Don Branson

Professor Charles Isbell

CS-7641 Machine Learning

11 October 2019

Randomized Optimization Assignment 2

Overview:

The purpose of this assignment is to perform detailed analysis on four Randomized Optimization algorithms discussed in lecture and to use three of these algorithms in place of backpropagation for a Neural Network. The assignment requires picking three interesting datasets that highlight the advantages of each algorithm, training the randomized algorithms on that data, and documenting the analysis. The challenge presented in the assignment is to optimize fitness function scores and accuracy by evaluating algorithms over a range of hyperparamters and fitness functions. The mlrose python library and abagail java library was heavily utilized in this assignment. Although mlrose offers several optimization problem spaces, only the discrete optimization (maximization) problem space is considered in this analysis. This problem space defines basic functionality required for discrete input strings, and, most importantly, it defines the neighbor functions. In addition to the optimization space, the following mlrose fitness functions were selected:

- **Flip Flop:** The Flip Flop fitness function uses a one-dimensional bit string of length x to evaluate the number of bit changes in the bit string. The optimal fitness for this function is the length of the bit string with each contiguous bit different than the previous. For example, a 10 bit string would be optimal with this configuration: 0101010101. The minimum fitness for this function is 0 which occurs if all the bits are either 1 or 0 (1111111111 for example). This problem is interesting because the fitness score depends of the ordering of the bits indicating that a good algorithm would need to represent structure to quickly converge on a solution. This problem space was chosen to highlight the advantages of the of the Mimic algorithm
- **Four Peaks:** The Four Peaks fitness function uses a one-dimensional bit string of length x to evaluate the leading 1s and trailing 0s. The function counts the leading 1s and 0s and then evaluates if the count of both 1s and 0s is greater than the threshhold parameter. If the function is greater than the threshold parameter then the fitness function gets a bonus score greater equal to the length of the bit sting on top of the max count between the 1s and 0s. For example, if the bit string 1110101000 with a threshold value of .1 would get 10 points (length of the bitstring) plus 3 points (max of the 1s and 0s). The bit string 1010101000 would only get 3 points (max of the 1s and 0s). In essence, the point of this problem space is to optimize the number of leading 1s and trailing 0s and most importantly to the count of those leading and trailing bits above a threshold. This problem is interesting because structure is critical and the attraction basin is very small due to the threshold bonus step function. This problem space is hard to explore and the leading 1s and 0s have mutual information that needs to be handled in the algorithm. This problem space was chosen to highlight the advantages of the Genetic algorithm.

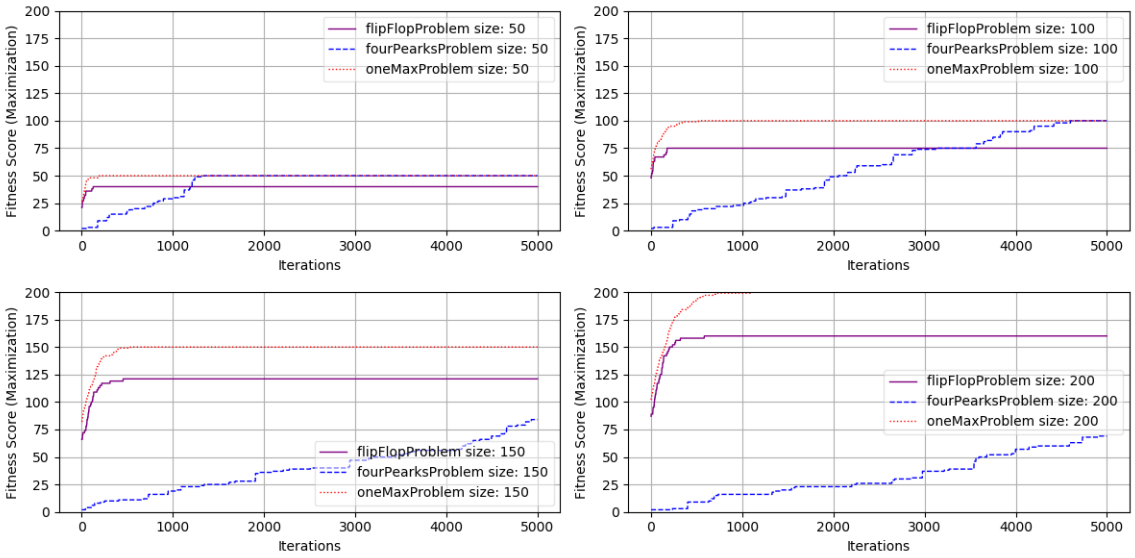
- **One Max:** The One Max fitness function uses a one-dimensional bit string on length x to evaluate the total number of high bits (1s) in the string. The optimal fitness for this function would be all 1s (1111111111 for example). The minimum fitness for this function would be all low bits (0000000000 for example). This problem is interesting because it is conceptual much simpler than the other two algorithms. The ordering of the bits does not matter and it does not matter what sequence is followed to move from a random state to the optimal state so long as eventually the algorithm can flip all the bits high. Given this information, it is reasonable to assume this problem space has a relatively high attraction basin and not many local maximum. The global maximum should be straightforward to calculate. This problem was chosen to highlight the advantages of the Simulated Annealing and Random Hill Climbing algorithm because it does not require structure and the algorithm should not get stuck.

Each algorithm that will be further explored in the next sections will attempt to traverse from a random state to an optimal state for each fitness function in the problem spaces defined above. Each of the four algorithms attempts to converge to the optimal fitness through different means and structures which can take varying amounts of time and iterations. Some algorithms may not converge for some datasets. Some algorithms may converge, but in a very slow fashion. Some algorithms are better suited for certain problems and this, along with hyperparameter optimization, will be the focus of next sections.

Randomized Hill Climbing

The Randomized Hill Climbing Algorithm attempts to solve problems spaces by randomly choosing a bit to switch to a random bit value (0 or 1 for this analysis), and then evaluating the new state with the switched bit. If the new state has a better fitness value than the previous state then the new state is copied and used for the next iteration. The function iterates until the maximum iterations are achieved (it does not have other exit criteria). Due to the nature of random bit flipping this algorithm can very easily get stuck at local maximum without finding the global maximum. To help alleviate this issue, mlrose provides a restart option where the algorithm resets and uses a new random state to continue. This option is not considered due to added complexity and the assignment does not call for random restart hill climbing. Instead several runs will be used to determine fitness which will provide a similar outcome.

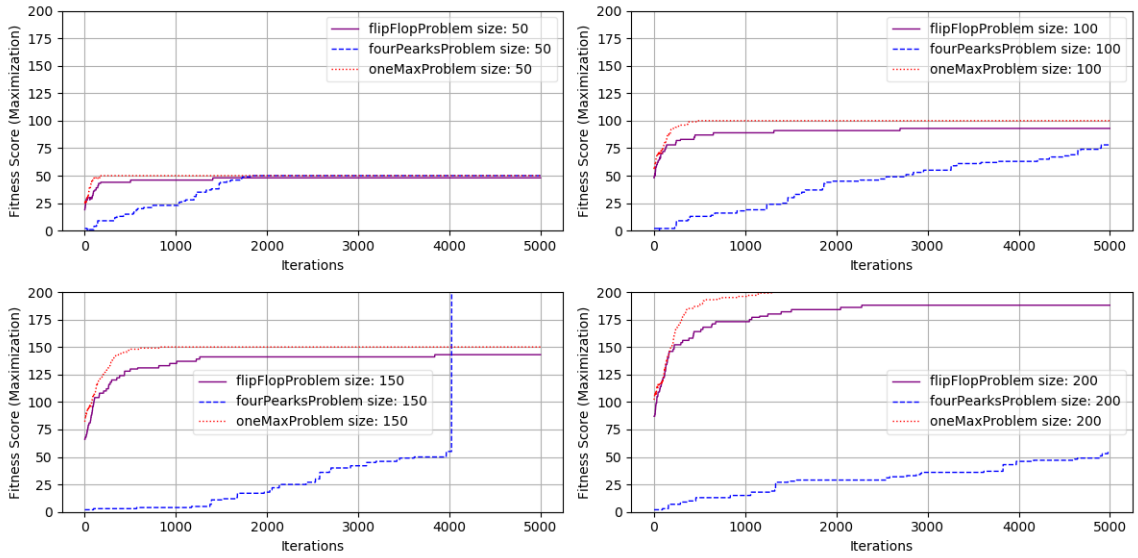
For the Randomized Hill Climbing experiment, the number of iterations and the size of the bit string will be evaluated to determine our algorithm behavior and optimal parameters. Each of the four charts below represent fitness score per iteration for each problem space. The four charts only differ but the length of the input bit string which is in the set [50,100,150,200]. These charts show performance of the datasets for different levels of complexity of the input bit string. It can be seen that as the complexity of the input bit string increases it takes more iterations to converge. Due to the randomness of the neighbor function, increasing iterations did improve the fitness functions for all problem spaces for all bit sizes. The One Max problem space performed very well with fast convergence and achieving perfect fitness function scores. These scores are achieved for One Max because the local maximum and global maximum are the same with the algorithm and problem space combination. The Four Peaks problem did not perform well because it took many iterations and it never reached the optimal fitness for most of the cases. The Randomized Hill Climbing algorithm really has no good way to understand the underlying structure of that problem due the max leading 1s and trailing 0s fitness function.



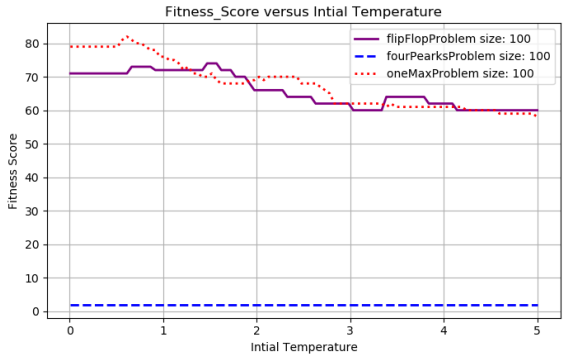
Simulated Annealing

The Simulated Annealing algorithm, much like Randomized Hill Climbing, attempts to solve problem spaces by exploiting and exploring new states probabilistically through a acceptance probability function defined in literature and lecture. This algorithm slowly reduces temperature using the mlrose.GeomDecay function (through the initial temp and decay hyperparameters). When the temperature is hot (high value) the algorithm is more likely to accept states that do not outperform the fitness of the previous state. As the temperature slows the algorithm act more and more like Randomized Hill Climbing. This algorithm should perform similar to Randomized Hill Climbing but it should be less susceptible to getting stuck at a local maximum.

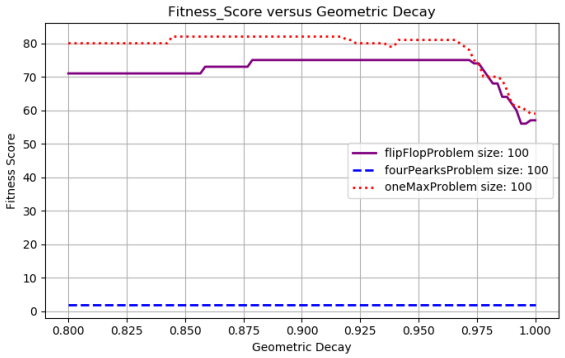
For the first Simulated Annealing experiment, the number of iterations and the size of the bit string will be evaluated to determine the best iteration value to use for further experiments. The charts follow the exact same structure as the Randomized Hill Climbing example. Default values for init_temp and decay in the temperature decay algorithm were used. The results are very similar to Randomized Hill Climbing and for similar reasons related to the randomness int the neighbor function and linear nature of this algorithm that does not store structure. It is important to note that on one of the runs this algorithm was able to get the bonus threshold score in the Four Peak problem space. The exploitation of the medium temperature was able to trigger that response when Randomized Hill Climbing was not. This effect may become more apparent with further parameter turning. It is also important to note that Simulated Annealing took slightly longer to converge on all problem sets. This is also related to the temperature effects.



For the second Simulated Annealing Experiment, a max iteration of 100 was chosen so the algorithms would not already be at maximum fitness. The init_temp and decay parameters will be varied in the mlrose.GeomDecay function. From the Initial temperature fitness curve chart, it can be seen that as initial temperatures increases the fitness score decreases indicating we additional exploitation is not effective. Using an initial temperature of 1, the decay factor was varied in the second chart. From the chart it can be seen that a decay value of .9 is optimal.



Initial Temperature Fitness Curve

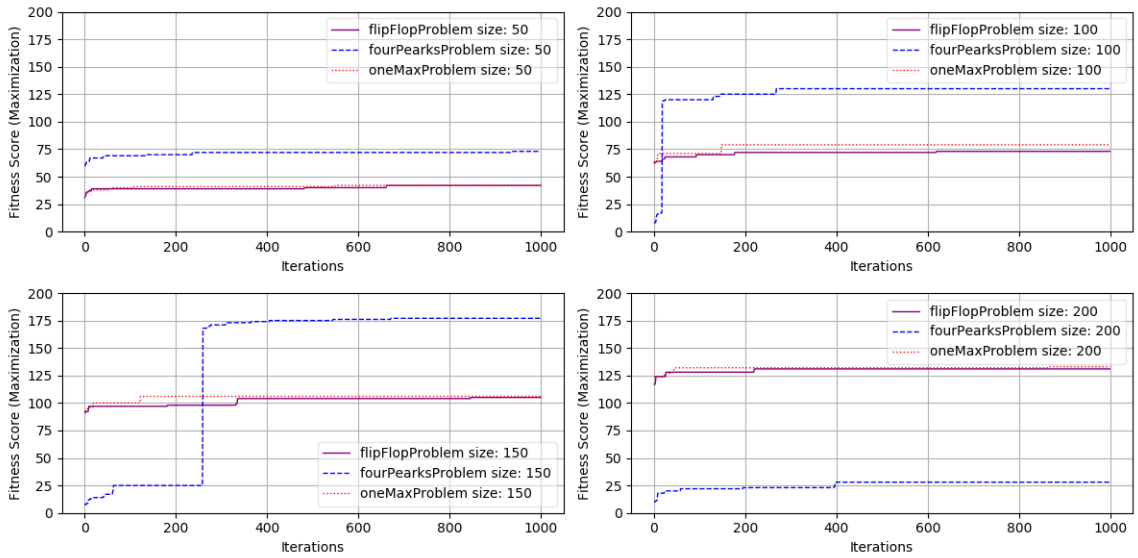


Geometric Decay Fitness Curve

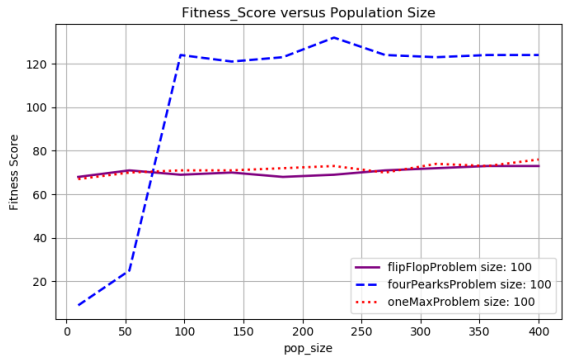
Genetic Algorithms

The genetic algorithm attempts to solve problem spaces by using populations, mutations, crossover, and generations to select the most fit states and removing the least fit states. In the mlrose implementaton, a population of pop_size is randomly created. Each iteration, a new child is created using mutation_prob probability to create the child. The children are created by taking a random (x) number of the leading bits from parent1 and a random number (len-x) of trailing bits from parent2. Then a small number of random bits (set by mutation_prob) in the child are set to random bit values. This child is then put into the population. This process of choosing two parents and creating a child continues for the number of maximum iterations. The algorithm uses the best child based on fitness to determine future states and algorithm fitness.

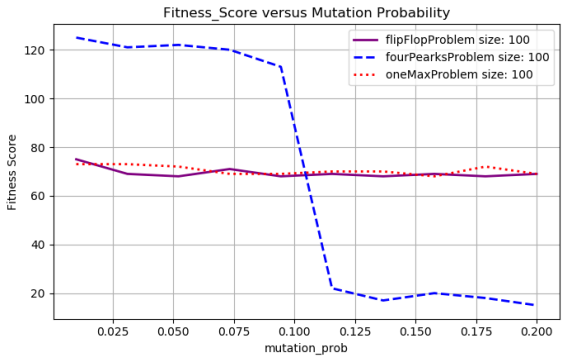
For the first experiment, the number of iterations and the size of the bit string will be evaluated to determine the best iteration value to use for further experiments. For this experiment, the pop_size was set to 200 and the mutation_prob was set to .1, both default values. From the chart, it can be seen that for the longer input bit strings for the problem space (more challenging problems) the longer it takes the Genetic Algorithm to find the bonus threshold value. The longer the bit strings, the more challenging it will be because there is more randomness and more data. It's important to note how well this algorithm performed on the Four Peaks problem space. This is due to the splitting of parents to make children. The splits represent data on the leading and trailing ends of the bit strings which is the area of importance in Four Peaks. It's also important to note that the hardest problem did not get the threshold bonus yet indicating how challenging the problem is. More iterations would likely converge



In the second experiment, a max iteration of 100 and a bit string length of 100 were selected. The parameters pop_size and mutation_prob were varied. From the first chart, it can be seen that larger population sizes and smaller mutation probabilities optimize the problem spaces. It should be noted that increasing population size drastically increase run time so a value of 100 will be used and a value of .05 will be used for mutation_prob.



Population Size Fitness Curve



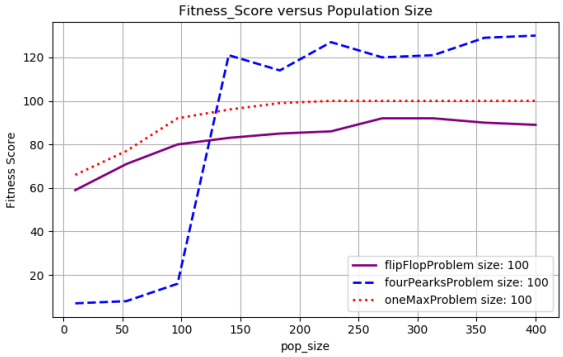
Mutation Probability Fitness Curve

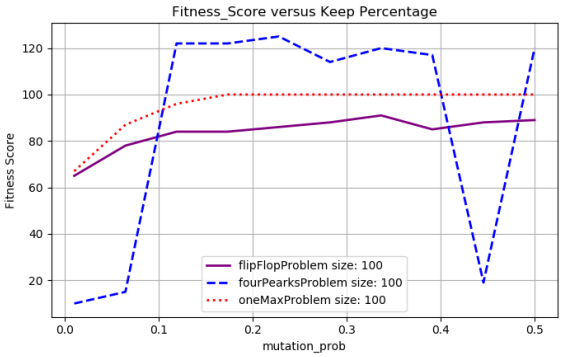
MIMIC

The Mimic algorithm in mlrose attempts to solve problem spaces by using randomly generated populations of pop_size and a keep-pct attribute used to iteratively determine theta through the pop_fitness array. As the population fitness array (pop_fitness) continue to get better then the keep_percentage attribute will continue to create a higher value of theta. As Theta gets higher, only the really good samples will remain which will in turn start the positive feedback cycle all over again. These iterations effectively model the probability distribution, convey data structure, and successively refine the model. The algorithm uses mutual information to create a spanning tree which estimates the probability densities. The mimic algorithm is known to perform very well with few iterations on problems that require some sort of underlying structure.

For the first experiment, the number of iterations and the size of the bit string will be evaluated to determine the best iteration value to use for further experiments. Default hyperparameter values were used for this experiment.

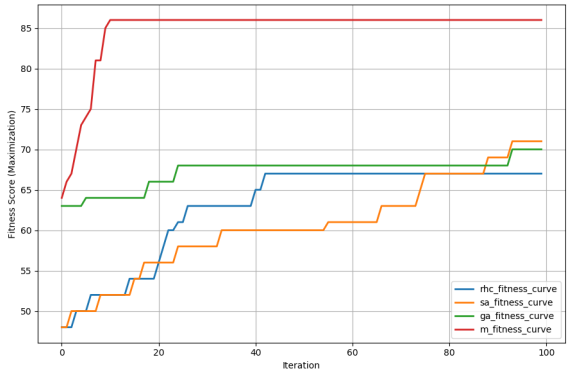
For the second experiment, a max iteration of 100 and a bit string length of 100 were selected. From the first chart is can be seen that increasing the pop_size continually improves the fitness score. Pop_size influences the number of samples generated from the probability density so increasing this number should help the algorithm handle complexity at the cost of run time. The optimal value is approximately 200 (the default value). The second chart shows the relationship between keep_pct and fitness score. It can be seen that low and high values have diminishing returns with optimal values around .2 (the default value). The keep percentage directly influences theta and it makes sense that theta needs to not move to quickly and not move too slowly.



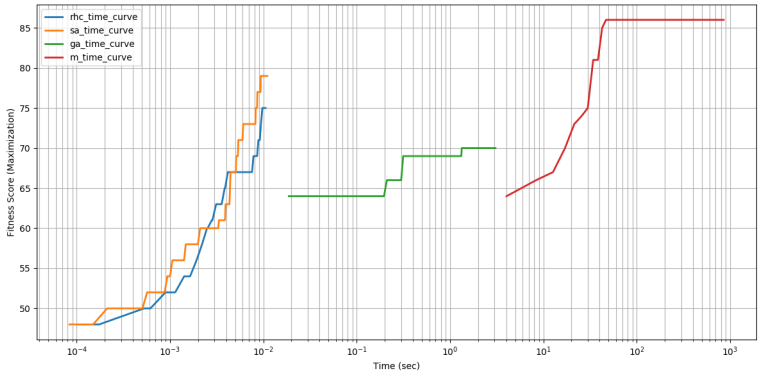


Algorithm Run Times

The final analysis for the 4 Randomized Optimization Problem looks at run times (logarithmic scale) versus fitness score for the flip flop problem. Only one of the three problem spaces are includes because the timing of the algorithms does not drastically change over the problems. From the chart, it can be seen that Mimic takes exponentially longer to run (even one iteration) but it converges in much fewer total iterations. The Randomized Hill Climbing and Simulated Annealing run very fast and Genetic Algorithms are in the middle. This is a result of the inherent complexity of each algorithm. If the fast algorithms are able to reliably converge then they may be a really good choice. In the example below, it may very well make sense to pay the cost of time with Mimic to achieve 20% better fitness.



Fitness versus Iteration (4 Peaks)



Fitness versus Wall Clock Time (4 Peaks)

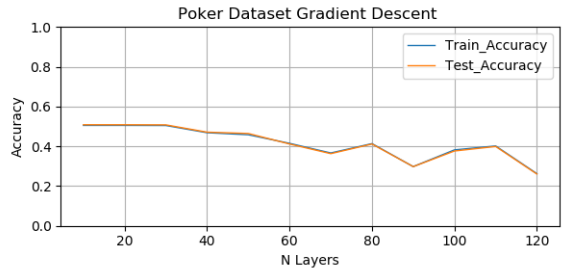
This relative timing and performance for mlrose was very similar to ABAGAIL which can be seen below.

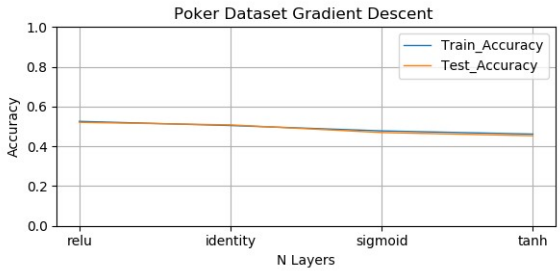
```
(base) don@don-desktop:~/Desktop/Assignment2/ABAGAIL$ java -cp ABAGAIL.jar opt.t
est.FlipFlopTest

Randomized Hill Climbing
Time: 5
Fitness Score: 43.0

Simulated Annealing
Time: 1
Fitness Score: 47.0

Standard Genetic Algorithm
Time: 187
Fitness Score: 46.0

MIMIC
Time: 2358
Fitness Score: 48.0
```

Neural Network Random Optimization

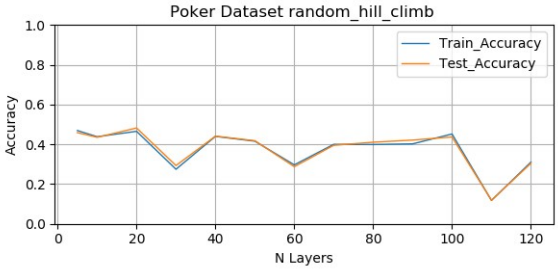
In this section, three of the Randomized Optimization Algorithms will be used to optimize the weights in a Neural Network used in the Poker Dataset Assignment 1. The Poker Dataset described in Assignment 1 classified cards between 1 and 9 based on a number of card features with many instances. I have perfect domain knowledge of this dataset; however, the formulas to convey that domain knowledge are complex and challenging to learn through machine learning algorithms. Assignment 1 used backpropagation to determine the weight on each node in the classification nueral network using the relu activation, lbfgs solver, 500 max iterations, and a layer size of 100 and 50. This assignment will be focused on deriving learning, timing, and validation curves for Randomized Hill Climbing, Simulated Annealing, and Genetic Algorithm applied to Neural Networks. It's important to note that the previous experiments solved relatively simple, bit string problem domains whereas the neural net domain requires continuous valued nodes on a challenging domain.

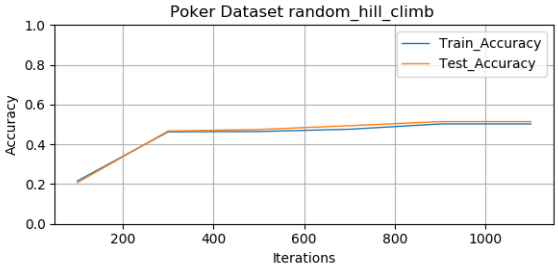
Gradient Descent: The solver for mlrose is similar to the solver for sklearn; however, they are not the same so a new, quick analysis needs to be completed to make sure we continue to get the intended results. In assignment 1 we looked primarily at layer size and activation type so we will also do that here for the poker dataset. In both experiments, the relu activation performed the best. With regard to hidden layers, the two algorithms performed quite differently highly the difference in the solver types. Layer size effects representational power and number of nodes that will be evaluated. For the purpose of this assignment, an optimal hidden layer size of 30 will be used although the volatility in the chart indicates more data exploration would be useful for future experiments.



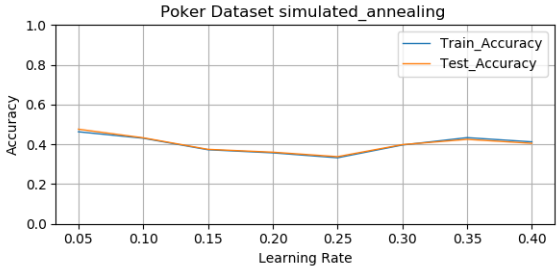


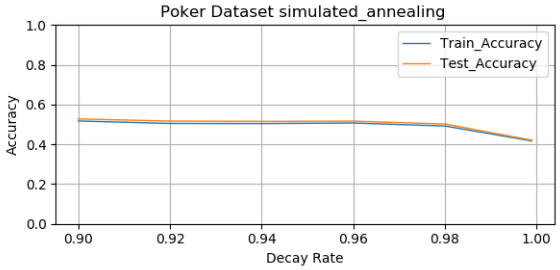
Randomized Hill Climbing: Next we will attempt to train our neural net using Randomized Hill Climbing in place of gradient descent. Gradient descent back propagates error defined by an error function in order to iterative adjust weights for data sample. Randomized Hill Climbing will instead iteratively use a neighbor function to optimize a fitness function across all nodes of the network. For this analysis we will focus on Number of Network Layers (to evaluate complexity) and Number of Iterations (to evaluate progression). We will not focus on restarts since we are performing this analysis with cross validation but this would be something to look into in the future. We are also starting with a higher value of learning rate since we are now in the continuous realm and may need to explore more than in discrete optimization. From the first chart, it can be seen that there is a lot of volatility between the various neural network layer size with the optimum at 5. The variablility is due to the low number of iterations used for this experiment (did not fully converge). For a future experiment higher iterations and more exploratory neighbor functions (through learning rate) are suggested. For the second dataset, the effect of increasing accuracy with increasing iteration can be observed. It can be seen that the optimal iteration is 1000 and a really good future experiment would be to further iterations.



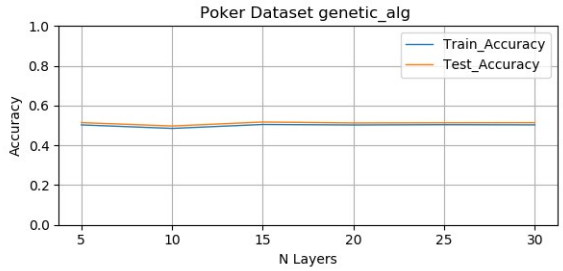


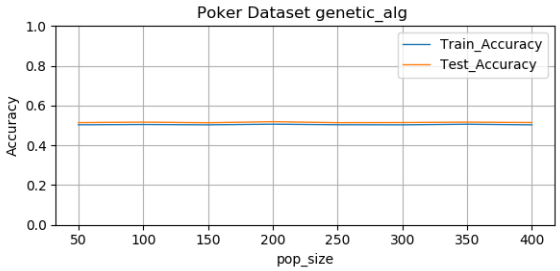
Simulated Annealing: Due to the similarity between Randomized Hill Climbing and Simulated Annealing, we will use the optimal value of 5 hidden layers and 1000 iterations. For this analysis, the effect of learning rate and decay rate will be evaluated. In the continuous optimization problem (versus discrete), learning rate is used as the step value which is how much the value of the selected neighbor from the neighbor function is changed each round. A large value influence more exploration more quickly. Decay rate is the rate the temperature changes as described earlier. In the first chart, the effect of learning rate on accuracy can be observed. The optimum values are approximately .05 and .30 so we will choose .30 to be more aggressive with exploration. The second chart used this .30 value and evaluated the effect of decay rate on accuracy. It can be seen that lower decay rates perform better meaning the algorithm performs better when temperature is moving a little faster (but not too fast). We chose a low value for the hidden layer making the problem space easier (lower number of nodes and weights) so it is not surprising that we don't need as much exploration. Larger hidden layers values may require a slower temperature decline.





Genetic Algorithm: The genetic algorithm tackles the weight optimization problem by creating populations (pop_size parameter) of node weights, mutating the populations through the mutation probability parameter, combining those populations through random crossover, and iterating on the populations through generations. For the purpose of this experiment, we will focus on the number of hidden layers which will effect the total number of nodes and complexity in the genetic algorithm and the population size used to solve the weights on each nodes. These two parameters are conditional on each other because as the number of nodes grows the needed population size will also likely grow. We did not focus on mutation probability and instead used the optimal value from the previous experiments. Mutation probability will effect how quickly we explore the weights and we set this value to a middle of the road value. Both charts below indicate that the number of hidden layers and the population size have minimal impact on performance. Since the algorithm has already met the highest values of the previous algorithms I would attribute this to the algorithm converging quickly to the optimal value with few iterations. A future experiment would look at the effect of iterations on performance and see if fewer iterations reduce performance due to lack of convergence. As with the other charts, the training and test curves have very little variance (curves are close to each other) and low bias (training error is rather high). The low variance indicates we are likely not overfitting the training data and the high bias means that we have room for improvement with the training error. Based on domain knowledge and previous analysis, what we need to be able to better apply our domain knowledge through an algorithm. The next assignment will take a further look into this.





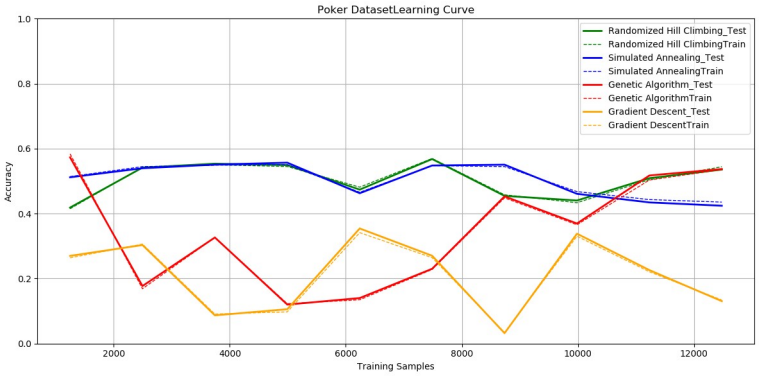
Conclusions and Comparisons:

For the final comparison, the optimized hyperparameters will be used to look at an aggregate of learning curves on a single chart. The optimized hyperparameters are as follows:

- Randomized Hill Climbing: `hidden_nodes=[5],activation='identity',max_iters=1000, learning_rate=0.2`
- Simulated Annealing: `hidden_nodes=[5],activation='identity',max_iters=1000, learning_rate=0.3,schedule=mlrose.GeomDecay(init_temp=20, decay=.9, min_temp=.0001)`

- Genetic Algorithms: `hidden_nodes=[20], activation='identity',max_iters=50, pop_size=100, mutation_prob=0.1,`
- Gradient Descent: `hidden_nodes=[30], activation='relu',max_iters=1000,`

These optimal valued were used to generate a learning curve to evaluate the effect of training samples on performance. In this analysis, each learning will iterate over successively larger batches of training data ranging from 10% to 100% of the data. The training and testing (validation) curves will be plotted using 5 fold validation from the sklearn learning_curve function. It's important to note here that the 5 fold validation performed here is much different than the sampling used in all previous examples. The previous sampling used 5 iterations of sklearn's function train_test_split with an iterative random seed. From the chart, it can be seen that more training data does improve accuracy; however, our chart has a tremendous amount of volatility between the training samples. This indicates to me that we do not have enough iterations or that we are possibly getting stuck at local maximum when we inject additional randomness through choosing different sizes of data. Even with the volatility, the overall trend of traversing from low accuracy to high accuracy with additional training data can be seen.



The final analysis shows the timing of each algorithm versus the sklearn log loss function which is used to calculate a fitness score in the neural net. The algorithms attempt to maximize this number. Please note the genetic algorithm got cut off early and that the x axis is logarithmic and did converge. From the chart, it can be seen that gradient descent converges very quickly to an optimal value while the genetic algorithm converges very slowly (consistent with previous analysis). For the poker problem space, all algorithms do eventually converge, but at a time cost. None of the algorithm were able to perform better than 60% accuracy, which I attribute to the problem space itself being difficult to solve with only a small (.1%) number of possible poker hands given.