) = 12. Thus, each success bit that is correctly matched rewards increasingly more fitness. Incorrectly matching a single bit, resets the cumulative fitness back to 0. There is an underlying structure to the problem that would tend to be ideal for MIMIC and genetic algorithms, especially since state parameters are related by sequence, and going into this experiment it was believed they would outperform SA and RHC in terms of fitness.

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, and guanine). This is why the problem was formulated to generate integers between 0 and 3 to represent each nucleotide in a gene. 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. 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.

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. For each problem size, all algorithms were set with a random seed. The algorithms' fitness for each iteration is shown in Figure 2-3 for each problem size. The algorithms' wall clock run time for the large problem averaged across each random seed is shown in Figure 2-4.

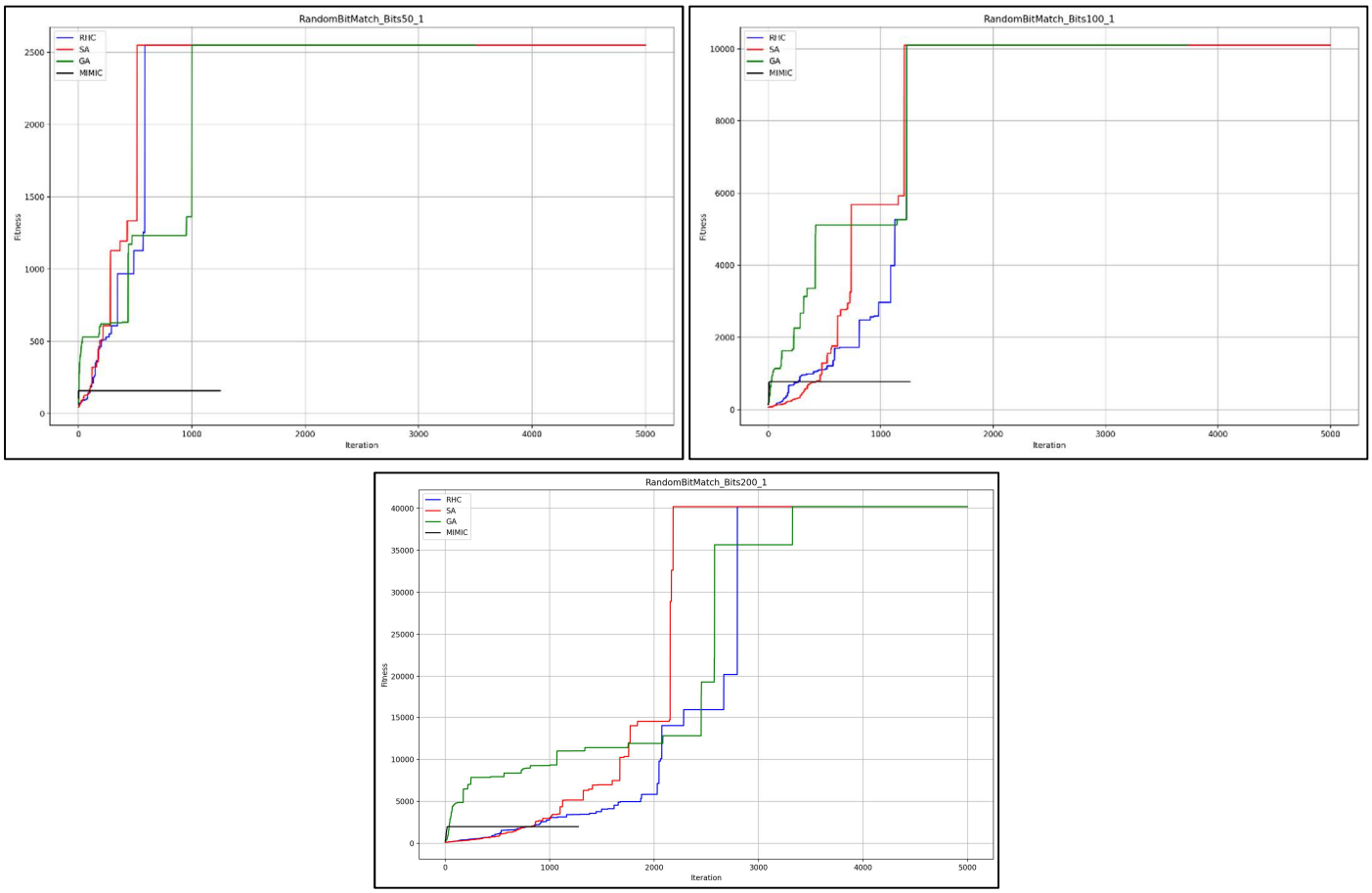

Figure 2-3 Performance comparison for the 50, 100, and 200 bit length Random Bit Matching problem between each random optimization algorithm

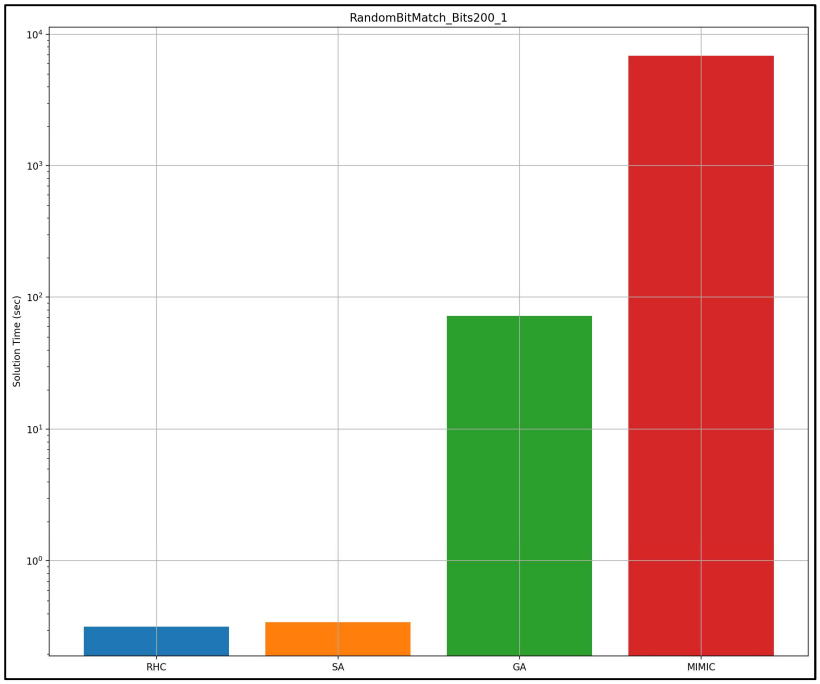

Figure 2-4 Wall clock run time comparison for the 200 bit length Random Bit Matching Problem between each random optimization algorithm on a log-scale in the y-axis averaged across 3 random seed runs

As shown in Figure 2-3 and Figure 2-4, the simulated annealing algorithm performs both well in terms of fitness of the estimated state and the run time. While the genetic algorithm performs equally as well in terms of fitness, the GA takes orders of magnitude longer to each the solution for this problem. Simulated annealing for this problem shows robust and strong convergences relatively quickly. Simulated annealing and RHC perform well in this problem despite initially designing the problem for MIMIC and the genetic algorithm. MIMIC on the other hand performs the worst in terms of both fitness and run time because of the significant time cost associated with each iteration of the algorithm through probabilistic estimations.

2.3 Flip Flop Problem

The third optimization problem was a classic and simple problem where the random optimization algorithms are to maximize the number of alternations in a bit string of a specified length, classically known as the Flip Flop problem. For example, the state [0,1,0] would evaluate a fitness of 2 (2 alternations) while the state [0,1,0,1] would evaluate 3, and [0,0,0,0] would evaluate to 0. For bit string lengths of 100, 200, and 300 bits long, each random optimization algorithm was run for 3 different random seeds, like the other experiments, in order to determine the effect of randomization and problem size on the performance of each algorithm.

This problem can be a representative optimization problem for scheduling tasks where the goal is to maximize the number of tasks in a given amount of time or for planning events in a sequence of events in order to maximally achieve a result. While the setup and structure of this problem is simple, it helps illustrate some benefits and drawbacks of each algorithm.

The algorithms' fitness for each iteration is shown in for each problem size. The algorithms' wall clock run time for the large problem averaged across each random seed is shown in Figure 2-6.

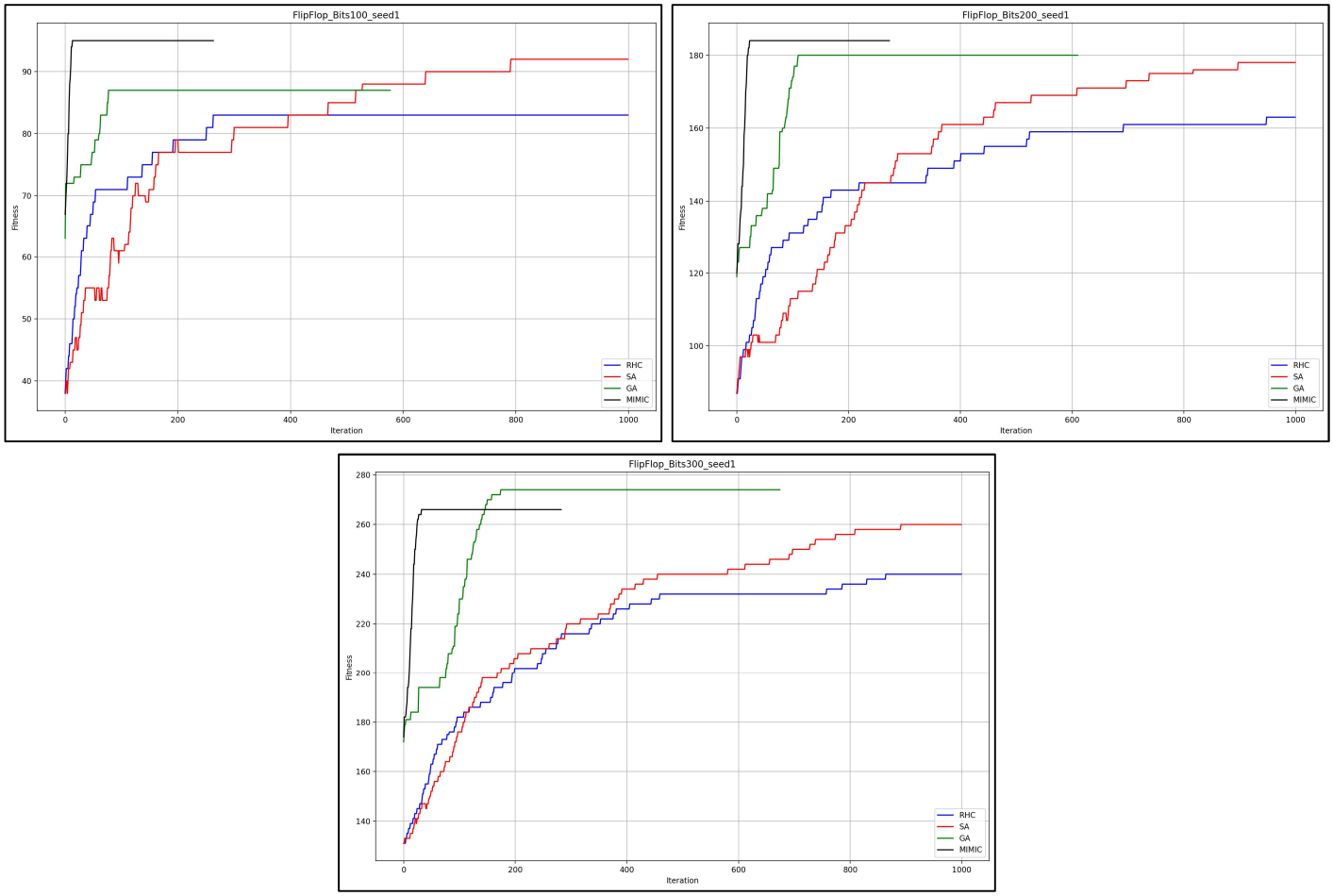

Figure 2-5 Performance comparison for the 100, 200 and 300 bit length Flip Flop problem between each random optimization algorithm

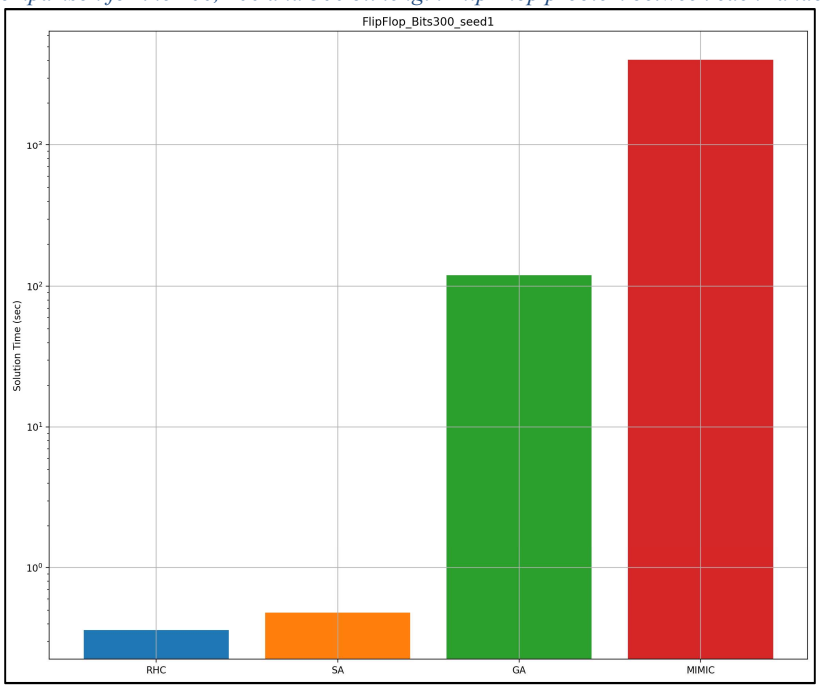

Figure 2-6 Wall clock run time comparison for the 300 bit length Flip Flop Problem between each random optimization algorithm on a log-scale in the y-axis averaged across 3 random seed runs

MIMIC overall performed best for this problem, finding the best fitness for each problem size except for the large problem where the genetic algorithm performed well. MIMIC achieved high fitness with many times fewer fitness function evaluations than the other algorithms and in fewer iterations. For this problem, the fitness function evaluation of a state runs in linear time with the size of the problem making it very fast. As MIMIC runs significantly longer per iteration than the other algorithms, this doesn't highlight MIMIC's ability to achieve high fitness scores with low number of function evaluations. However, if this problem required a higher order time complexity fitness function to be evaluated every iteration, the other algorithms particularly RHC and SA would run significantly longer while MIMIC's time performance wouldn't increase as much since more computation is occurring outside of the fitness function when running MIMIC.

3 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.

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 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, as well as the F1 score and the wall clock fit time of the

model. The F1 score incorporates the algorithms precision and recall giving more information about the performance of the algorithm. 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.

After varying hyperparameters for the neural network, the *relu* activation function and the mean squared error loss function were found to perform the best for all algorithms. For the genetic algorithm the mutation probability was set to 0.01. For the other algorithms a learning rate of 0.05 was used. All algorithms were run for 1000 iterations to evaluate their performance as shown in Figure 3-1. Iterations were run up to 5000 but there were no meaningful increases in performance in the way of fitness, accuracy, or F1 scores and only contributed increased fitness time.

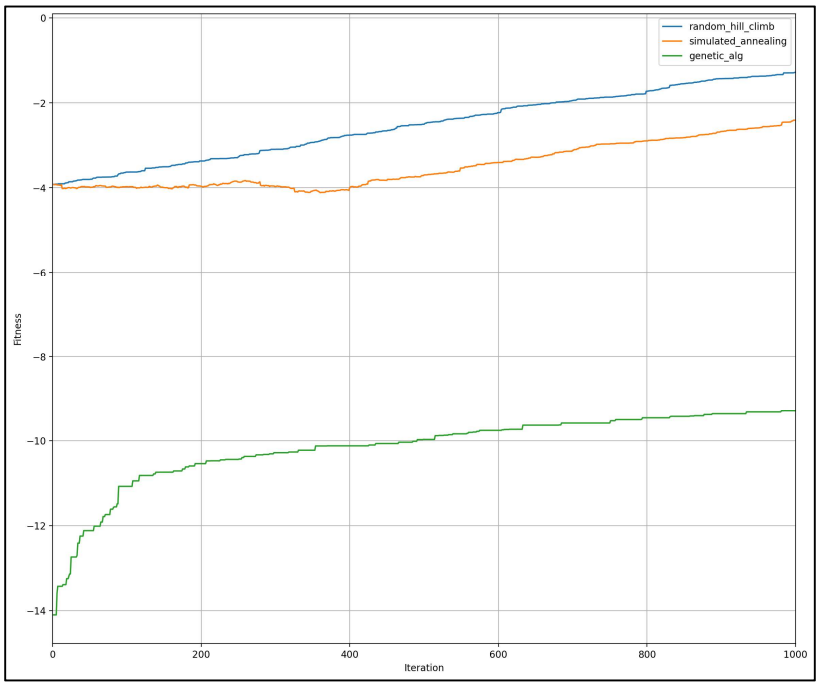

Figure 3-1 Performance comparison between each random optimization algorithm and backpropagation for neural network weight optimization

Similar to the first experiment for each optimization problem, the random hill climbing and simulated annealing optimization algorithms solved the fitness problem orders of magnitude faster than the genetic algorithm and achieved a better fitness. However, in terms of accuracy and F1 score on the test set, the genetic algorithm performed better than SA and RHC. After 1000 iterations, the performance of each algorithm is summarized in Table 3-1.

Table 3-1 Neural Network Weight Optimization Performance Comparison of Three Random Optimization Algorithms

Algorithm	Test Case Accuracy (Weighted)	Test Case F1 Score	Fitness Time
Simulated Annealing	0.619	0.646	280 seconds
Random Hill Climbing	0.713	0.731	290 seconds
Genetic Algorithm	0.765	0.773	11,000 seconds

While the fitness being shown in Figure 3-1 is calculated by the continuous optimization problem when forming the neural network problem, this still doesn't represent the performance of the algorithm on the test set which is why these additional performance metrics were included. Additionally, the fitness and optimization problem formed to fit the neural network weights is specific to the algorithm which explains why the fitness values in Figure 3-1 don't correlate to the performance on the test set data in Table 3-1. It is important to note that these randomized optimization algorithms are well suited for discrete state optimization while continuous state estimation is a more challenging task such as neural network weight

optimization. Transforming a continuous state optimization problem can be done in several ways. One of which is using a bit string representation of a floating point value for a single state parameter. However, this increases the number of states for a problem by a large factor which contributes to longer time to convergence, instability, and even poor overall performance. This is one explanation for the poor performance of these algorithms relative to the original MLPClassifier used previously for this data set. Additionally, using other loss and activation functions for the network with each algorithm negligibly affected each algorithms' performance.

Each of the algorithms performed worse than the original Multi-layer perceptron classifier model used for this data set predictions which achieved a near perfect accuracy on both the training and test data sets upon tuning. The genetic algorithm, simulated annealing, and random hill climbing achieved lowed accuracy and F1 scores than the MLPClassifier. Additionally, despite significant hyperparameter tuning, the algorithms all trained in significantly longer time than the MLPClassifier.

4 SUMMARY

In summary, this report and the two experiments outlined previously, explore randomized optimization algorithms and their application in supervised machine learning, spotlighting random hill climbing (RHC), simulated annealing (SA), genetic algorithms (GA), and MIMIC. Two experiments were conducted: one involving discrete combinatorial problems (Travelling Salesman, Random Bit Matching, and Flip Flop) and another optimizing weights in a neural network. Key metrics for evaluating algorithm performance included accuracy, F1 score, and computation time. Results showed that genetic algorithms performed well for larger, complex instances in the Travelling Salesman Problem. In the Random Bit Matching Problem, simulated annealing exhibited robust convergence and performance especially in terms of computation time when compared to the genetic algorithm. MIMIC stood out as the most effective algorithm for the Flip Flop Problem, consistently finding the best solution compared to the other algorithms which would get stuck on local non-global optima.

For the second experiment involving optimizing neural network weights for mushroom classification, each random optimization algorithm except MIMIC were fit to a training data set and tested against a test data set. Their performance were poor relative to the original MLPClassifier used for this problem and their training times were significantly longer. Of the random optimization algorithms, the genetic algorithm performed the best in terms of fitness and F1 score; however, as previously shown, it required a significantly longer time to train than the simulated annealing and random hill climbing algorithms. Simulated annealing and random hill climbing performed relatively similarly and trained in the time on the order of the original MLPClassifier but still performed significantly worse.

5 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.