Randomized Optimization

Josh Adams
jadams334@gatech.edu

Abstract—Analysis of random optimization using four variations of random search algorithms. An example of this would be random restart hill climbing, where the algorithm continually tries to climb the hills in the data and once it gets to a stopping point, it restarts to a random location and starts again. Random search algorithms are useful when attempting to find the global optima, as they each implement various methods to escape a local optimum.

1 INTRODUCTION

Using four different random search algorithms randomized hill climbing (RHC), simulated annealing (SA), a genetic algorithm (GA), and MIMIC to solve three different discrete optimization problems. The problems will be K-Colors, Flipflop and Knapsack. Each of these problems will be represented by bit strings of varied length. The longer the string, the larger of a state it represents which is indicative of a more difficult problem.

2 GENETIC ALGORITHM SHOWCASE

The K colors problem is tasked with setting colors for various nodes within a graph, with the stipulation that no two adjacent nodes are the same color. This is a unique problem which proves to be difficult for some algorithms, but others perform reasonably well. Genetic algorithms perform extraordinarily well when comparing fitness as well as run time. One reason behind its performance is the algorithms methodology which takes the most fit examples — like a form of selective breeding — and produces a new set subset of examples. The expectation is that the offspring should perform at least as well as their parents.

Figure 1 shows a comparison between the fitness achieved by the four stochastic algorithms. Each algorithm was tested ten times with ten different seeds. The seeds were generated on demand using the system clock, the averages of all iterations are shown. The lighter areas of the same color indicate the standard deviations of each algorithm of its fitness and runtimes. The genetic algorithm used was able to reach one of the local maxima in less than 100 iterations, while randomized hill climb was able to reach the same point in almost 250 iterations. Mimic and simulated annealing performed poorly on this problem in terms of fitness. I would expect increasing the number of available iterations to simulated annealing would allow it to reach one of the maxima'.

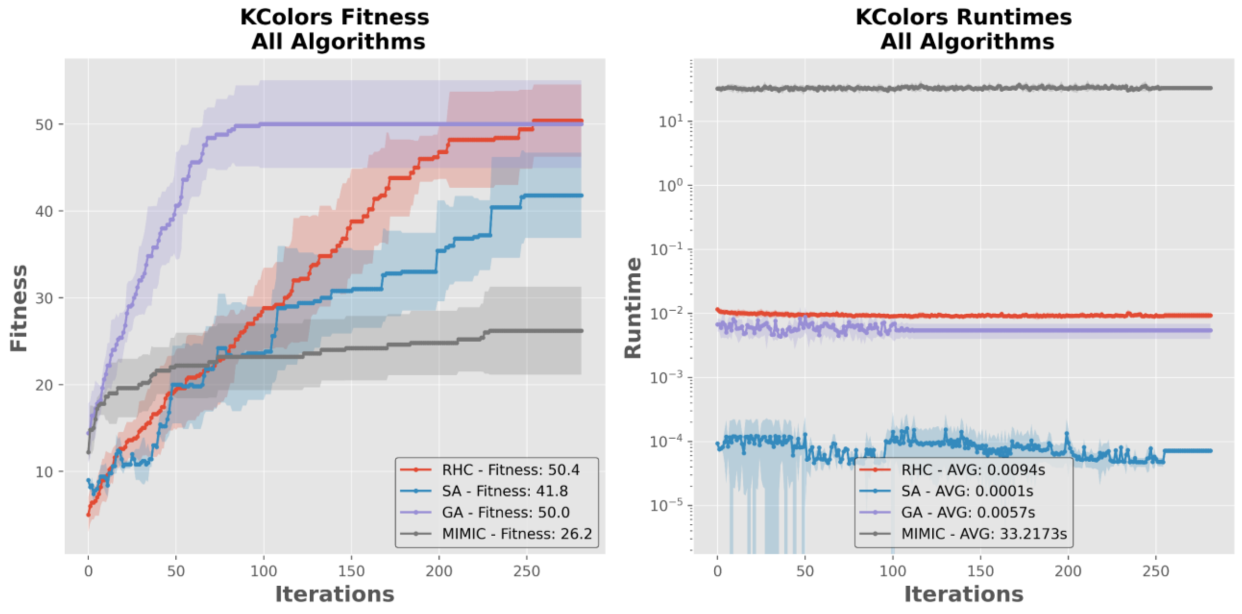


Figure 1—Fitness and runtimes for RHC, SA, GA, and MIMIC on the K-Colors problem—final fitness next to the algorithms name in the legend and average runtime.

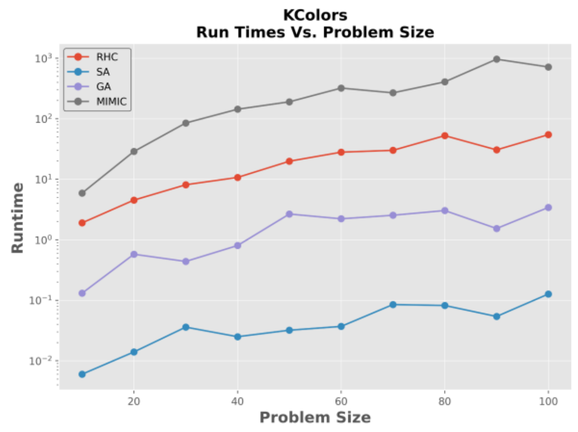
When looking at the runtimes of the algorithms, there is a clear loser and that is MIMIC which was over 3,000 times slower than randomized hill climbing and over 330,000 times slower than simulated annealing on average. While it may seem that mimic is a lackluster algorithm, there are situations in which mimic shines. Using gridsearch I was able to find each algorithm an appropriate set of hyper-parameters. Mimic would have benefit from more searching of parameters, but I was restricted by mimic's incredibly long run times. This limited the utilization of gridsearch as it would take in some cases days for the grid searches to finish. Establishing a max number of iterations was beneficial for comparison. It highlights nuances of the different algorithms such as being able to get to a local maximum quicker than others. Looking at the runtimes graph shows some interesting points in the lower iterations for simulated annealing. It has a large standard deviation. To me this would suggest it was having trouble exiting some of the local extrema or was in a large basin and was having difficulty escaping. Increasing the initial temperature because as the temperature is higher this makes it easier for the algorithm to move over or away from local optima.

Random hill climbing was the easiest algorithms to tune, just increasing restarts and iterations quickly produced results. The most difficult to work with was simulated annealing. There are many hyper-parameters to tune and minor changes dramatically change the algorithms behavior depending on the decay schedule used. Using the geometric schedule, the algorithm was able to stay in an area longer exploring to find more optimal routes. While the exponential was quick to leave local optima, but as the temperate reduces its ability to leave those local optima quickly diminishes. I found the exponential schedule to be difficult to work with.

Table 1 — K-Color results for the various algorithms along with the optimal hyper-parameters found using gridsearch.

Algorithm	Iterations	Attempts	Unique Hyper-parameters	Avg Iteration Time (sec)	Fitness
RHC	*256*	*128*	*Restarts — 256*	*0.0094s*	50.4
SA	*256*	*32*	*Geom_Decay (init_temp=4, decay=0.97, min_temp=0.001)*	*0.0001s*	41.8
GA	*256*	*128*	*Mutation_Probability — 0.75*	*0.0057s*	50.0
MIMIC	*256*	*128*	*Population_Size — 60, Keep_Percent–0.15*	*33.2173s*	26.2

2.1.1 *Function Evaluations vs Problem size*



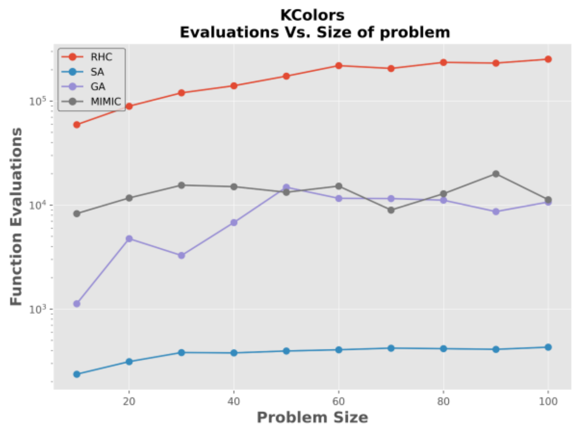


Figure 2 — Comparison of function evaluations vs problem size for KColors.

Function iterations are different than function evaluation. This paper assumes an iteration happens when the algorithm takes a step. An evaluation is just a call to the evaluate function which is not indicative of an iteration. For example, random hill climbing will make hundreds of evaluations during a single iteration. This why we see random hill climbs function evaluations increase so quickly. This chart is a little misleading because it would suggest that the random hill climb algorithm is horrible in comparison because it makes many thousand more evaluations than all the other algorithms. Even though the random hill climb makes 200,000 evaluation, each evaluation is very quick. Unlike with mimics very few evaluations which take a long time to finish. I would have expected mimic to have performed better than it had, the only rational explanation was that I did not provide it with proper hyper parameters.

3 SIMULATED ANNEALING SHOWCASE

Figure 3 shows the performance of the algorithms on the flipflop problem. Instead of focusing on the fitness of the algorithm, I am going to be focusing on the runtimes. Each of the algorithms were able to reach around the same maximum fitness. The big difference between the algorithms here is the speed. Simulated annealing was almost one hundred times faster than the genetic algorithm on each iteration. A very interesting structure to the simulated annealing's fitness line. It highlights one if the best features of simulated annealing, where it will take a loss in the short-term with the hopes it finds an even better solution later. Around the 90th iteration we see simulated annealing was having trouble in its environment its momentum lowers. And it begins to take new positions which are worse than previous. As the 'temperature' returns the algorithm begins making more progress. How this works is when the temperature is high, the algorithm is more capable of leaving sub-optimal areas. Effectively having a very large learning rate. As it progresses the temperature lowers which allows the algorithm to explore for of the data in hopes of find the global optima.

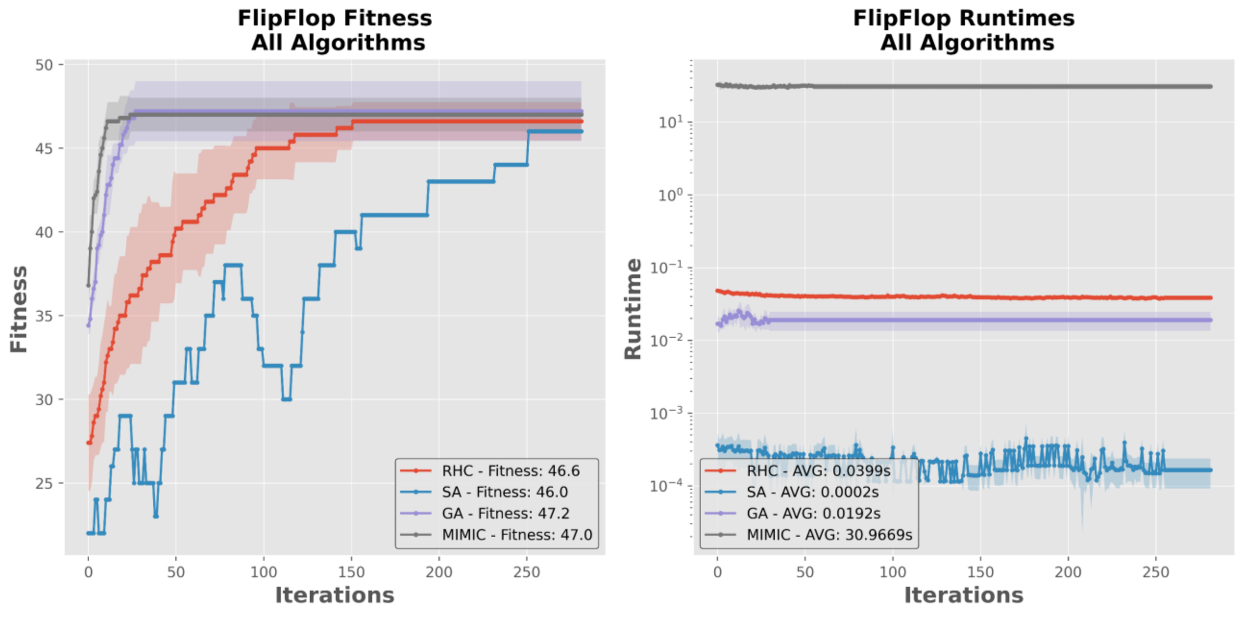


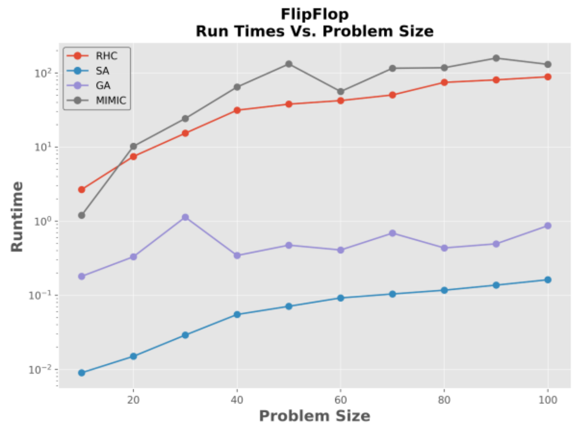
Figure 3—Make sure your flowcharts are more useful than this one. Source: XKCD.

Table 2 — Parameters for the algorithms on the flip flop problem.

Algorithm	**Iterations**	**Attempts**	**Unique Hyper-parameters**	**Avg Run Time (seconds)**	**Fitness**
RHC	*256*	*160*	*Restarts - 160*	*0.0399s*	46.6
SA	*256*	*32*	*GeomDecay (4, 0.97, 0.001)*	*0.0002s*	46.0
GA	*256*	*16*	*Mutation - 0.8*	*0.0192s*	47.2
MIMIC	*256*	*128*	*Amount Kept – 0.12*	*30.9669s*	47.0

3.1.1 *Function Evaluations vs Problem size*

A similar situation as the last example where random hill climbing produced the most evaluations. While unlike last time both mimic and random hill climbing had similar runtimes as the size of the problem increased. Looking at simulated annealing it produced the fewest evaluations as the size of the problem increases, while also being hundreds of times faster than the other algorithms evaluations.



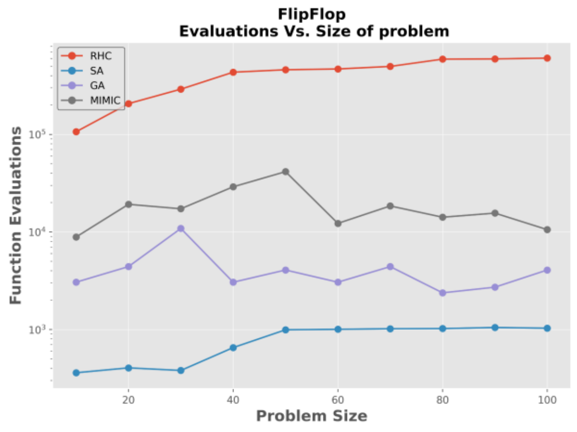


Figure 4 — Comparison of function evaluations vs problem size for Flipflop.

4 MIMIC SHOWCASE

The knapsack optimization problem is based around the value of a container. The containers value corresponds to the items inside of it. Each item has a weight, and the container is only able to hold up to a certain weight. The goal is to maximize the containers value, this means optimizing what is held in the container. Mimic was able to reach one of the maxima in the fewest iterations with the genetic algorithm right behind it. While mimic was the top performer in fitness it once again was the slowest by multiple orders of magnitude. Each of the algorithms were able to reach one of the maxima in the problem. Grid search was used to find optimal hyper-parameters for each of the algorithms. The resulting hyper-parameters are provided in *"Table 3"*. Using a correlation matrix on many of the subsets of hyper-parameters help point me in the right direction. I plotted the training accuracy with different parameter values in the x and y axis. This would show how the changes in these hyper-parameters would impact accuracy.

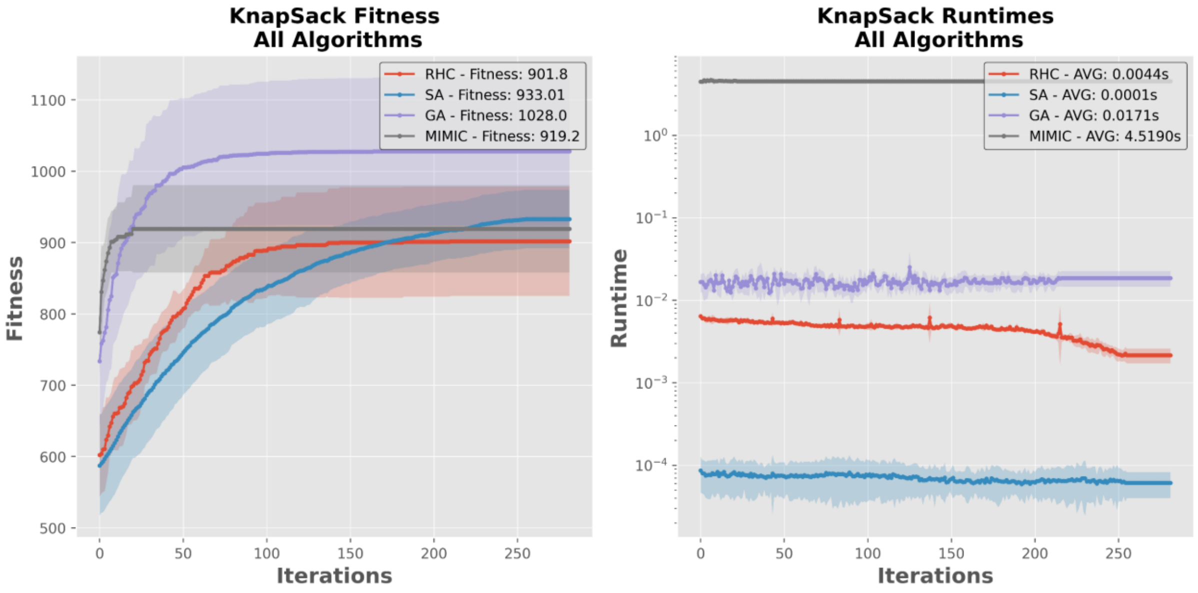


Figure 5 — Fitness and runtimes for RHC, SA, GA, and MIMIC on the FlipFlop problem. Their final fitness and average runtimes are within the legend.

Table 3 — Knapsack results for the various algorithms along with the hyper-parameters used.

Algorithm	**Iterations**	**Attempts**	**Unique Hyper-parameters**	**Avg Run Time (sec)**	**Fitness**
RHC	*256*	*256*	*Restarts — 256*	*0.0044s*	901.8
SA	*256*	*32*	*Geom_Decay (init_temp=2, decay=0.99, min_temp=0.001)*	*0.0001s*	933.01
GA	*256*	*4*	*Population_Size — 338, Mutation_Probability — 0.6*	*0.0171s*	1028.0
MIMIC	*256*	*32*	*Population_Size — 239, Keep_Percent–0.12*	*4.5190s*	919.2

5 NEURAL NETWORK OPTIMIZATION

The dataset that will be used through the remaining experiments will be the Fashion-MNIST dataset. The reason I am using this dataset is that it is more difficult to classify correctly than the MNIST dataset and gradient descent performs well but not as well as I had expected. Using three different randomized optimization algorithms; RHC (randomized hill climbing), SA (simulated annealing), and GA (genetic algorithm), I will optimize the weights for the neural network and attempt to outperform gradient descent.

5.1.1 *Assignment 1 Neural Network*

In the first assignment through various optimization technique the following hyper-parameters were the most optimal. Having a single hidden layer with 40 nodes within that layer. Setting the regularization parameter *'alpha'* set to 1.0, and the maximum number of iterations to be 400. The activation function was *'relu'* and the solver for weight optimization was based on gradient descent.

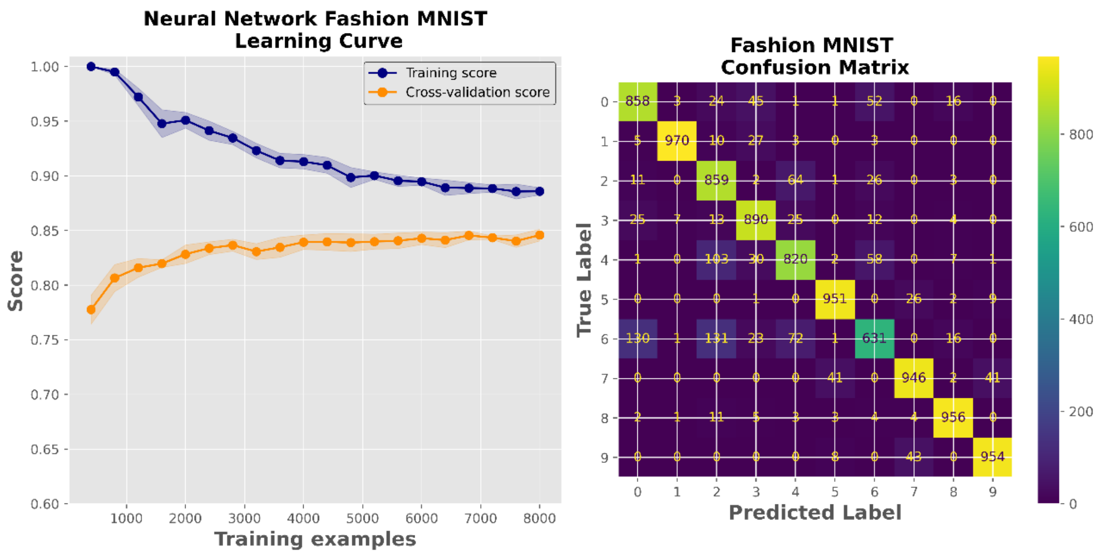


Figure 6— Learning curve, and confusion matrix of neural network from assignment_1.

The left-hand side of the graph —which corresponds to lower number of training examples— shows high variance because we have very high accuracy on the training set and low accuracy on the cross-validation set. At those points my network is overfitting the training set and not generalizing well to unseen data. Increasing the number of training examples helped reduce the overall impact of the over-fitting. Adding more data would reduce the gap between the training accuracy and cross-validation accuracy. Being computationally constrained limited the testing to 10000 examples. The dataset I am using has 60000 training examples, which could be used if a more powerful computer were available. Ideally the two curves will converge to a point at which they will get no closer, this is due to irreducible error within the dataset.

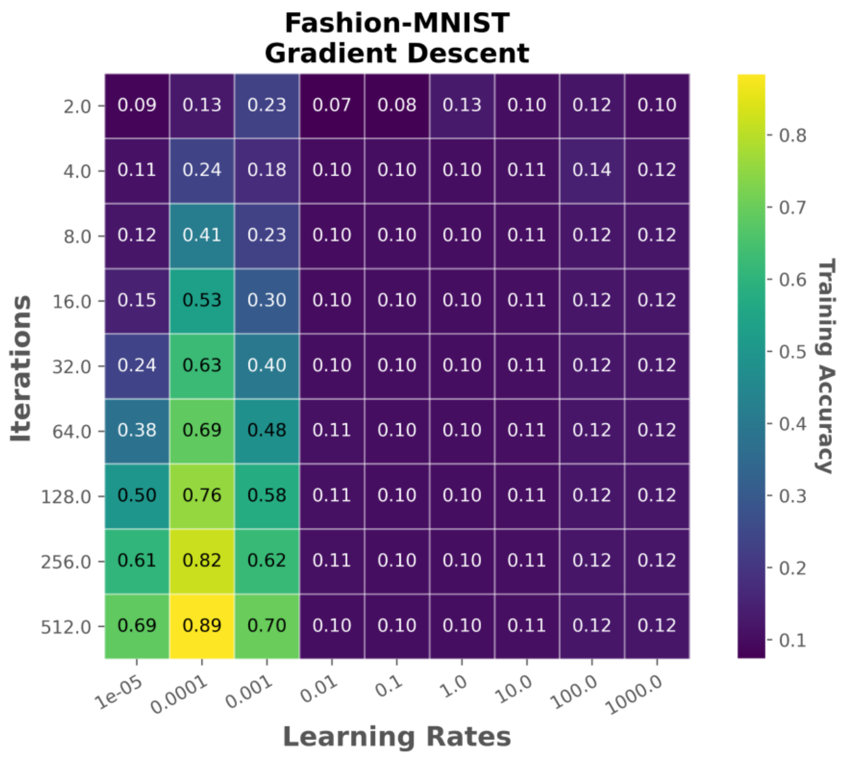


Figure 7 — Correlation matrix for learning rates and iterations.

Figure 11 is a correlation matrix which illustrates the how learning rates in conjunction with the number of iterations, impacts the accuracy. *Figure* 11 would suggest that training accuracy would increase when using lower learning rates and higher number of iterations. The correlation matrix provided guidance to a much smaller, albeit a better subset of hyper-parameters to search through with gridsearch.

Table 4 — Neural Network optimization with different algorithms on Fashion-MNIST

Algorithm	**Nodes**	**Iterations**	**Attempts**	**Learning Rate**	**Avg Run Time (seconds)**	**Loss**
Gradient Descent	*40*	*4096*	*1*	*0.001*	*0.0094s*	~0.0169
RHC	*40*	*10000*	*100*	*1.0*	*0.0001s*	1.043
SA	*40*	*300*	*100*	*30.0*	*0.0057s*	11.543
GA	*40*	*1000*	*5*	*7.0*	*33.2173s*	25.835

5.1.2 *Gradient Descent Vs Randomized Optimization*

Randomized hill climbing, simulated annealing and genetic algorithms can be used to find the weights of a neural network, but gradient descent is a much better option. Each of the three random optimization algorithms use various methods for searching a problem space. Random hill climb will search over an area until it is unable to progress, it then will restart by selecting another random location in the problem space and start again iteratively working its way through the problem. When the problem is discrete this works well because given enough iterations and time, the algorithm will be able to visit every state. Gradient descent was able to minimize the loss almost instantly which is completely different than how randomized optimization algorithms perform. Randomized optimization uses randomness to sample its environment and make more informed decisions about where to go next. Gradient descent works by calculating the gradient with respect to each weight within the network. What this means is that gradient descent does not have to sample some arbitrary number of states to make progress. It is able to update the weights in a manor to provide progress, in a single pass of backward and then forward propagation.

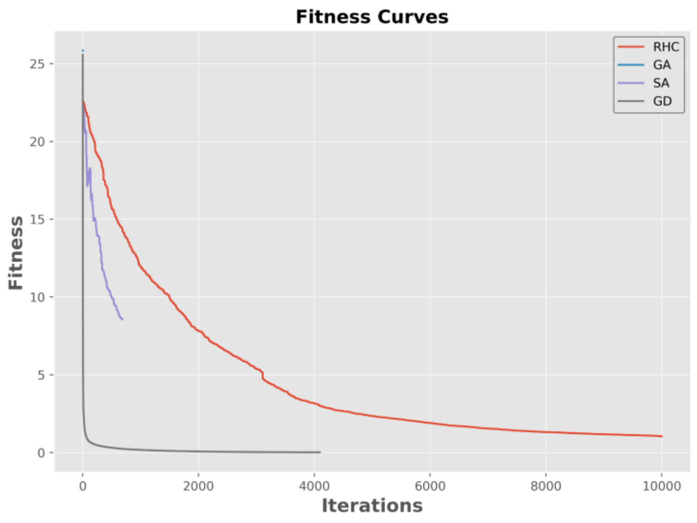


Figure 8— Fitness curves of the different algorithms compared to iterations

6 CONCLUSION

Genetic algorithms were far superior in almost every instance when it came to discrete problem spaces. They were able to reach one of the highest maxima in the problem state as well as be the first one to a maximum. Randomized optimization may not be a solution for all problems, but I see where it could be applicable such as hyper parameter tuning. A genetic algorithm would be a great replacement over the forms of gridsearch we currently are using. Randomized optimization algorithms have a purpose, but that purpose is not to find the weights of a neural network. Comparing the various randomized optimization algorithms to gradient descent is an unfair battle. Gradient descent should win in practically any scenario for the reasons discussed in the above sections comparing gradient descent to randomized optimization.

7 REFERENCES

1. Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. https://github.com/gkhayes/mlrose. Accessed: 01 10 2020.
2. Rollings, A. (2020). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python, hiive extended remix. https://github.com/hiive/mlrose. Accessed: 01 10 2020.
3. https://mlrose.readthedocs.io/en/stable/
4. https://github.com/zalandoresearch/fashion-mnist
5. https://en.wikipedia.org/wiki/Stochastic_hill_climbing
6. https://en.wikipedia.org/wiki/Hill_climbing
7. https://en.wikipedia.org/wiki/Simulated_annealing
8. https://en.wikipedia.org/wiki/Genetic_algorithm
9. https://developers.google.com/optimization/bin/knapsack
10. https://en.wikipedia.org/wiki/Graph_coloring
11. https://scikit-learn.org/stable/
12. https://numpy.org/
13. https://pandas.pydata.org/
14. https://matplotlib.org/