Joshua Philipose

Dr. Charles Isbell

CS 4641

3/3/19

Randomized Optimization Analysis

Introduction

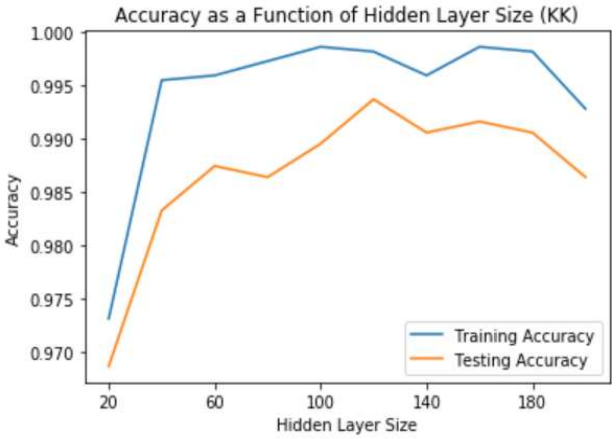
This analysis of randomized optimization algorithms is divided into two parts. The first part covers the performance of neural networks whose weights have been set by randomized hill climbing, simulated annealing, or genetic algorithms. Neural networks are trained with backpropagation, but in an effort to explore these randomized optimization algorithms, I ran various hyperparameter experiments to compare the performances of the resulting neural network node weights. The dataset used for the first part is detailed below. The second part of this analysis showcases the strengths and weaknesses of MIMIC and the previous three algorithms on three different optimization problems: knapsack, n-queens, and the travelling salesman problem. Each algorithm was run with the optimal hyperparameters as determined by the previous experiments. Functions were evaluated by accuracy as calculated by the percentage of instances classified correctly, or fitness function. All algorithms were run in a Jupyter notebook and implemented by scikit learn and a python library named mlrose. The test/train split was 75/25 for all algorithms with random selection. Graphs and scores may vary slightly when running the accompanying code, but this is due to random initialization and the trends depicting in this analysis still hold.

Classification Problem Recap (Part 1)

The king-rook vs. king-pawn binary classification problem attempts to determine if white, who has both a king and a rook, can win the game against black, who has a king and a pawn. However, the twist that makes this problem interesting is that black's pawn is always on a7 – one move away from promoting to a queen. As the problem's premise is that it's white's turn to move, the problem essentially boils down to the ability of white in preventing black's pawn from promoting while maneuvering a checkmate or draw. There are 3196 instances, almost evenly split between 1669 instances with a white win and 1527 instances with a black win. For each of these instances, there are 36 attributes describing the board and piece positions. Because of this dataset's use of true/false values and a column of three categorical values, I had to use one-hot encoding to transform the dataset into numerical format to run on algorithms provided by scikit learn and mlrose. This dataset was provided by UCI's machine learning database.

Starting Point

Fortunately, in beginning our experiments, we could use our prior knowledge from Project 1's analysis to help set baseline accuracies and ideal setups for the neural network hyperparameters. In figure 1 below, you'll notice that a 20-node hidden layer had pretty much the same accuracy as a 100 node hidden layer, and was much faster to train. I confirmed this in the project again – a 20 node hidden layer had an accuracy of .9833, while a 100-node hidden layer had an accuracy of .9864. This is not much of a difference at all, and I noticed a sizeable speed boost when using only 20 nodes in our randomized optimization experiments. It makes sense that using 20 nodes is much faster for randomized optimization, as there are less



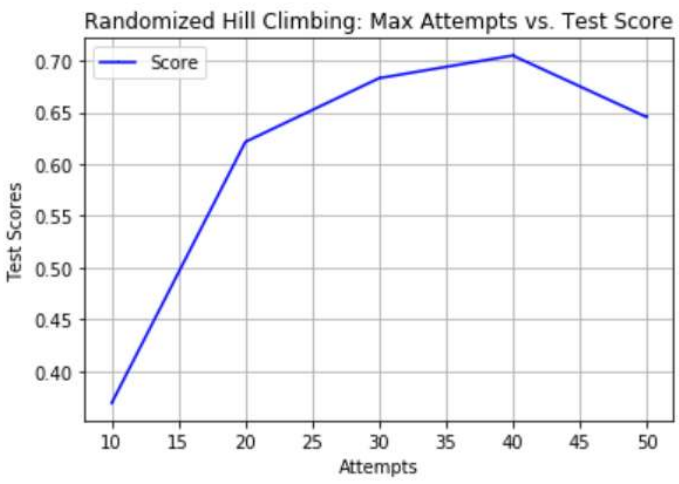
weights to optimize. I had experimented in Project 1 to find that one hidden layer was optimal in terms of accuracy vs. runtime, and that the target function could be accurately modeled with just one hidden layer.

Part One

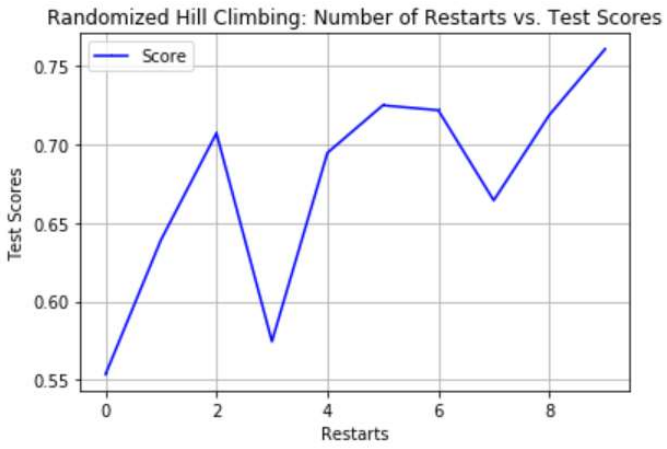
Randomized Hill Climbing (with Random Restarts)

Randomized hill climbing is a randomized optimization function that works by randomly initializing itself, and then attempting to find the maxima or minima of a target function by altering its inputs for that function. Essentially, we pick a random point, then scan a range of neighboring points, move to the point with highest fitness (in our case), and repeat till an optima or the maximum number of iterations is reached. For the purpose of our experiments with randomized optimization, our fitness function is going to be maximizing accuracy, which is similar to minimizing sum of squared error. Essentially, if we are initialized into a certain basin of attraction, we'll climb the corresponding hill. To kick off our experiment, I ran randomized hill climbing with default hyperparameters to determine the weights for the 20-node neural network, achieving a test score of .53 in .9 seconds.

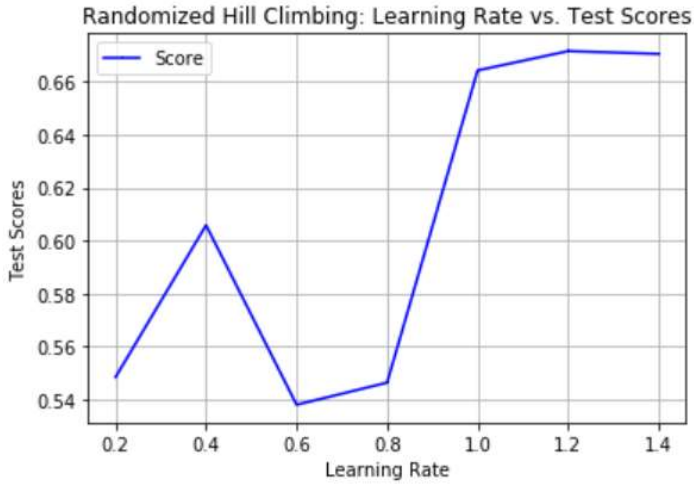
In randomized hill climbing, the maximum attempts hyperparameter is the number of times the algorithm will attempt to find a better neighbor to move to within its neighborhood range. The results below are very intuitively explained: as maximum attempts increase, there's a higher chance that we'll eventually find a more optimal neighbor each time, but it takes more time to sample more neighbors in each iteration. There's a clear positive correlation between both attempts and test scores, and attempts and time taken in the graphs below, so our intuition is bolstered by our results.

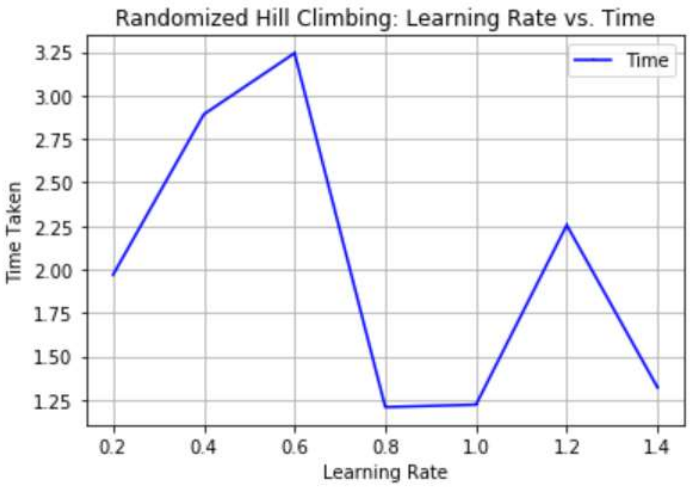


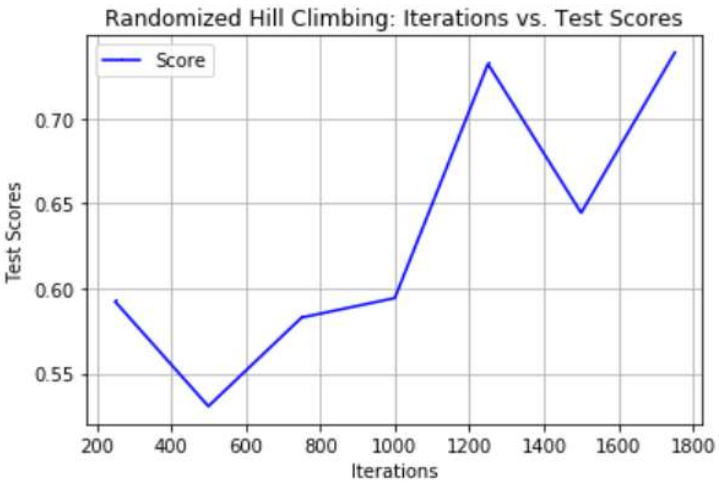
The next hyperparameter I chose to vary was number of random restarts. As with all algorithms in the hill climbing family, randomized hill climbing suffers from the relatively high probability of getting stuck in local maxima. In order to combat this, we take a fairly straightforward approach: we run randomized hill climbing multiple times with varying random initializations to see whether the answer will change. If our answer is consistent every time, we are fairly sure that we are ending up in the global maxima. If our answer varies from run to run, we know that our algorithm is getting stuck in local maxima, and we can tweak other hyperparameters to fix this. However, by changing number of random restarts alone, we can maximize our chances of finding the global maxima by keeping track of the inputs we used to achieve the highest score on the fitness function over all runs. As you can see in the results below, as we increase the number of random restarts, our test scores generally increase. Using nine restarts yielded an accuracy of .76, much higher than the .55 accuracy of only one run of randomized hill climbing. The changes in test score accuracy throughout the graph can be attributed to the random initial starting point – sometimes we just repeatedly get unlucky and end up with a worse score, but as you can see the general trend is that a higher number of restarts means a higher test score. Of course, there's no free lunch – if we run our algorithm more times, it logically follows that our total runtime will be longer. This can clearly be seen in the number of restarts vs. time graph below, where time is plotted in seconds.



Another important hyperparameter for randomized hill climbing is the neighborhood size. Neighborhood size is the size of the neighborhood explored in each iteration of the algorithm. As the graphs below show, as neighborhood size (parameterized as learning rate in mlrose) increases, the test score generally increases, and the time to train decreases. Upon first glance, this seems perfect – why not just increase the learning rate to the maximum amount? However, if our learning rate (and therefore neighborhood size) gets too large, we'll start trying to find which way to go in the entire domain of the function, meaning our efforts will be spread far too thin and that convergence at an optima not guaranteed.







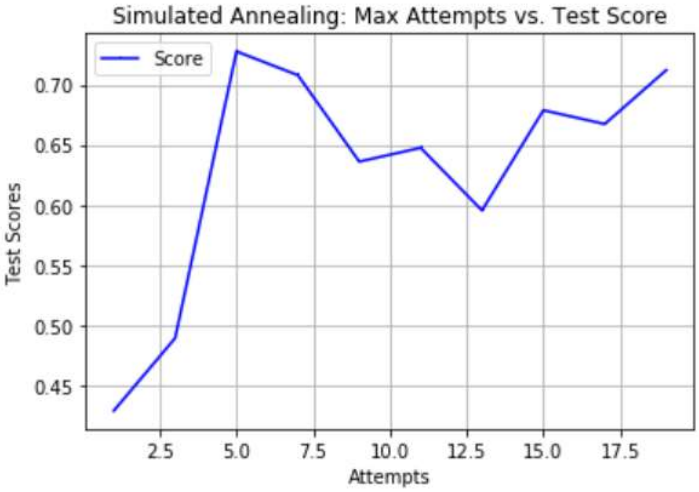
Lastly, as randomized hill climbing is an iterative algorithm, we naturally want to see the effect of iterations on its accuracy. In the graph to the right, we can see that as iterations increase, our test scores increase as well. This is because in each iteration, we move "upward" on our current hill, so as our iterations increase, we're afforded more upward mobility, which translates to a more optimal position on the fitness function and therefore better weights for our neural network. This upward trend will continue until there are enough iterations given to us that we will reach the top of whatever hill we're currently on throughout the fitness function as determined by our current optimization problem. However, being at the top of the hill doesn't mean we're at the optimal solution – remember that we can get stuck in local maxima, upon which increased number of iterations will not help our case. This is likely what happened at the 1500 iterations datapoint in the graph to our right.

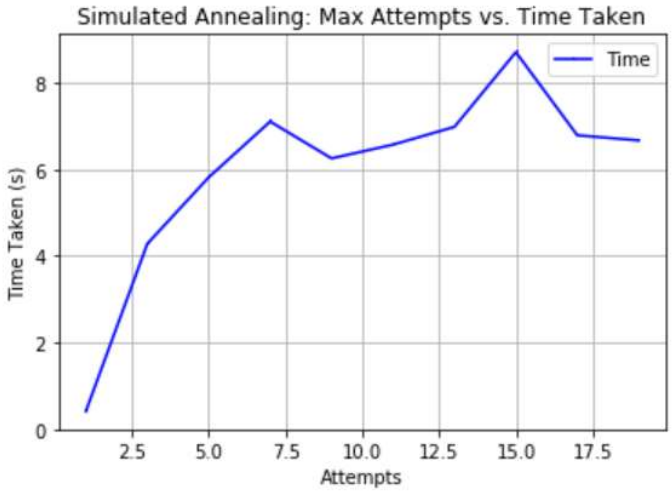
Simulated Annealing:

Simulated annealing is a randomized optimization algorithm built off of randomized hill climbing. Simulated annealing starts off just like randomized hill climbing: it initialized at a random point. Then, the algorithm samples points in a neighborhood around it, and if the sampled point has a higher fitness function value than the current point, we jump to the sampled value. However, where simulated annealing differs from randomized hill climbing is that if the sampled point has a lower fitness function value than our current point, we may still jump to it with a probability determined by the current "temperature" T. If the temperature is high, we are more likely to jump to a lower point, and if temperature is low, we are less likely to do so. The results in simulated annealing acting like a random walk when temperatures are high and acting like hill climbing when temperatures are low. The benefit to this algorithm is that we explore more areas of the fitness function before climbing up a hill, theoretically leading to a lower probability of becoming stuck in local maxima. Simulated annealing therefore attempts to strike a good balance between exploration of hills and exploitation of a specific hill. To start my experiments with simulated annealing, I ran the algorithm with the default hyperparameters, resulting in an accuracy score of .506 in 7.5 seconds – barely better than chance, and quite slow compare to randomized hill climbing.

Further along in my experiments with simulated annealing, I realized that randomized hill climbing usually performed better, and at quicker speeds. This is probably due to my specific dataset, as if there aren't many local maxima for randomized hill climbing to get stuck in, simulated annealing is wasting a lot of time in its "exploration" phase. It's likely that the fitness function has a relatively wide basin of attraction for its global maxima, meaning that randomized hill climbing will do better as it focuses on exploiting the current hill and climbing as high as possible instead of exploring multiple hills.

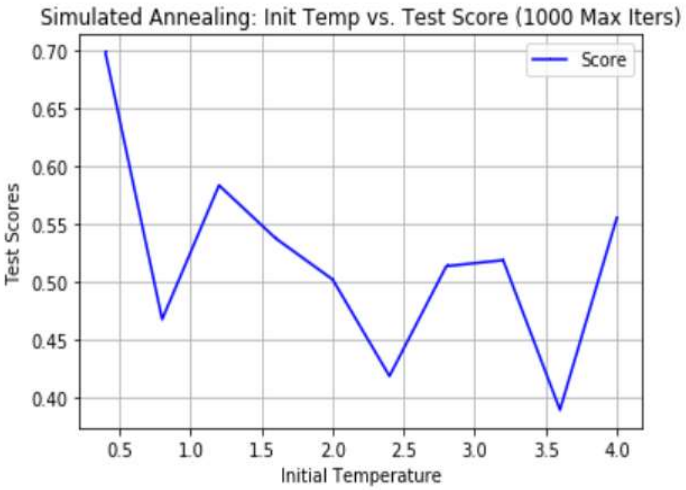
To increase our accuracy, I started moving around max attempts as that had increased the performance of randomized hill climbing. Fortunately, the similarities rang true, and a higher number of attempts correlated to both higher test scores and long time taken, as you can see in the graphs below. The reasons for this effect are covered in the previous section on randomized hill climbing. The initial spike in test accuracy at the 5 attempts datapoint is most likely due to a beneficial randomized initialization, also known as a lucky initial guess.

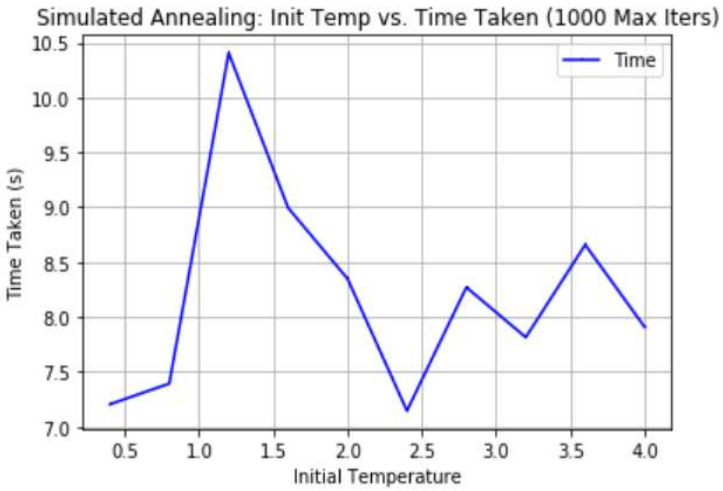




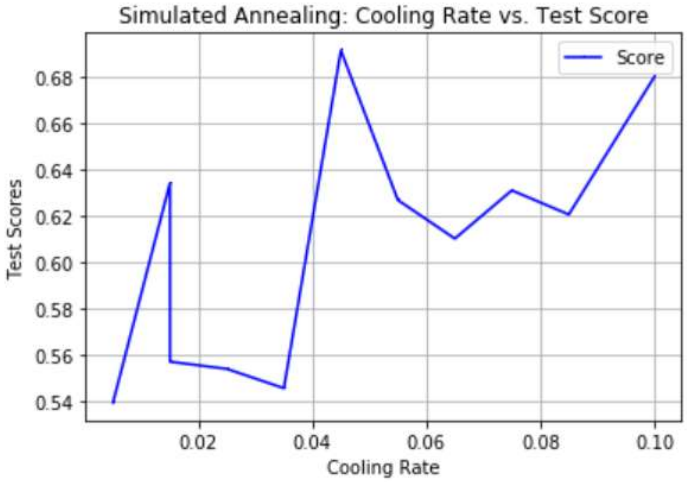
Now that we've gotten a good feel for a baseline, let's vary the initial temperature. Initial temperature is the initial value of T before any cooling occurs. Remember, a higher value of T means a higher probability that we'll explore a point lower than our current point. Therefore, a higher initial value for temperature means that we'll be in the "exploration" phase for longer than usual, and we may not have the time to "exploit" the hill we're on when the temperature cools down because of limitations imposed by the maximum number of iterations. This reasoning explains the graphs below quite well. Although the initial temperature vs. time graph initially looks odd, it's important to remember the impact of chance and that the y axis is limited to relevant numbers and doesn't show the scale of the fluctuations as well as it should. If we look at the other points on our initial temperature vs time graph, we'll notice that while there's a slight uptrend, the line hovers around 7.5 to 8.5 seconds. This isn't a significant increase in time, which makes sense because the amount of iterations and calculations

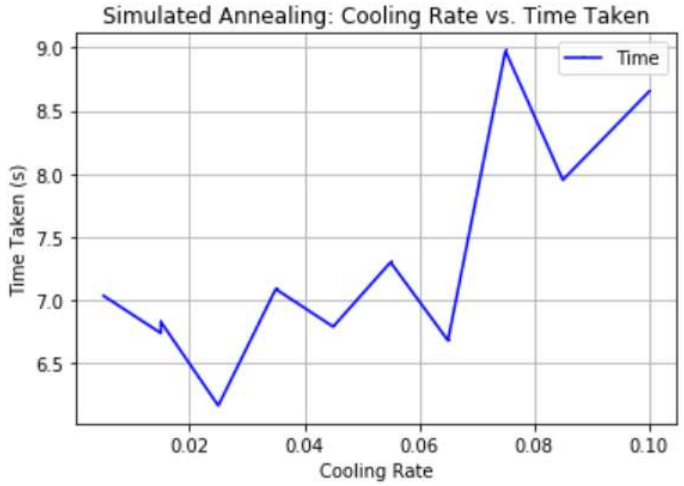
are constant, so temperature shouldn't influence the runtime much. The fluctuations we see here are due to the natural fluctuations in system runtime as my laptop runs other applications or processes.



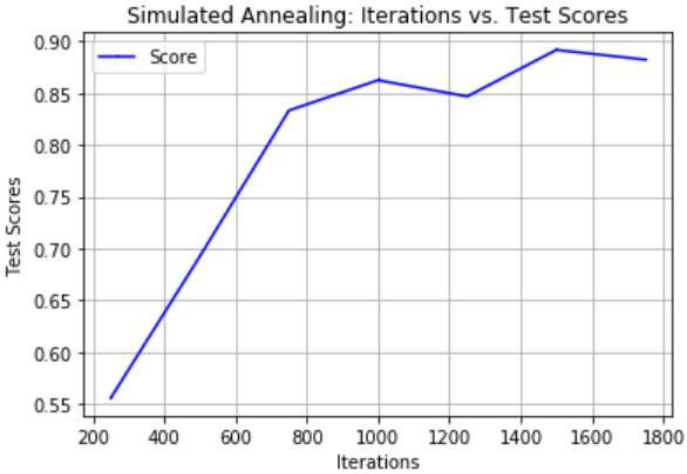


If the lower accuracy is due to our algorithm not having the time to "cool down" and exploit the hill it's on before max iterations run out, why not increase the cooling rate and see if that has an impact on test score? Below you're see the outcome of this experiment: as cooling rate increases, test score generally increases as well. This result is consistent with our previous hypothesis – by cooling down our temperature more quickly, we increase the amount of time spend exploiting the current hill and decrease the amount of time spent exploring other hills. This reinforces the notion that our fitness function has relatively few local maxima, and a relatively wide basin of attraction for the global maxima.





Let us then focus once again on the impact of iterations on our test score. As mentioned in the two previous experiments, number of iterations is the limiting factor that allowed us to analyze the impact of initial temperature and cooling rate. As we conduct this experiment, we see the beautiful graph to the right. As predicted, number of iterations improves the test score to the highest levels yet recorded until further iterations are redundant because our algorithm has reached the top of the hill. This graph shows us that 1200 iterations is the amount of iterations needed to both explore and exploit our current fitness function. At this point, simulated annealing actually performs significantly better than randomized hill climbing, because there's enough



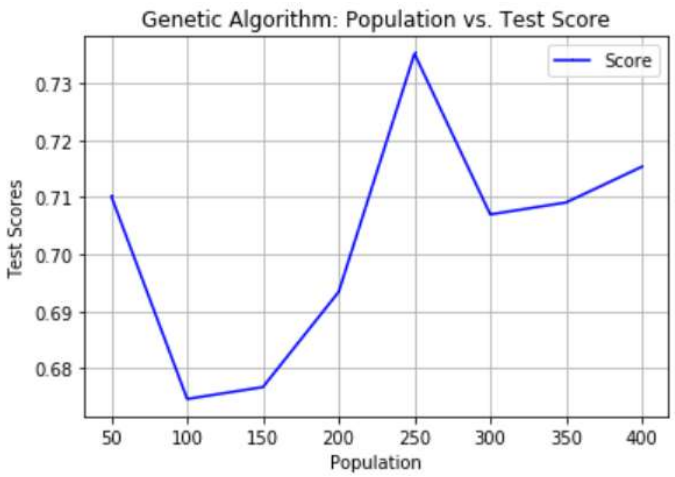
iteration time to both explore and exploit the topology of the fitness function. As is expected, the runtime increases almost linearly with iteration number, so I spared you the graph. The need for domain knowledge is highly evident throughout all of these simulated annealing experiments.

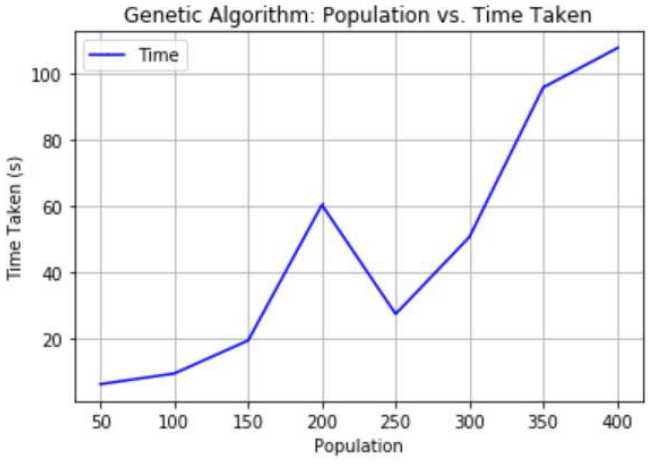
Genetic Algorithms:

Genetic algorithms are based on the principles of biological evolution. Various assortments of neural network weights, in our case, are randomly initialized, evaluated by a fitness function, and then bred together through a crossover function. This breeding occurs between a sample of the population, generally very heavily skewed towards the half of the population that performs best according to the fitness function. The technical implementation of the crossover function varies, and it is important to ensure that the crossover function for a particular problem provides meaningful results with respect to the problem. Furthermore, additional randomness is added in the form of mutations to encourage exploration. As per usual, I ran a neural net trained with the default genetic algorithm on mlrose to establish a baseline before further experimentation.

The result was our best test score with the worse runtime: .69 accuracy in 21 seconds. This exhibits a clear tradeoff between time and complexity. A good reason for higher accuracy in genetic algorithms is the inherent safety net in genetic algorithm implementation. With each generation, the bottom half of performers are removed, meaning that unless a highly unlikely severe negative mutation occurs, generations will not regress back to poor performance over time. This is in direct contrast to simulated annealing, which may jump off a global maxima if the temperature is too high. In this way, genetic algorithms combine the "commitment" attribute of randomized hill climbing (as we know, if randomized hill climbing finds a global maxima it won't leave it for something lesser) with the exploration skill of simulated annealing.

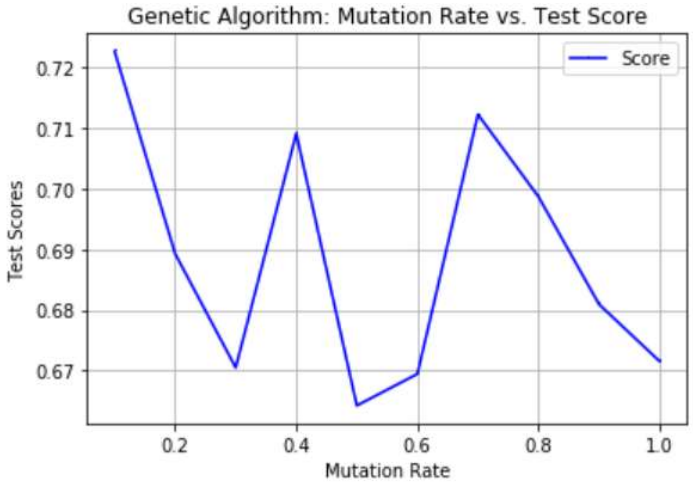
With our groundwork in place, let's take a look at population size. Below is the result of varying the initial population size from 50 to 400 in steps of 50. As you can see from the first graph below, population amount didn't have a clearly defined trend, and the test score for a population size of 50 is not significantly different from that of a population size of 400. What this reveals is that there's enough "exploration" in the randomized initialization of 50 individual weight sets for our specific classification problem, and that future population increments are not necessary because test score changes will be due to chance. However, increasing population size does come with a strongly defined increase in runtime, as the fitness function and crossbreeding has to be run many more times as population increases. With these facts in mind, it doesn't look to be worth it to increase population size too much for our specific problem, as it doesn't give us better results but takes significantly longer – a population of 400 takes over ten times as long to run as a population of 50.

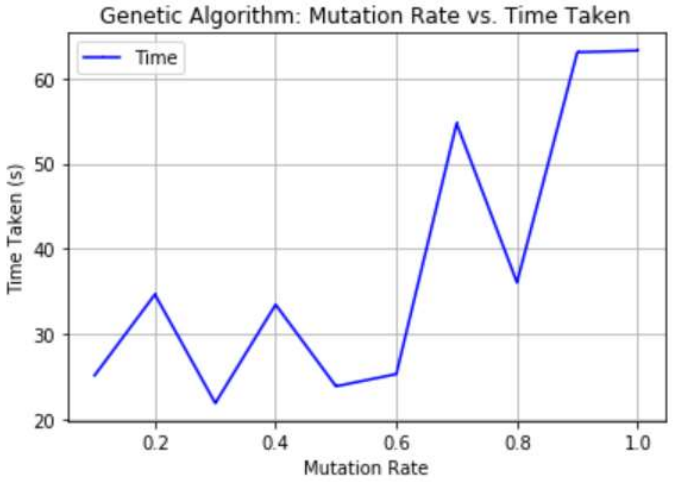




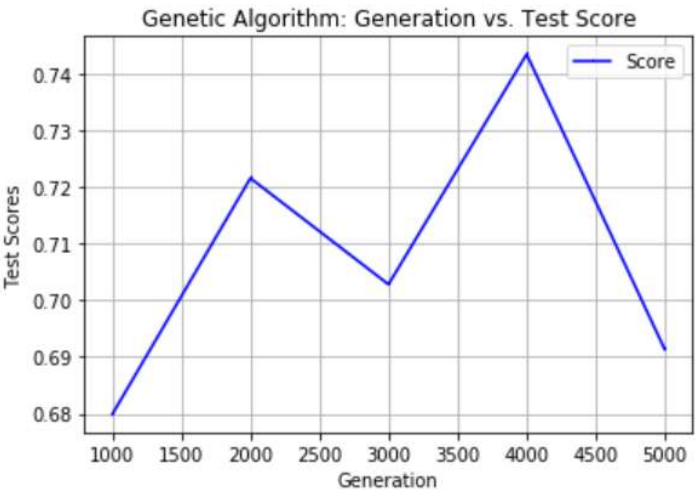
Since we shouldn't alter our crossover function away from a meaningful function, let's instead focus on something more quantifiable: the effect of mutation rate on test score and runtime. Though it's a bit difficult to see in the graph below, our genetic algorithm seems to generate poorer weights for our neural network as mutation rate increases. This is most likely due to the fact that as mutation rate increases, any benefit garnered by the crossover function and elimination of worse

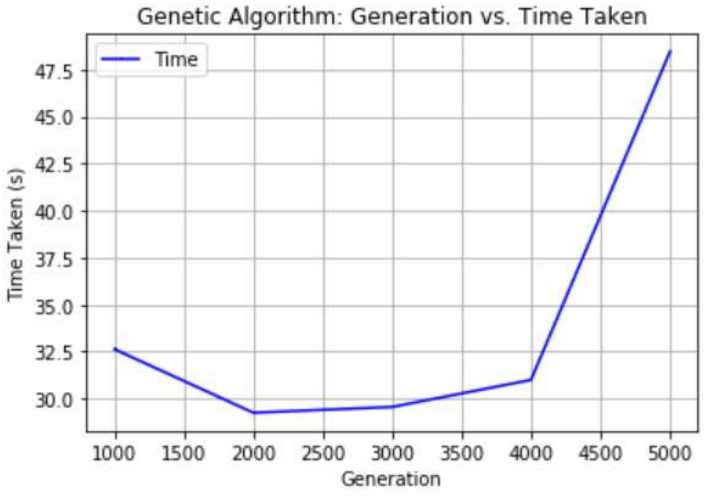
performing weight sets is offset by the randomness introduced by a higher mutation rate. The mutation rate vs. time graph is also fairly simple to explain, as when mutation rate goes up, the amount of mutations that need to be calculated and executed also goes up.





Finally, because genetic algorithms are just as iterative as the previous two algorithms, we should look at the effect of these iterations on test score and runtime. In the graph below, we can clearly see that aside from the last datapoint of generation size 5000, there's an upward trend in test scores. This makes sense because as each generation passes, the best sets of weights are refined and the worst ones are eliminated. However, what most likely happened between the 4000th generation and the 5000th generation is a severe set of mutations (unlikely) or overfitting to the training data (much more likely).





All in all, the "best" algorithm for our specific classification problem depends on how you define best. If runtime is very important to you, the randomized hill climbing algorithm performs with the best. If one defines best as highest test score, simulated annealing with a higher maximum iterations cap will give you the best accuracy, at the cost of might higher runtime. Regardless, our experiments have shed great insight onto these three algorithms.

Part Two

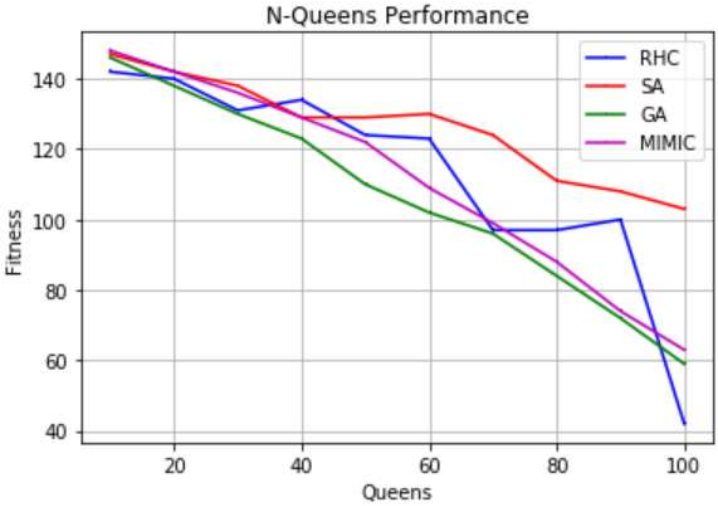
In the second part of this analysis, I explored three separate optimization problems in the hopes of identifying and highlighting the relative strengths and weaknesses of randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC. The N-Queens problem is solved best by simulated annealing, the knapsack problem is solved best by MIMIC, and the traveling salesman problem is solved best by genetic algorithms, if our definition of best is highest test score possible.

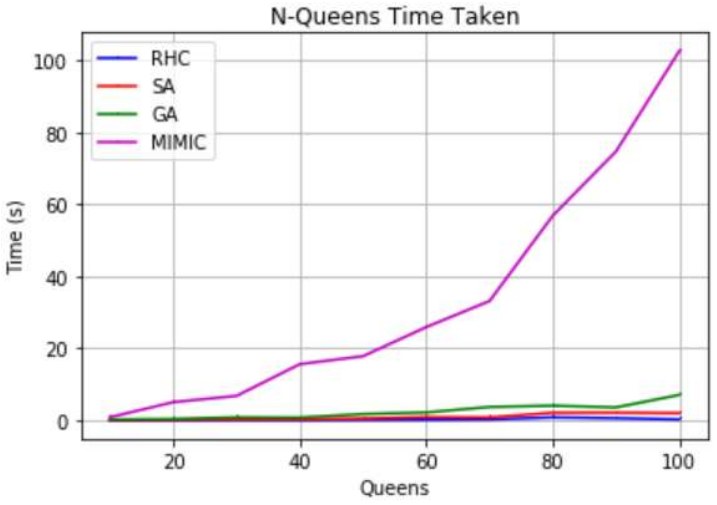
MIMIC:

Let's take a brief paragraph to introduce MIMIC before we look at its relative performance in the next few sections. MIMIC is an improvement of genetic algorithms developed by Dr. Charles Isbell. The main performance boost comes from the fact that a probability function prioritizes certain attributes over others during crossover, and codependent factors are changed with each other accordingly. The sample selected for crossover is weighted towards members of the population with the most potential of resulting in a globally maximal strain. In most cases, MIMIC is more accurate than genetic algorithms in situations where genetic algorithms would do well, but it typically is much more computationally expensive as a result of the probability functions (no free lunch theorem).

N-Queens Optimization Problem:

The N-Queens problem is the problem of finding a placement of N queens on a N by N chessboard such that no queens are attacking the others. The N-Queens problem has discrete states with many local maxima as moving a queen one space may move it into the line of fire of another queen. Our fitness function looks at the number of queens that are successfully placed without attacking another or being attacked. The implementation of this problem in mlrose actually has the fitness function f(x) equalling the number queens who are attacking each other, but I quickly set that amount to 150 – f(x) in order to have a fitness function we could maximize. I picked 150 because it would fit the specific range of hyperparameters I selected. This is an interesting machine learning problem because it deals with a large number of local maxima in a rules based, discrete system. Personally, I find it interesting because I like chess. Below is the performance and time of each of the four randomized optimization algorithms as a function of the number of queens.

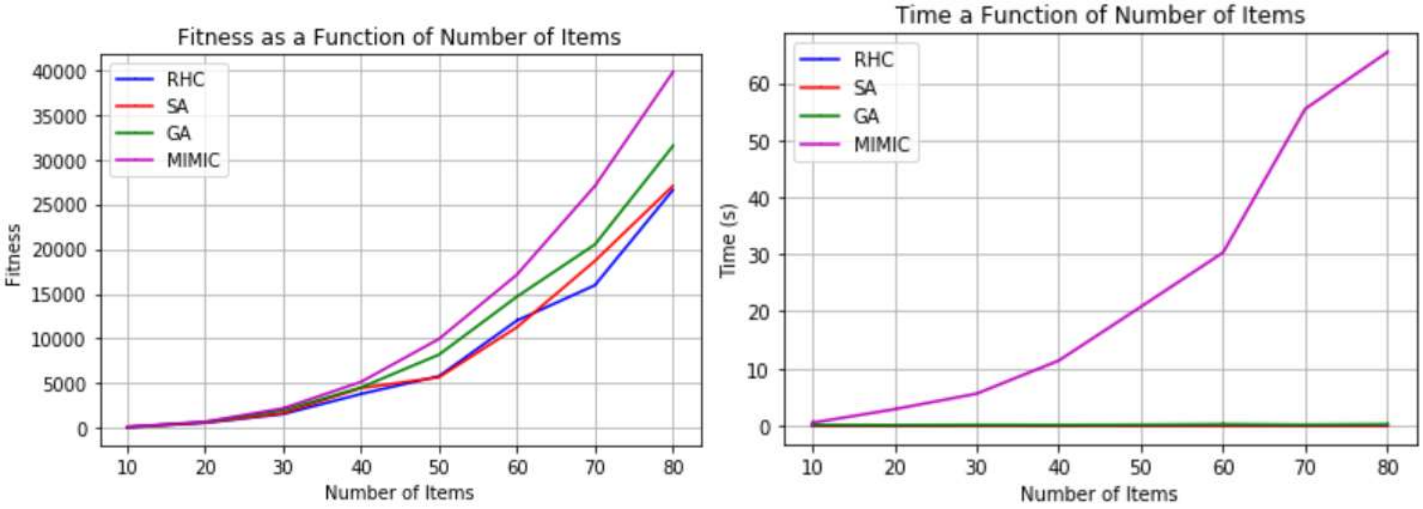




As you can see in the graphs above, simulated annealing works the best across various Ns, if we define best as the highest fitness function for each individual N. This is because simulated annealing is the least hindered by the realities of its implementation. Randomized hill climbing is hindered by the fact that as you increase the number of queens, moving a queen one space will likely move it into the line of fire from other queens, therefore causing a plethora of local maxima. Genetic algorithms and MIMIC by extension are hindered because crossover is not meaningful in this context. Taking half the positions of queens from one set and half from another may result in conflicts, and in general each queen position is no more important than another. Therefore, the primary way of increasing fitness function through generations is through dropping the lower half of the population, rather than the crossover function. Simulated annealing solves the local maxima problem as explained in the simulated annealing section of Part 1.

Knapsack Optimization Problem:

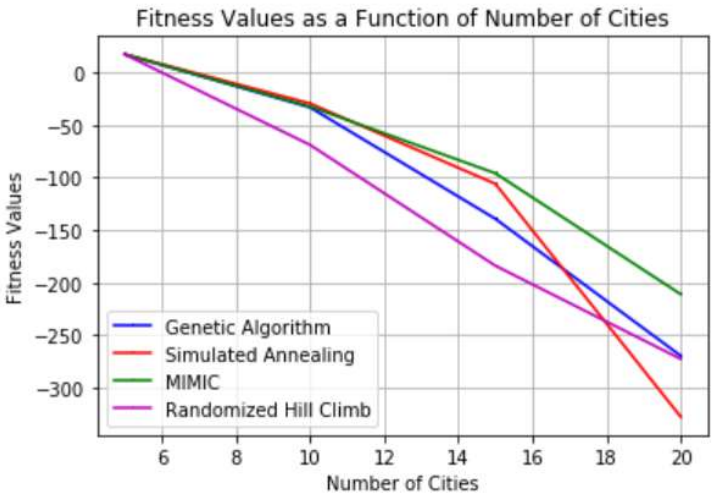
The knapsack problem is an optimization problem where we try to fit as many items into a "knapsack" with a maximum weight limit. Each item has its own weight and value – we're trying to maximize the resulting value of our knapsack of selected items by either taking an item or not until our knapsack can no longer hold more items. This problem is quite interesting because it tackles the problem of scarce resource allocation and has many applications to real world financial problems. Below is the performance and runtime of each of the four randomized optimization algorithms as a function of

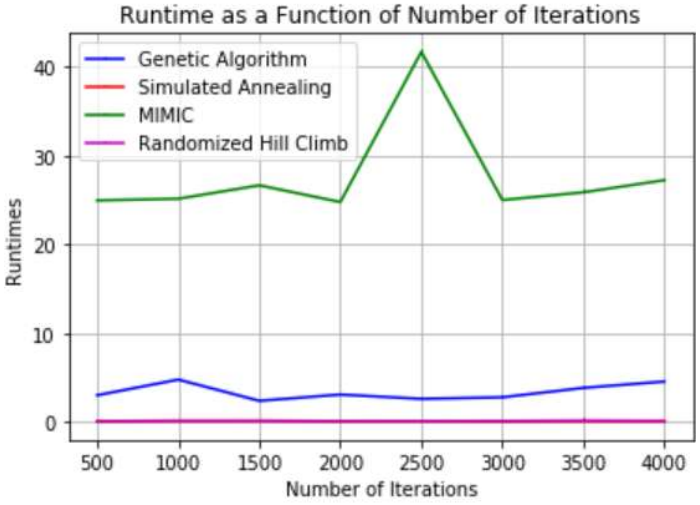


As you can see above, MIMIC is clearly defined as the optimal function in for this problem, with genetic algorithms coming in as a close second. The reason for this is that MIMIC and genetic algorithms select for the high value to weight ratio items as generations weed out the lesser fit members of the population. In addition to this, MIMIC understands the codependence of several items (namely that the weight of some items will preclude other items from being added) and intellgiently plans around this. In this matter, both genetic algorithms and MIMIC are almost analogous to dynamic programming, in that they attempt to find a globally maximal solution from locally maximal results. The reason that randomized hill climbing and simulated annealing perform poorly in relation is that they cannot model the dynamic programming element of this optimization problem as they don't learn from locally optimal solutions to finish filling up their knapsack optimally. Although MIMIC is also very computationally expensive, I believe the increase in the fitness of the end result more than justifies it.

Travelling Salesman Problem:

The travelling salesman problem is a graph traversal problem where one attempts to visit all nodes and return to the starting node with the least travel cost possible. The travel cost can be thought of as a distance function, or as an actual cost to take a route between nodes. The travel cost as implemented in my code is Euclidean distance between nodes. Our fitness function for this problem is very straightforward: a lower travel cost is better. The traveling salesman problem is directly applicable to real world shipping and freight problems where the shipment vehicle must return to its home base, and various other similar business functions where a resource must navigate a maze of travel costs before returning to home base. Below is the performance and runtime of each of the four randomized optimization algorithms as a function of number of cities and number of iterations, respectively. Note that the fitness function values on the Y axis are negative because mlrose implemented fitness as distance, so I flipped the fitness function by subtracting it from a constant in order to minimize distance.





As you can see above, MIMIC works better than the other algorithms in terms of maximizing fitness function as number of cities increases but is much more computationally expensive as can be seen in the runtime as a function of iterations graph. In this case, MIMIC and genetic algorithms perform better than simulated annealing and randomized hill climbing because they comb the search space much more efficiently, partially due to their discarding of the lesser fit members of the population. For this problem, the search space is vast as there's an exponential amount of travel cost totals as the number of cities increases. However, I believe that genetic algorithms perform better than MIMIC in this optimization problem because its runtime is a fraction of MIMIC's runtime for a similar ending fitness function score.

Conclusion:

Part 1:

While I'm quite pleased with the tuned performance of our various algorithms in Part 1, it is a humbling fact that none of them were able to outperform the backpropagation algorithm for training neural network weights. We saw many tradeoffs in the various types of randomized optimization functions we explored, however, and though we may not want to apply them to neural network training in practice, we've definitely gained a strong intuition for how they work and where we may be able to use them practically. It is important to note that neural network weights are continuous and real valued, meaning it's far more difficult to limit the search space. However, continuous values also mean smooth fitness functions, which can lend itself to differentiation in more complex implementations of the previous hill climbing algorithms. In addition, far more minute adjustments can be made by these optimization algorithms, which logically implies higher accuracies if sufficient resources are made available to the algorithm to take advantage of this feature.

Part 2:

The key lesson learned throughout our experimentation in Part 2 is that domain knowledge of what algorithms to apply to different problems is of utmost importance. All four learning algorithms have their strengths and weaknesses and are therefore suited to a wide range of differing problems – not even the brainchild of Dr. Isbell can be the best at every problem it encounters. All in all, I've learned an inordinate amount from this assignment, and am excited to put this knowledge to good use in future projects and in industry.