Assignment 2: Randomized Optimization

Jainil Modi
JMODI30@gatech.edu

I. Abstract

In this assignment, we explore the pros/cons of four randomized optimization algorithms, as well as the performance of the first three on Neural Networks trained previously in assignment 1 using backpropagation as the optimization technique. The algorithms are as follows: Randomized Hill Climbing (also referred to in this assignment as RHC), Simulated Annealing (also referred to as SA), Genetic Algorithm (also referred to as GA), and MIMIC. There also are 3 problems that these four algorithms are being applied to as well. The descriptions of the problems can be found below:

A. *Problem 1 - Max K Color*

This is a problem that originates in graph theory. You are provided with an undirected graph, and a positive integer k. The objective is to determine the possibility of assigning colors to vertices in such a way that no two adjacent vertices (two vertices that share an edge) share the same color. We would like to find the largest k that satisfies each said graph.

B. *Problem 2 - Queens*

Given an NxN chess board, the goal is to find out how to place queens on the board so that they are not able to attack each other. This can be done either horizontally, vertically, or diagonally, as queens can move in any way, as the Queen can move in any unspecified number of spaces. To make this idea a little clearer, a pawn, at the start, can move 3 spaces, unless impeded. Then after the first move, the pawn can only move 1 space at a time.

C. *Problem 3 - KnapSack*

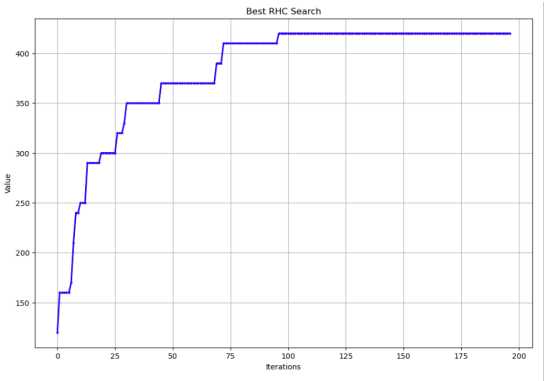
A foundational problem in combinatorial optimization, the aim of this problem is maximize the value or utility of limited resources. Given a set of items, where each item has a value and a weight assigned to it, determine the highest value that can be achieved by choosing a subset of these items and keeping the total weight within a specified limit, which happens to be the capacity of the knapsack.

II. Problem 1 - Max K Color

For the setup of this problem, a list of edges and nodes is required. I have the following combinations for each of the four algorithms to optimize: [(0, 1), (0, 2), (0, 4), (1, 3), (2, 0), (2, 3), (3, 4)]. I also have a custom fitness function defined to be c times the state (either off - 0, or on - 1), which is a list: [0, 1, 0, 1, 1]. Because I initialized 5 states and allowed 100 restarts for the first 3 algorithms, the highest score achievable by this fitness function is 500. For MIMIC, I set restarts equal to 200, so this number doubles to 1000.

A. *Algorithm 1 - Randomized Hill Climbing*

In general, randomized hill climbing runs the fastest of all four of these algorithms, for any of the problems. In this particular problem, randomized hill climbing ran the second fastest of the three renditions: the CPU time and Wall Time were 1 min 25s and 1 min 26s, respectively.



The graph above demonstrates the powerful ability of randomized hill climbing to get out of local optima: there are portions of the graph where the algorithm seems to plateau or ”get stuck”, but then it bounces back quickly and continues to improve the fitness score. The mean score and max score by randomized hill climbing were 419.30693 and 420.0, respectively. These are fairly good scores.

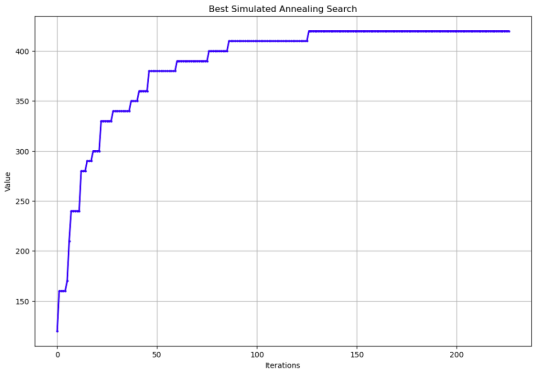
B. *Algorithm 2 - Simulated Annealing*

In this section, I had the following temperatures: [1, 5, 10, 100, 250, 500, 750, 1000, 5000, 10000], as well as using both Exponential and Geometric Decay. This gives 20 linear combinations of optimizations. Of all three problems, simulated annealing performed extremely quickly here: 3.77s and 5.14s for CPU and Wall Time, respectively.

I suspect the reason to be that the initial graphs that I provided were not complex by any means, so going through all the linear combinations and trying to find the best version was not too difficult. This is also evidenced by the fact that the max score and mean score achieved by simulated annealing for MaxKColor was 420 (the same).

From this graph, it is demonstrated that SA does not really seem to get stuck in local optima, and once it finds the true maxima, it stays there. For the few times it does get stuck, it is able to move out of the local optima quickly.

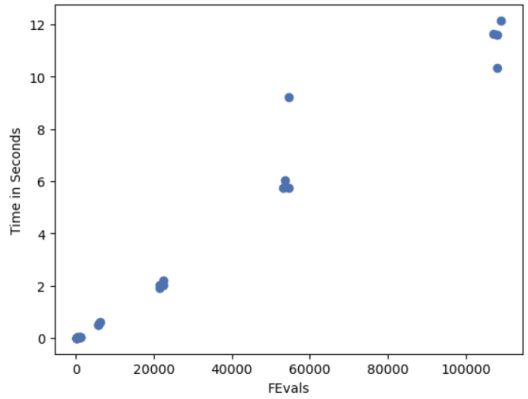
When done with the gridsearch for simulated annealing, the output for the best linear combination is ExpDecay(init_temp=1, exp_const=0.005, min_temp=0.001)



C. Algorithm 3 - Genetic Algorithm

Like other algorithms for this problem, GA ran very quickly: 1min 6s and 1min 22s for CPU and Wall Time, respectively. In this section, I used four mutation rates and four population sizes: [0.1, 0.25, 0.5, 0.75] and [50, 200, 500, 1000]. My initial hypothesis was that the best combination would be a 0.25 mutation rate and population size of 1000, because this would give the most population size with a mutation rate that would not cause the function to overshoot.

My intuition was proven wrong: neither the mutation rate nor the population size contributed to a higher score. The scores for each of the four mutation rates and population sizes remained steadfast at 420. The one thing that seemed to grow linearly was the computation time:



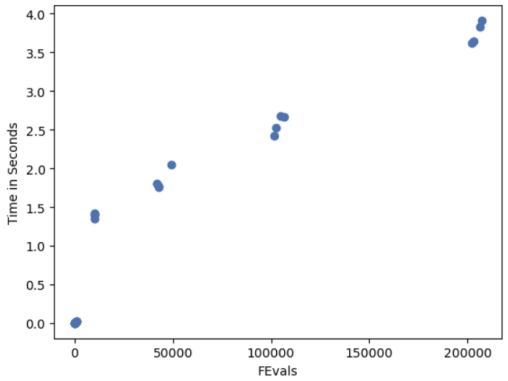
The graph above illustrates the sparse linear relationship between FEvals and Time.

D. Algorithm 4 - MIMIC

For this section, I used four keep percentages and four population sizes: [0.10, 0.25, 0.5, 0.75], [50, 200, 500, 1000], giving a total of 16 linear combinations try for MIMIC. As expected, this algorithm was the most computationally expense, but not by much when comparing it to Queens or KnapSack: 37.3s and 38.4s for CPU and Wall time. Similar to SA, I hypothesized that a higher keep percentage and higher population size would perform better.

My hypothesis held true for the population variable but not the keep percentage: for each of the population sizes, the scores were: 360.0, 415.0, 420.0, and 420.0. For the keep percentage, the score improved for the first three iterations, then started to decrease: 395.0, 407.5, 410.0, and 402.5. From this, I can interpret that MIMIC benefits from larger populations and is able to estimate the probability distribution over the solution more accurately. However, I think the keep percentage attribute worked a little differently here: it found a maxima around 50 percent of the population for each population size, and things start doing worse afterwards.

The one interesting part of this rendition of MIMIC was that population size did not really impact the speed: for each population size, an increase in population did not linearly correlate to a linear increase in computation time:



III. Problem 2 - Queens

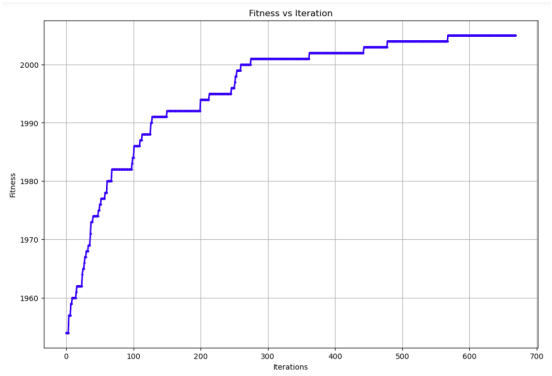
For the setup of this problem, I went with a 64 queen setup (8x8), which means that the number of attacks that need to avoided are 2080. This is then the maximum number for fitness function. Additionally, I initialized the max_attempts for each of the first three algorithms to be 100, just for simplicity's sake. But, for MIMIC, I set it to 200, as doing any more would take ages to complete.

A. Algorithm 1 - Randomized Hill Climbing

randomized hill climbing is an algorithm takes incremental steps to improve the fitness function. Because its predecessor Hill Climbing can get stuck in local optima, it is better to use randomized hill climbing to avoid not finding the most optimal fitness. Of the four algorithms for this section, randomized hill climbing took the second least amount of time. For this instance of RHC, the algorithm took 582 seconds to run. If we average that by the 100 iterations that we ran, we get 5.82 seconds per run. Additionally, each iteration had about 560 steps, bringing the total iterations 56000.

In this list of 100 RHCs, the average value we ascertain is 1999.267. When our max for this function is 2080, 1999 is not a bad score at all. The highest score that randomized hill climbing achieved in these was 2005, a difference of only 6! I think that RHC performed well here because complexity of this problem: there are many dimensions. Because the average

it got was 1999, there was not much room for improvement as is, and because there are so many queens, it was likely hitting some sort of local optima, one of the pitfalls that it can face. Below is a graph of the Fitness (y-axis) versus Iteration (x-axis).



From the graph above, it is noticeable that randomized hill climbing hits a plateau several times, and when it does, it tends to stay there for some time before improving. This is likely due to the fact that it is getting stuck in a local maxima and not being to move very far afterwards.

B. Algorithm 2 - Simulated Annealing

Here, I used both functions for Temperature decay - Exponential and Geometric. I also used ten different initial temperatures: [1, 5, 10, 100, 250, 500, 750, 1000, 5000, 10000].Multiplying both of these (decays and temperatures), there was a total of 20 different algorithms ran for this particular problem.

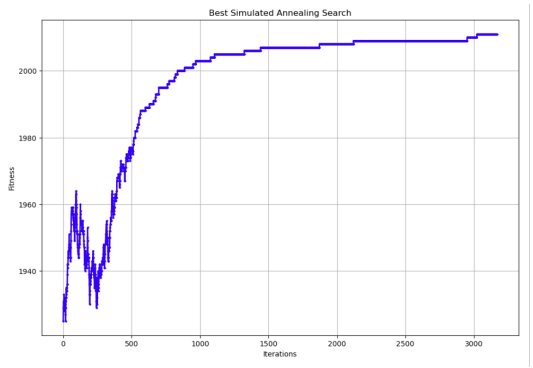
The ”gridsearch” done here returned the following linear combination as the best fitness: GeomDecay(init_temp=100, decay=0.99, min_temp=0.001).

In comparison to RHC, simulated annealing has a much higher performance speed in terms of Wall Time and CPU time. The algorithm was able to compute the 20 linear combinations in 146 seconds, which is close to about 7 seconds per compute. As a whole though, SA is definitely a better approximator here: the average value it computed was 2009 out of 2080, which means the average was even better than the best that was computed for randomized hill climbing (2005). The best value is could achieve was 2011, which was higher than RHC, but the difference between the average and the best was much less. This seems to suggest that overall, simulated annealing does better. The graph of fitness versus iterations for simulated annealing can be seen below.

When comparing the graphs for RHC and SA, we can see that simulated annealing rises much quicker, but then plateaus just as fast. This is different from randomized hill climbing, which grew a little more steadily, and plateaued later on.

C. Algorithm 3 - Genetic Algorithm

In this algorithm’s implementation for Queens, I used a linear combination of four population sizes: [50, 200, 500, 1000], and four mutation rates: [0.1, 0.25, 0.5, 0.75]. This algorithm was the second most computationally expensive for Queens: 57 mins 55s, and 63 mins 16s respectively for CPU and Wall Time.



The most interesting takeaway from this algorithm was the fact that the mutation rate and population size had little-to-no effect on the fitness score at all: for each of the mutation rates (0.10, 0.25, 0.50, and 0.75), the fitness scores were 2000.00, 2001.25, 2001.50, and 2000.00 respectively. For each of the population sizes (50, 200, 500, 1000) the fitness scores were 2001.75, 2009.25, 2010.75, and 1981.00.

Below, you can see the relationship between FEvals and time:

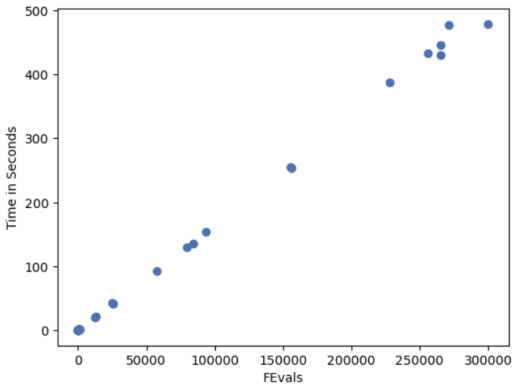


Fig. 1. There seems to be a positive linear relationship between FEvals and Time

D. Algorithm 4 - MIMIC

Of all the algorithms for this problem, MIMIC was by far the most computationally intensive. I thought I would begin with the CPU time and Wall time: 2 hours and 55 minutes, and 2 hours and 59 minutes, respectively. These were both despite using the setting ”use_fast_mimic=True”, which uses mutual information in a vectorized form. MIMIC also performed the worst of the four algorithms: the average score that it got was 1973 out 2080, and the max score was 1988, much lower than RHC or SA, for example. This can be attributed to MIMIC’s preference for simpler problems. This problem, which I chose to do on an 8x8 board, was a lot of computing for MIMIC.

MIMIC likely performed worse than the other algorithms here because of the fact that it got stuck in a local optima, whereas simulated annealing, due to its probabilistic nature, has a potentially better chance of escaping local optima.

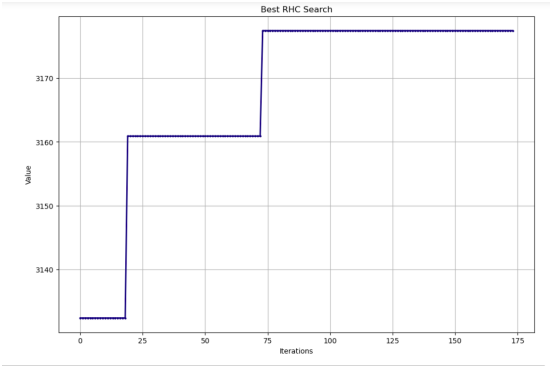
In terms of results, MIMIC behaved similarly to GA: the keep percentage and population size did not really effect the fitness score all that much: for both 0.25 and 0.75 keep percentages, there was no change in the score at all: 1973.666667. The score increased at 0.50 to 1974.333333, which suggests to me that 0.50 was a maxima (either global or local) for this problem, and the curve is likely uniform on both sides of this point. In terms of population size, there seems to be a weak positive correlation between it and fitness: between the population sizes of 50, 200, and 500, the fitness score went up 1961.000000, 1975.666667, 1985.000000, suggesting a diminishing margin of return.

IV. Problem 3 - Knapsack

In this problem, I set the Knapsack size to be 300 items, just arbitrarily choosing a weight. Next, I set the maximum weight percentage to be 50 percent, again arbitrarily choosing a value. This value means that up half the items will be chosen. Each item can weigh between 5 and 40, but no more and no less. For each item, there is also a random chance between 10 and 30 that they will be chosen.

A. Algorithm 1 - Randomized Hill Climbing

For this optimization technique, I set the random restarts and max attempts to 100. The CPU and Wall Time for this section were incredibly quick: 21.6s and 24s, respectively. I think this is because of how the algorithm inherently works: when weights exceed limits, the fitness function gets trapped at zero and this makes it very easy for it to get stuck at local optima. This is where the random restarts come in handy. The fact that randomized hill climbing gets stuck at local optima is evidenced by the fact that the average score for fitness was 1357.80228 and its maximum was 3177.419288, showing it could do much better. This average was much lower than its counterparts for this same problem, as we will see later. When trying to diagnose why, it may be helpful to see the graph of Score vs Iteration:

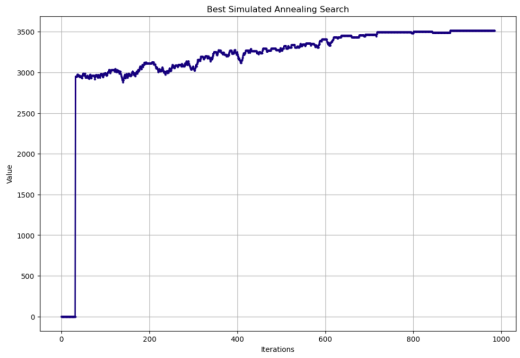


We can see that the algorithm almost looks like a step function here. It gets stuck in local maxima, and when it does, it tends to stick there for a long time before getting unstuck.

B. Algorithm 2 - Simulated Annealing

For this section, I used a similar set up to the other two problems: I performed the "gridsearch" with both geometric and exponential decay, and my temperatures were [1, 10, 50, 100, 250, 500, 1000, 2500, 5000]. This gives 20 linear combinations yet again. Of all the iterations of SA, this one was by far the fastest iteration: 2.36s and 2.9s for CPU and Wall Time, respectively.

Not only did simulated annealing perform the fastest, but it also generated better scores than RHC: 3236.7699286 for the average score and 3516.2727556 for the max score. I think this is because the temperatures do a better job of avoiding the pitfalls of being stuck in a local optima. We can see this illustrated in the graph below:



When comparing this graph to the graph for RHC, we can see easily that this does not have that step-function shape. This means that simulated annealing, for the purpose of this problem, is not getting stuck for long periods of at local maxima. It quickly improves, and then remains steady in its improvement of performance.

C. Algorithm 3 - Genetic Algorithm

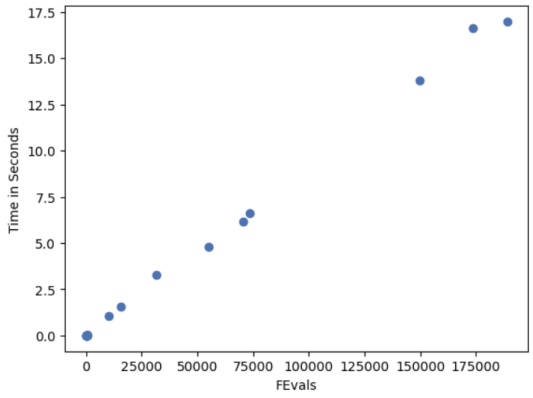
For this section, I created 16 linear combinations with mutation rate and population size:[0.1, 0.25, 0.5, 0.75] for mutation rates and [50, 200, 500, 1000] for population sizes. I hypothesized, similarly to earlier, that a larger population would do better.

In terms of speed, GA was relatively quick: 1min 1s and 1min 10s for CPU and Wall Time, respectively. The algorithm outperformed SA, which already did significantly better than RHC: 4204.956732201425 for the max score and 4108.211262383422 for the average score.

It was interesting to see again that the increase mutation rate did not really effect the score all that much: between a mutation rate of 10 percent and 50 percent, there was only a score difference of about 100 (4063 and 4163, respectively). In fact, a similar story was also depicted for population size: between a population size of 50 and population size of 300,

one that is 6 times larger, there was a difference of less than 200 (3925 and 4204 for 50 and 300, respectively).

However, the thing I thought was interesting was the relationship between FEvals and Time, which happened to be linear here as well:



The relationship seems obviously linear, but there are sparse points on the top right. Most points seem to be on the lower left side.

D. Algorithm 4 - MIMIC

To no surprise of mine, this algorithm took an extensive time. It was not nearly as long it took for Queens, but still a very long time: CPU times: 1h 21min 39s and Wall time: 1h 33min 56s. I think this is because it was computationally expensive for the 200 iterations to be performed, especially with 16 linear combinations of population sizes and keep percentages.

Despite the fact that MIMIC takes the longest to compute, it absolutely does not perform the best. For each of the three problems, simulated annealing does a better job. Although the max score of 4193.345625 is very good, the average score was 3676.744507. Even the average score for simulated annealing was than the max score for MIMIC.

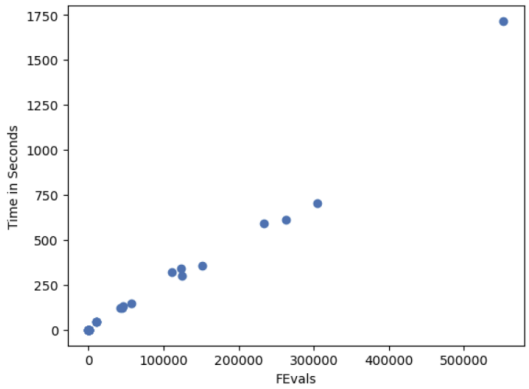
Next, there was no significant improvement in score with an increasing in keep percentage, nor for population size:

Keep Percent	
0.10	3737.632185
0.25	3870.762852
0.50	3846.385439
0.75	3252.197555

Lastly, we can see that the algorithm is not the strongest when we plot Time vs FEvals:

Most of the values are in the bottom left, but there is one value that is in the upper right.

Population Size	
50	3122.740404
200	3714.509433
500	3903.471546
1000	3966.256647



V. Neural Networks

From Assignment 1, I used the PIMA Indians Diabetes dataset. In it, we were trying to predict the presence (or lack thereof) of diabetes. I used the metric of f1-weighted score because the lack of balance in the outcome variable (diabetes). In this section, we will explore the performance of four neural networks: the baseline one uses backpropagation to improve itself, and the last three use the randomized optimization techniques we have been discussing.

A. Baseline Neural Network

For the initial neural network, we tried many parameters for the gridsearch portion in order to hyperparameterize it. We end up finding a neural network for our baseline that used the following parameters: MLPClassifier(activation='tanh', early_stopping=True, hidden_layer_sizes=(128, 64, 32), max_iter=1000). As such, to make sure the rest of the algorithms had uniform parameters, I made sure to keep the hidden layers and activation function the same for each. As for a baseline classification report, see below for both training and testing reports:

	precision	recall	f1-score	support
0	0.70	0.93	0.80	401
1	0.65	0.26	0.37	213
accuracy			0.69	614
macro avg	0.67	0.59	0.58	614
weighted avg	0.68	0.69	0.65	614

	precision	recall	f1-score	support
0	0.70	0.95	0.80	99
1	0.74	0.25	0.38	55
accuracy			0.70	154
macro avg	0.72	0.60	0.59	154
weighted avg	0.71	0.70	0.65	154

	precision	recall	f1-score	support
0	0.74	0.79	0.76	99
1	0.56	0.49	0.52	55
accuracy			0.68	154
macro avg	0.65	0.64	0.64	154
weighted avg	0.67	0.68	0.68	154

Here, it is easy to spot the fact that the neural network was likely overfitting: an f1-score in training of 0.77 for 1s and 0.52 for testing 1s. For 0s, it was 0.89 and 0.76, for training and testing, respectively.

Lastly, the baseline times were CPU: 6min 44s and Wall time: 1min 35s.

The loss function associated with this was log-loss, also known as cross-entropy loss, where the closer the non-negative value is to zero, the better.

B. Randomized Hill Climbing Neural Network

After performing the worst for all three problems above, I did not have any better expectations for this algorithm as it pertains to maximizing the neural networks. However, I was actually incorrect to believe this, as randomized hill climbing actually did the best in training for f1-score.

For the grid search, I set it up as follows: grid_search = 'max_iters': [100, 500, 1000], 'hidden_layers': [[10], [20], [30, 10], [50, 30, 10]], 'learning_rate': [0.1, 0.01, 0.001], "restarts": [0, 5, 20], "activation": [mlrose_hiive.neural.activation.tanh]

All in all the algorithm from mlrose actually tells you how many linear combinations that it is trying:

Fitting 5 folds for each of 324 candidates, totalling 1620 fits

I thought the CPU and Wall Time difference was interesting: CPU times: total: 15.9 s, Wall time: 25min 34s. This means that it was like only running on a single thread.

The result I got was as follows:

'max_iters': 500, "hidden_layers_sizes": [[128, 64, 32]], 'learning_rate': 0.1, "restarts": 0, 'activation': ¡function mlrose_hiive.neural.activation.tanh.tanh(x, deriv=False)¿

As expected, the activation function remained the same. However, I think it was interesting that it chose 0 restarts. I think this means that when it chooses optima, it does not move from that spot. This could also be corroborated by the graphs from the randomized optimization problems above: RHC gets stuck in plateaus and tends to stay there.

	precision	recall	f1-score	support
0	0.87	0.92	0.89	401
1	0.82	0.73	0.77	213
accuracy			0.85	614
macro avg	0.84	0.82	0.83	614
weighted avg	0.85	0.85	0.85	614

As expected for neural networks, the f1-score is lower in testing than training. However, when comparing the 1s score for both the baseline neural network and the one created using randomized hill climbing, it did a better job in training and testing.

C. Simulated Annealing Neural Network

I expected this algorithm to perform the best, but surprisingly, it did not. This is because I thought it would be able to get past the hump of local optima. In the grid search for this algorithm, I tried a few learning rates (0.1, 0.001, 0.025, and 0.0005).

The computation for this algorithm took a little longer than the random hill climbing algorithm: CPU times: 4min 33s and Wall time: 1min 17s. Although randomized hill climbing performed better in testing, I would argue that simulated annealing is the more reliable algorithm.

For the grid search, I set it up as follows: grid_search = "max_iters": [2500, 5000, 10000, 25000, 50000], "learning_rate_init": [0.001, 0.1, 0.5, 1], "hidden_layers_sizes": [[128, 64, 32]], "activation": [mlrose_hiive.neural.activation.tanh], "is_classifier": [True], "schedule": [mlrose_hiive.GeomDecay(1), mlrose_hiive.GeomDecay(10), mlrose_hiive.GeomDecay(100), mlrose_hiive.GeomDecay(5), mlrose_hiive.GeomDecay(50), mlrose_hiive.GeomDecay(500), mlrose_hiive.ExpDecay(1), mlrose_hiive.ExpDecay(10), mlrose_hiive.ExpDecay(100), mlrose_hiive.ExpDecay(5), mlrose_hiive.ExpDecay(50), mlrose_hiive.ExpDecay(500)]

As mentioned before, the reason I used tanh was because that is what the best estimator spit out for the MLP Classifier that used backpropagation, so I wanted each neural network to be level in that regard.

The algorithm also told us again how many linear combinations it was iterating through: "Fitting 5 folds for each of 192 candidates, totalling 960 fits".

This was by far the most intensive and time consuming algorithm I had run. Originally, I had it run with 360 candidates and 1800 fits. However, I thought something might have been broken, because I left it running overnight, for a minimum of 10 hours, and it still had not completed. This gridsearch, which has slightly more than half the original, took CPU times: 1min 37s, Wall time: 9h 55min 32s.

The estimator it returned was as follows:

'activation': ¡function mlrose_hiive.neural.activation.tanh.tanh(x, deriv=False)¿, 'hidden_layer_sizes': [128, 64, 32], 'hidden_layers': [10], 'learning_rate': 0.1, 'max_iters': 100, 'decay': 'mlrose_hiive.ExpDecay(10)'

Even though this gridsearch had to iterate through 960 fits, it ended up sticking with starting hidden layer size, hidden layers, and max iterations that it started with. From this, I can interpret that I had picked good weights for the parameters, because I had prior knowledge from the neural network I trained for assignment 1. I think the reason that the algorithm preferred an exponential decay is because it helps escape local maximas a little better than geometric decay.

	precision	recall	f1-score	support
0	0.74	0.83	0.78	401
1	0.59	0.45	0.51	213
accuracy			0.70	614
macro avg	0.66	0.64	0.65	614
weighted avg	0.69	0.70	0.69	614

	precision	recall	f1-score	support
0	0.71	0.86	0.78	99
1	0.60	0.38	0.47	55
accuracy			0.69	154
macro avg	0.66	0.62	0.62	154
weighted avg	0.67	0.69	0.67	154

The algorithm does not have drastic differences between training and testing, which illustrates the idea that simulated annealing, due to its probabilistic nature, is better at avoiding overfitting. The performance in testing being similar, although not better, also shows that simulated annealing tries to remove bias. An algorithm like randomized hill climbing that had such high variance is likely overfitting, which it certainly was. Simulated annealing also performed better than the baseline algorithm which used stochastic gradient descent as the optimizer.

D. Genetic Algorithm Neural Network

Of all four neural networks that were trained, the genetic algorithm took the longest time to optimize and train: CPU time and Wall Time were 11mins 33s and 4mins 3s, respectively. This means that in comparison to the other neural networks, the genetic algorithm was the most computationally expensive. When the difference between the two is this much, it means that there is threading that is happening, and the machine needs to split the process up so that it can be computed quicker.

When comparing the three algorithms, the genetic algorithm did not perform well when it came to the f1-score for 1s. It performed similarly to backpropagation: 0.31 f1-score in training, 0.34 in testing. This demonstrates that although the genetic annealing does not perform the best, it is an algorithm that does not really overfit. If anything, this means that there is high bias, which may be the result of underfitting.

The gridsearch for this algorithm had me worried for my PC's health. I had to run this gridsearch for over 24 hours, coming in at a whopping 25h 19m 37s.

The parameters and the tested attributes are as follows:

"max_iters": [100, 500, 1000, 2500], "learning_rate_init": [0.001, 0.1, 0.5], "hidden_layers_sizes": [[6, 6]], "activation": [mlrose_hiive.neural.activation.tanh], "is_classifier": [True], "mutation_prob": [0.1, 0.25, 0.5, 0.7], "pop_size": [100, 200, 500, 750, 1000]

After spinning for longer than a day, the algorithm returned: 'activation': ¡function mlrose_hiive.neural.activation.tanh.tanh(x, deriv=False)¿, 'hidden_layer_sizes': [128, 64, 32], 'hidden_layers': [10], 'learning_rate': 0.1, 'max_iters': 100, 'mutation_prob': 0.1, 'pop_size': 100

See the tables below for classification reports regarding both training and testing of the genetic algorithm:

	precision	recall	f1-score	support
0	0.74	0.83	0.78	401
1	0.59	0.45	0.51	213
accuracy			0.70	614
macro avg	0.66	0.64	0.65	614
weighted avg	0.69	0.70	0.69	614

	precision	recall	f1-score	support
0	0.71	0.86	0.78	99
1	0.60	0.38	0.47	55
accuracy			0.69	154
macro avg	0.66	0.62	0.62	154
weighted avg	0.67	0.69	0.67	154

The key attribute, besides f1-score to take a look at here is the recall, also referred to as sensitivity. In a problem like this, where we have sparse 1s (presence of diabetes), we care when the algorithm misclassifies a positive case (false negative). In general, the higher the recall score, the better. However, we can see that the recall score was lower in testing than training, and both of them were under 0.50, or 50 percent. A reasonably good score would be 0.6-0.7 (sixty to seventy percent), but we are far from that.

VI. Conclusions & Final Takeaways

Problem	CPU Time	Wall Time
MaxKColor	1 min 25s	1 min 26s
Queens	10 min 14s	12 min 12s
KnapSack	21.6 s	24.9 s
Neural Networks	2 mins 21s	1 min

TABLE I
Comparison of CPU and Wall Time for Randomized Hill Climbing

Problem	CPU Time	Wall Time
MaxKColor	3.77 s	5.14 s
Queens	2 mins	2 min 21s
KnapSack	2.36 s	2.9 s
Neural Networks	4 mins 33s	1 min 17s

TABLE II
Comparison of CPU and Wall Time for Simulated Annealing

Problem	CPU Time	Wall Time
MaxKColor	1 min 6s	1 min 22s
Queens	57 mins 14s	1 hr 3 mins 16s
KnapSack	1 min 1 s	1 min 10s
Neural Networks	11 mins 33s	4 mins 3s

TABLE III
COMPARISON OF CPU AND WALL TIME FOR GENETIC ALGORITHM

Problem	CPU Time	Wall Time
MaxKColor	37.3s	38.4s
Queens	2 hr 55m 48s	2 hr 59m 14s
KnapSack	1 hr 21m 39s	1 hr 33m 36s

TABLE IV
COMPARISON OF CPU AND WALL TIME FOR MIMIC

Taking a look at the tables above, we can compare in general across algorithms and problems and their average run times.

From this, we see that Simulated Annealing performs the fastest when compared to all other optimization problems. This is likely due to the fact that there are temperatures that control the probability of accepting solutions that are not as optimal. At higher temperatures, the algorithm is more likely to accept suboptimal solutions, and therefore, escape local optima and compute faster.

For the slower algorithms, it was no surprise that MIMIC took the longest time. This is because the problems I chose are computationally complex. For example, the larger you make a chess board for queens, the more spaces you have to check. MIMIC needs to check three things every single time: horizontally, vertically, and diagonally. This can be extremely painful when the boards get larger and larger.

Knapsack is also computationally complex: MIMIC has to keep track of the weight of the bag, the weight of each item, and the likelihood that they get chosen, as well as the amount of items that are already in the bag. Additionally, for larger populations, this can result in longer computation times.

MaxKColor	RHC	SA	GA	MIMIC
Mean	419	420	420	403
Maximum	420	420	420	420

TABLE V
MEAN AND MAX SCORE COMPARISON - MAXKCOLOR

From this table, when you put all the scores together, it sort of illustrates that MaxKColor, at least how I defined it, was not a computationally difficult. All the algorithms performed reasonably well. However, the trend has been the poor performance for MIMIC. Like I mentioned previously, MIMIC does not perform well. This likely due to the fact that there is no dependency on any distribution.

Queens	RHC	SA	GA	MIMIC
Mean	1999	2009	2000	1973
Max	2005	2011	2013	1988

TABLE VI
MEAN AND MAX SCORE COMPARISON - QUEENS

From this table, we can see that although all algorithms perform reasonably well, simulated annealing does the best. If I had to pick any solver for this problem, I would likely lean on SA over any other algorithm. MIMIC again performs worse, and significantly so. The best score for MIMIC was not even the average score for any of the other algorithms.

KnapSack	RHC	SA	GA	MIMIC
Mean	1357	3236	4108	3676
Max	3177	3516	4204	4193

TABLE VII
MEAN AND MAX SCORE COMPARISON - KNAPSACK

f1-score	Backpropagation	RHC	SA	GA
Train Score	0.69	0.77	0.51	0.31
Test Score	0.70	0.52	0.47	0.34

TABLE VIII
NEURAL NETWORKS COMPARISON OF TRAIN AND TEST F1-SCORE

When comparing all the randomized optimization algorithms, it is clear that backpropagation is the winner. Evidenced by the fact that training score was essentially the same as the test score for f1, it is a good indicator that the bias-variance trade-off is met somewhere in the middle when using backpropagation. This is also true for the genetic algorithm's iteration, but it performed only half as well.

The poor scores from the randomized optimization algorithms indicate that they are likely better suited for other types of problems: ones that involve discrete optimization. Although they work similarly to maximize a function, backpropagation works better in the context of neural networks, and the other four algorithms work better when faced with combinatorial problems. That is to say, the term "better" and "best" are relative to the problem we tried to solve.

REFERENCES

[1] Jeremy Smith and Jane Doe. Pima indians diabetes dataset. https://www.kaggle.com/uciml/pima-indians-diabetes-database, 1988. Accessed: 2023-09-01.

[?]