Project 2: Randomized Optimization

Scott Merrill
smerrill7@gatech.edu

I. Introduction

Optimization algorithms seek to find a set of inputs that maximize or minimize the value of a given objective function. These techniques are particularly useful when objective functions are complex, multidimension, or lack a closed form solution. Randomized optimization methods make repeated random steps to incrementally improve a solution. Unlike gradient-based optimization methods, randomized optimization techniques can be applied to discontinuous, non-differentiable objective functions and thus are applicable to many more domains. This paper explores several toy examples of discontinuous objective functions to compare the performance of popular randomized optimization algorithms. In particular, we look at Random Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithms (GA) and Mutual-Information-Maximizing Input Clustering (MIMIC) to identify the types of problems in which each of these algorithms excel. Further, the hyperparameters of each algorithm will be altered to identify their effects on objective function optimization and run time. Finally, we will apply RHC, SA and GA to a continuous objective function—particularly the loss function of the Neural Network (NN) used in Assignment 1—and compare their performance to gradient decent. We will then comment on when randomized optimization algorithms may be preferred to gradient-based methods in continuous domains.

II. Knapsack Problem

We first consider a problem with complex interdependencies between variables. Given a set of item values and corresponding weights, the goal of the Knapsack problem is to identify the combination of items that maximizes value subject to a limit on the weight of the knapsack. The fitness function for this problem is expressed in (1).

$$Knapsack(x) = \begin{cases} \sum_{i=0}^{n-1} v_i x_i, & if \sum_{i=0}^{n-1} w_i x_i, \leq W \\ 0, & if \sum_{i=0}^{n-1} w_i x_i, > W \end{cases} \qquad (1)$$

A vector of 40 weights and corresponding values was randomly generated, and the total weight of the knapsack was limited to 75% of the sum of the weight vector. Finally, the maximum number of any particular item that can be included in the knapsack is 3. RHC, SA, GA and MIMIC were used to identify the allocation of items that maximizes the value of the knapsack.

A. Random Hill Climbing

RHC with 10 random restarts and 500 iterations per restart was first used to optimize the fitness function. Figure 1 shows, the fitness scores from each hill climbing episode. As can be seen, the fitness scores are quick to converge in each restart. Since RHC attempts to identify a better solution by evaluating the fitness scores of neighboring states, these plateaus indicate a local maximum was found with respect to the neighbors set. Further since, the algorithm stabilizes to a different value on each restart, it is clear that the problem has many local optima.

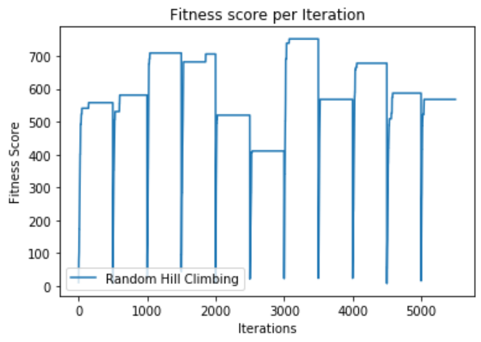

Figure 1

Figure 2 plots the fitness scores with respect to iteration for the 7th restart of the algorithm which resulted in the most valuable knapsack of 752. We note that the resulting graph is monotonic, highlighting the fact that RHC only accepts answers that improve the current best solution.

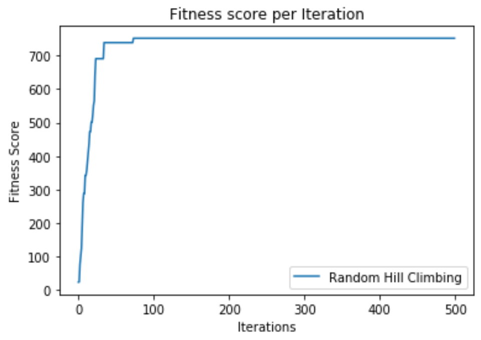

Figure 2

B. Simulated Annealing

Unlike RHC, SA can accept a candidate solution that is worse than the current best solution. While an improvement over the current solution is always exploited, SA accepts inferior answers according to the probability function defined in (2) where x_t is the state being evaluated, x is the current state and T is the temperature. From the equation it's important to note that with larger temperatures the algorithm is encouraged to explore the search space and take suboptimal steps while temperatures closer to 0 encourage the algorithm to only take steps that improve the fitness function. The hope is that slowly decaying the temperature will allow the algorithm to take steps away from a local maximum and eventually settle on the global optimum (or at least a better local optimum).

$$P(x, x_t, T) = \begin{cases} 1, & if\ f(x_t) \geq f(x) \\ e^{\frac{f(x_t)-f(x)}{T}}, & otherwise \end{cases} \qquad (2)$$

Moreover, SA is highly sensitive to the value and decay schedule of the temperature parameter. Grid search was used to identify the optimal initial temperature, geometric decay speed and minimum temperature for the Knapsack problem and a plot of the fitness scores achieved by varying these constraints is shown in Figure 2. The initial temperature, decay speed and minimum temperature that resulted in the best fitness score were 1.0, 0.9 and 0.001 respectively. The best decay speed of 0.9 is rather quick and indicates that the temperature will reach its minimum value after only 66 iterations ($0.9^{66} < 0.001$). This low optimum decay speed indicates that exploring the search space isn't very beneficial. This is likely due to existence of many local optima and their proximity with each other. For example, given a local optimum, a second local optimum can be found by replacing a knapsack item with value v, with another item of equal weight but value 2v. In this example, the two knapsacks would be nearly identical (differing only by two items) and their values would be different by v. Such demonstrates the close proximity of many local optima. And, because of this relative closeness, randomly exploring may inadvertently result in the algorithm getting stuck in an inferior local optimum. Moreover, a quick decay schedule makes sense as it will cause SA to act more greedily and take fewer potentially detrimental exploration steps.

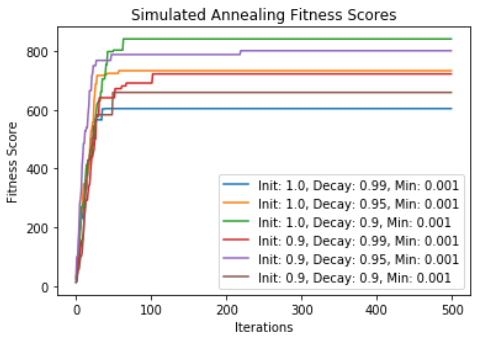

Figure 3

C. Genetic Algorithms

GA were next used to value the knapsack; both the population size and mutation probability were specifically tuned for the problem. The population size parameter defines the number of candidate solutions the algorithm considers at each iteration for mutation and cross over. With larger populations, more candidate solutions are generated, resulting in more fitness function evaluations and longer runtimes. Increased population size, however, often results in improved fitness scores as with more candidate solutions, the probability of a favorable cross-over or mutation event is larger. The mutation probability parameter adjusts the probability by which a random element in the state vector is changed. Larger mutation probabilities should enable more exploration at the expense of potentially degrading the natural improvement of the population. Figure 4 shows the fitness scores achieved by GA with varying population sizes and mutation probabilities.

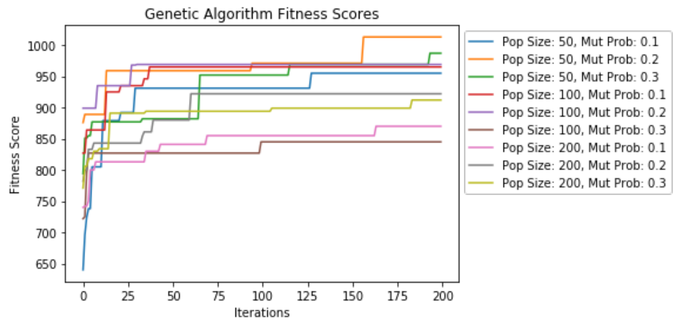

Figure 4

The best population size and mutation probability were found using grid search and determined to be 50 and 20% respectively. A surprising note from Figure 4 is that smaller populations appear to outperform larger ones. While, larger population sizes enable for more candidate solutions, better performance of larger populations isn't guaranteed. If smaller populations are more diverse, for example they can outperform. In our experiment however, population members were initialized randomly and thus the anomaly in Figure 4 is likely due to chance; either the smaller populations were randomly more diverse or had more favorable mutations than larger populations.

D. MIMIC

Grid search was used to optimize the population size and keep percentage of the MIMIC algorithm on the Knapsack problem. It was found that a population of 100 and a keep percentage of 30% optimized the algorithms performance. Figure 5 shows the fitness scores from MIMIC using several different pairs of these parameters. From the figure, larger population sizes and keep percentages result in larger fitness scores. This makes sense as larger values for these parameters should result in more accurate estimations of $P(x)^{\theta_t}$. Further in problems with complex structures where elements in the state vector are all interrelated, estimating this distribution correctly is extremely valuable.

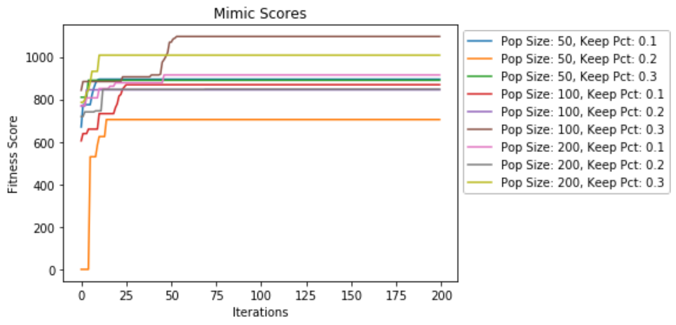

Figure 5

E. Algorithm Comparison

Figure 6 shows a comparison of all the optimized algorithms in terms of fitness score per iteration and Figure 7 shows a comparison of each algorithm in terms of fitness score per function evaluation. Evaluating each algorithms performance on iterations alone is clearly biased as MIIMC and GA can make hundreds of function evaluations in a single iteration. Moreover, comparing the number of function evaluations provides a better picture of resource utilization.

Overall, the optimized GA produces the largest fitness score at any given iteration and function evaluation. This likely occurs because the Knapsack problem can be broken down into several smaller optimization problems. Maximizing the number of high value to weight ratios, for example is one subspace where initial optimization is important. GA enables for the optimization of independent subspaces and thus is well suited for this problem.

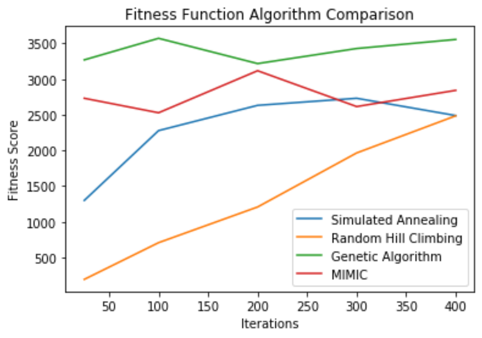

Figure 6

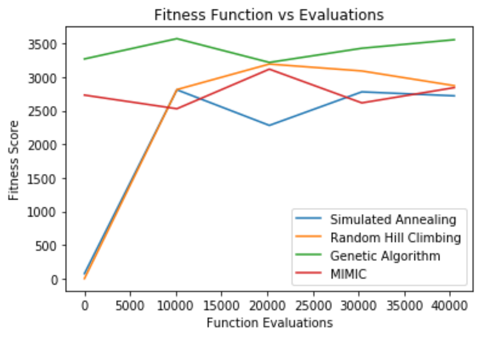

Figure 7

The run time for each algorithm is shown in Figure 8. As can be seen, MIMIC and GA take considerably longer than both RHC and SA which is expected since both these algorithms are making hundreds of function evaluation calls within each iteration. In addition, MIMIC is sampling from the distribution $P(x)^{\theta_t}$ at each iteration thus requiring additional time and resources.

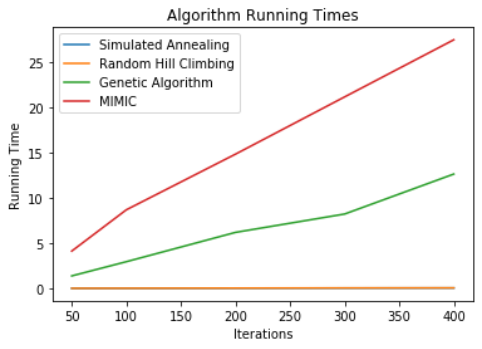

Figure 8

The performance of each algorithm was next evaluated with respect to varying problem sizes. All algorithms were given 2,000 function evaluations to solve the problem and the best fitness score for each algorithm is plotted with respect to problem size in Figure 9. As can be seen from the chart GA outperforms the other algorithms irrespective of problem size. Interestingly, however, as problem size increases, MIMIC's performance shows relative improvement over RHC and SA. This is likely due to the increased structural dependencies between state vector inputs as problem size is increased. MIMIC is able to capture some of this structure, whereas RHC and SA are not; these algorithms are still randomly modifying the state vector which becomes increasingly difficult as the size of the state space increases.

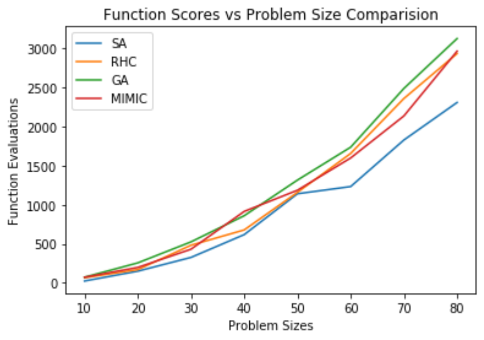

Figure 9

III. One Max Problem

We next consider a much simpler model with no interdependencies between state vector elements. The One Max fitness function returns a score equal to the sum of all the elements in the state vector and is defined in (3).

$$OneMax(x) = \sum_{i=0}^{n-1} x_i \tag{3}$$

RHC, SA, GA and MIMIC were each used to optimize a state vector of size 200 where each element in the vector can take values from 0 to 10. RHC and SA were given 1,000 iterations to solve the problem, while GA and MIMIC were given 100 iterations.

A. Random Hill Climbing

RHC with 10 random restarts and 100 iterations per restart found a state vector with a fitness score of 472. The fitness function for all 10 episodes and the best episode are shown in Figures 10 and 11 respectively. Unlike in the Knapsack problem, none of the restarts reaches a plateau; this is also apparent in Figure 11 which shows the fitness score increasing monotonically. This behavior is due to the existence of a single local optimum in the One Max problem. Moreover, the global maximum or a better solution can always be found by evaluating every neighboring state vector. Thus, with infinite iterations, RHC is guaranteed to converge to the global maximum in this problem.

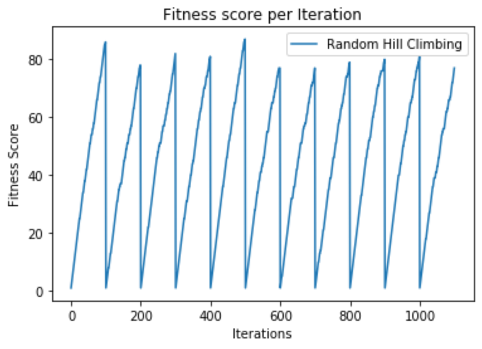

Figure 10

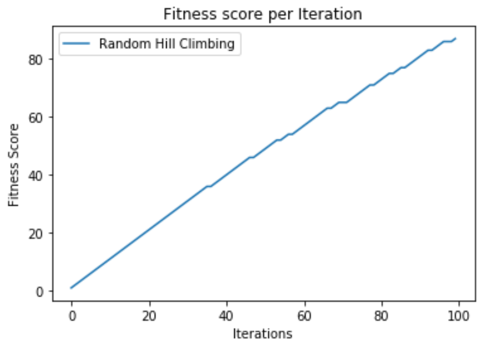

Figure 11

B. *Simulated Annealing*

A SA algorithm was tuned using grid search to identify the optimal initial temperature, decay rate and minimum temperature for the One Max problem. These values were determined to be 0.5, 0.5 and 0.001 respectively. Figure 12 compares the fitness scores achieved using several different initializations, decays and minimum thresholds for the temperature parameter. Given there's no local maxima, there is little value in exploring suboptimal states as such will only lead to taking more steps away from the global maximum and thus slower convergence. Always exploiting a solution that is an improvement over the current solution should lead to guaranteed convergence in the least amount of time. Moreover, lower initial temperature values and faster decay speeds should outperform, which is consistent with Figure 12.

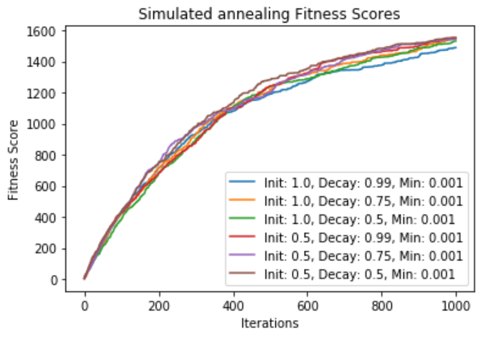

Figure 12

C. *Genetic Algorithms*

The optimal population size and mutation probabilities for a GA on the given One Max fitness function were found to be 100 and 10% respectively. Figure 13 shows the performance of several population sizes and mutation probability pairs. An interesting note is that smaller mutation probabilities appeared to outperform larger ones. This indicates that exploration isn't extremely valuable and that the global optimum may already be found using crossover within the existing population.

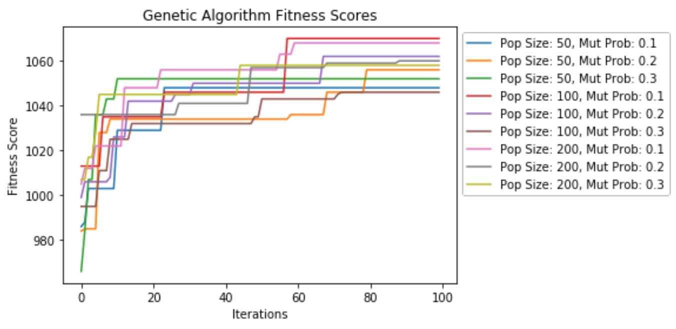

Figure 13

D. *MIMIC*

A Mimic algorithm was next carefully parameterized to maximize the One Max function. The optimal population size and keep percentage were found to be 200 and 30% respectively. Figure 14 shows the performance of MIMIC with respect to varying population sizes and keep percentages and indicates that larger population sizes and keep percentages outperform smaller ones. While larger values for these parameters certainly enable more accurate probability distribution calculations of $P(x)^{\theta}$, they also require more function evaluations, computational resources and time.

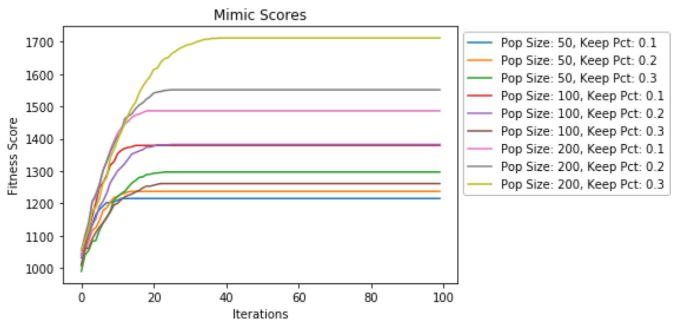

Figure 14

E. *Algorithm Comparison*

A comparison of the fitness scores achieved at each iteration for each algorithm is shown in Figure 15. As can be seen, MIMIC appears to outperform, however, this is simply due to the fact that MIMIC is making many function evaluations within each iteration.

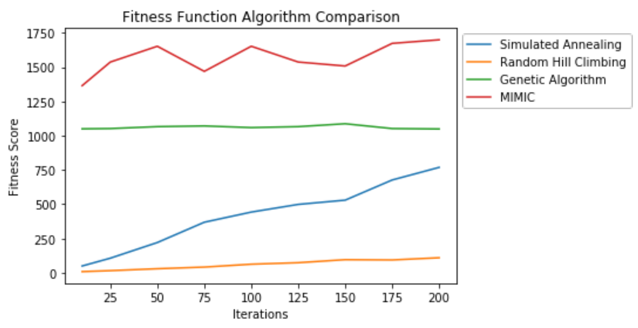

Figure 15

Figure 16 shows a comparison of the algorithms when compared in terms of fitness scores per function evaluation. This chart creates a fairer comparison amongst the algorithms as it adjusts for the computational resources used by each algorithm. When comparing algorithms based on resource utilization, SA performs the best. Since there is little structure in the problem and each state vector element needs to be optimized independently, using an algorithm like MIMIC to model structure and keep track of probability distributions is a wasteful use of time (as shown in Figure (17)) and resources. In contrast, greedily exploiting favorable neighboring states works well in problems with a single global maximum (and no local maxima) and is guaranteed to converge with infinite iterations. Thus, SA with a quick temperature decay schedule is appropriate for this problem.

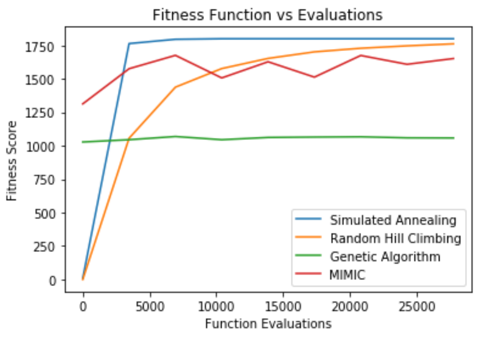

Figure 16

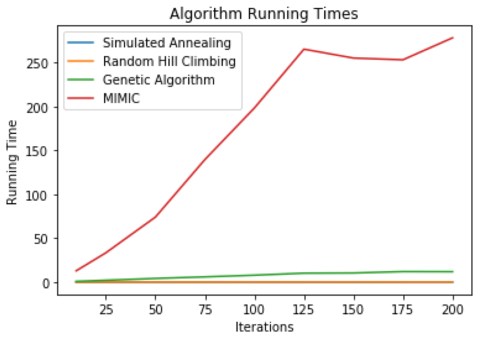

Figure 17

We finally compare the performance of each algorithm in optimizing the One Max function with respect to the size of the state vector. State vectors of size 100, 250, 500, 750 and 1,000 were tested. Each algorithm was allowed 20,000 function evaluations and the max fitness score achieved with respect to problem size is shown in Figure 18. As can be seen, SA appears to show more dominant performance with larger problem sizes. In addition, we see that with larger problems, MIMIC's performance degrades. This makes sense as with larger problem sizes the algorithm wastes more effort trying to model structure.

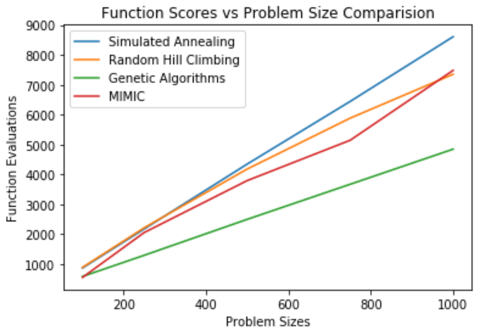

Figure 18

IV. Flip Flop Problem

The Flip Flop fitness function is more complex than One Max, however, has less structural dependencies than the Knapsack problem. The Flip Flop fitness function returns the count of consecutive pairs $\langle x_i, x_{i+1}\rangle$ in the state vector that are different. There is thus a chain dependency structure in the problem whereby the best value for any element in the bit string is dependent only on its immediate predecessor. RHC, SA, GA and MIMIC were each used to model this dependency structure on a problem of size 1,000.

A. Random Hill Climbing

Figure 19 shows the fitness scores for RHC climbing with 10 random restarts and 2,000 iterations per restart. The optimal value was found on the 2nd restart which produced a fitness score of 827; the fitness score of each iteration for this restart is shown in Figure 20. In Figure 19 we note that none of the restarts completely reach a plateau and converge to a local optimum. This indicates more iterations in each restart would likely lead to better fitness scores. This relationship can be seen more clearly in Figure 20; with more iterations the algorithm appears to be increasing slowly and has yet to completely level off. Another note is that all restarts started and ended at roughly the same fitness score, which potentially indicates the existence of fewer local optima.

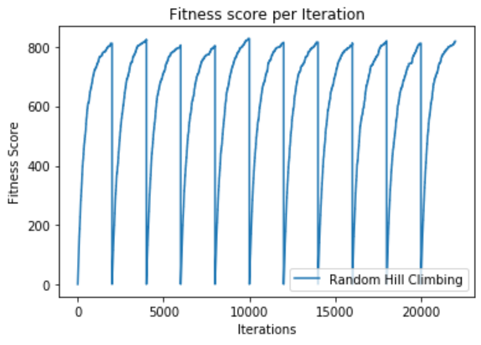

Figure 19

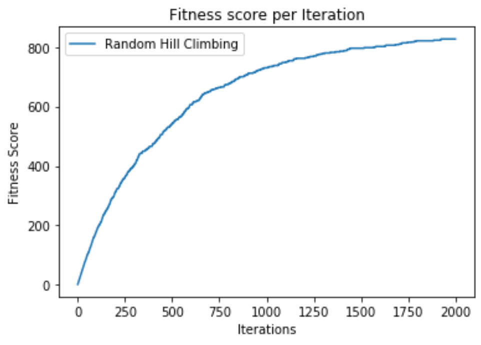

Figure 20

B. Simulated Annealing

SA was next used to maximize the Flip Flop function. Grid search was used to identify the optimal initial temperature, decay rate and minimum temperature of the algorithm. These values were determined to be 0.9, 0.999 and 0.001 respectively. Figure 21 compares the fitness scores achieved using several initializations, decays and minimum thresholds for the temperature parameter. From the chart, it appears that higher decay rates outperform lower ones. This makes sense as keeping the temperature elevated will enable for more exploration. And in large search spaces, with few local optima, exploration is unlikely to run the risk of the algorithm getting stuck in an inferior local optimum as we saw in the Knapsack problem.

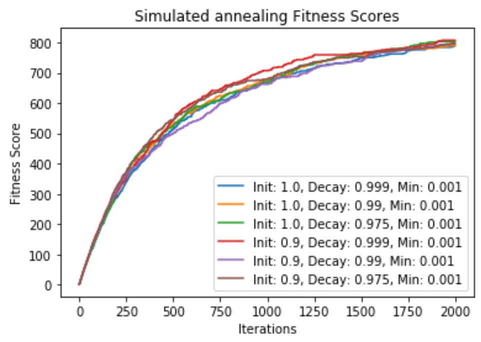

Figure 21

C. *Genetic Algorithms*

Grid search was used to find the best population and mutation probability for a GA to maximize the objective function. These values were found to be 200 and 20% respectively. Figure 22 plots several combinations of population size and mutation probability parameters. From the chart larger population sizes and mutation probabilities outperform lower ones. While larger populations should be expected to outperform, the superior performance of larger mutation probabilities is not a given. Considering the implemented GA used one-point crossover, these results make sense as it's unlikely the case that the bit string selected from each parent is optimal (or even close to optimal). For example, if the splitting point is at the 500th element in the state vector, the probability that the first 500 bits from parent one is alternating is essentially 0 (0.5^{500}). Thus, the need for exploration and mutation is very reasonable in the scope of this problem.

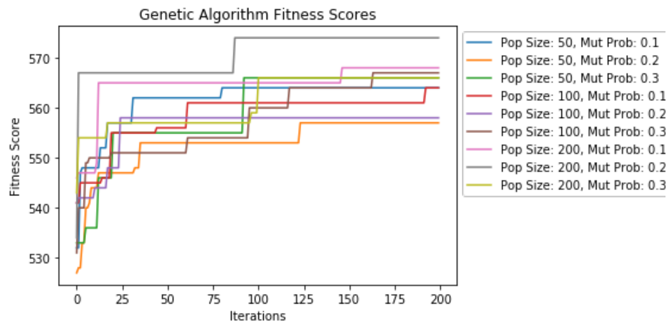

Figure 22

D. *MIMIC*

We finally optimize the Flip Flop problem using MIMC. The best population size and keep percentage were found using grid search and determined to be 200 and 30% respectively. Figure 23 shows the performance of several different population sizes and keep percentages. As can be seen, larger populations and keep percentages appear more appropriate than smaller ones, which is likely due to the fact that larger values for these parameters will enable the algorithm to better model the problem's chain dependency structure.

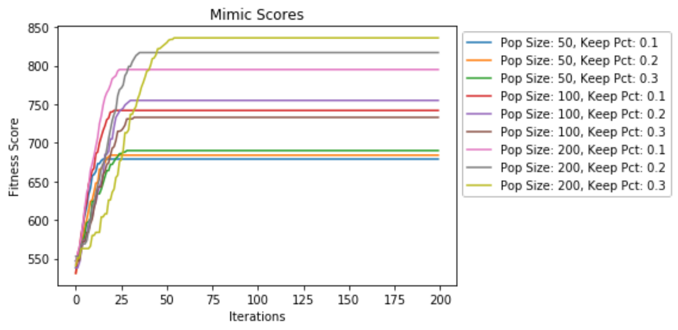

Figure 23

E. *Algorithm Comparison*

Figure 24 shows a comparison of the performance of all the models in terms of fitness score per iteration. As can be seen, MIMIC appears to outperform suggesting the algorithm is able to appropriately model the problem's structure thus allowing for more efficient optimization. Creating this dependency structure, however, requires more computational resources than the other algorhitms. As seen in Figure 25, MIMC takes considerably more time than RHC, SA and GA. This is due to the fact that MIMIC requires many function evaluations and the sampling from a candidate distribution with each iteration.

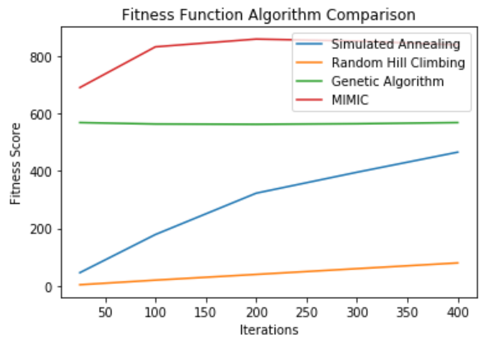

Figure 24

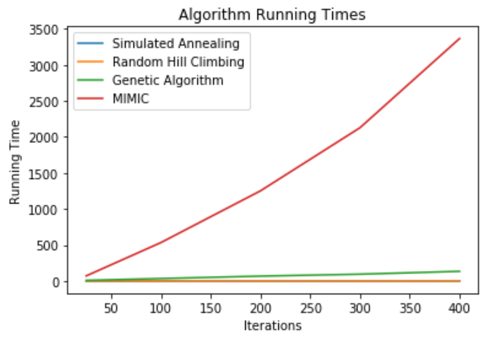

Figure 25

To compare the algorithms on a level playing field, Figure 26 shows the fitness scores achieved by each algorithm in terms of function evaluations. MIMIC still outperforms the remaining algorithms but RHC and SA achieve significant improvements with these additional function evaluations.

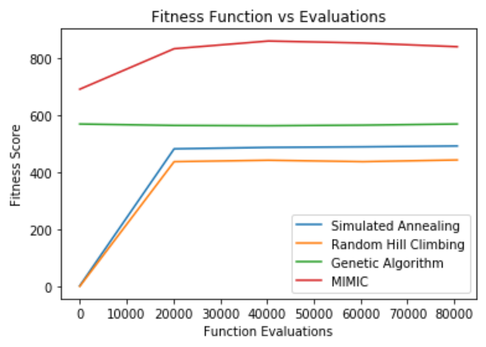

Figure 26

The performance of each model was finally compared to varying problem sizes. Five problem sizes ranging from 100 to 1,000 are considered. RHC and SA were given 2,000 iterations to converge while MIMIC and GA were given 200. Figure 27 shows that both MIMC and GA achieve larger outperformance over RHC and SA with larger problem sizes. This indicates that the guess and check strategies of RHC and SA are less dominant and reliable as the state vector size increases. With extremely large problem sizes, these strategies may be infeasible altogether. Moreover, with larger state spaces, modeling structure is more valuable and results in more efficient optimization.

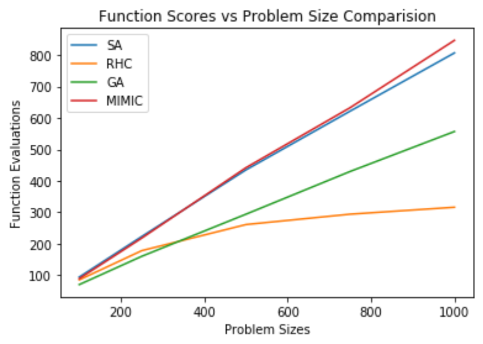

Figure 27

V. NEURAL NETWORK WEIGHT OPTIMIZATION

RHC, SA and GA were next used in a continuous domain to optimize the weights of the Neural Network (NN) used in Assignment 1. The NN in Assignment 1 was used to predict NFL game lines with a target accuracy of 52.4%; since Vegas spread bets pay -110 (meaning a $110 bet wins $100), an average prediction accuracy of 52.4% is necessary to break even. The raw dataset contains NFL scores, game lines and weather conditions but additional feature engineering and preprocessing were done to extract additional explanatory variables. In total, 34 continuous features were fed into the NN and in Assignment 1 it was determined that the optimum NN architecture consisted of a single hidden layer of size 3 and used a sigmoid activation function. Moreover, we will use this NN architecture when tuning the weights using RHC, SA and GA. We will finally compare each to the performance of gradient decent.

A. Random Hill Climbing

RHC with 10 random restarts was given 2,000 iterations to minimize the loss function on the NN. Random Search with 100 attempts was used to find the optimal step size for the algorithm. The step size parameter determines the amount by which weights will be changed on each iteration. All step sizes from 0 to 0.2 were considered. It was found that a step size of 0.113 minimized in-sample loss. The loss curve produced by RHC with this step size is shown in Figure 28. The curve shows favorable properties of gradual decline and appears to be converging.

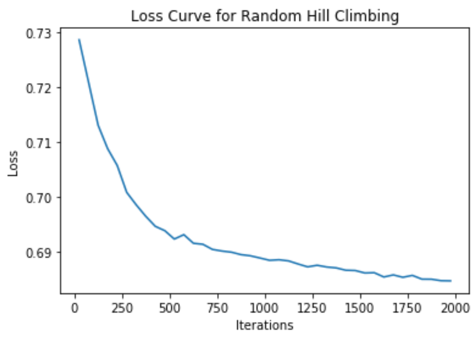

Figure 28

Predictions were next made out-of-sample with the optimized classifier. Figure 29 plots the training and testing accuracy curves for the trained NN; since the dataset is balanced, the accuracy measure used is simply the percentage of correct classifications. Both training and testing accuracies increase monotonically up to 1,000 iterations, however after 1,000 iterations, the two series deviate. This divergence between train and test accuracies indicates overfitting. Early stopping after 1,000 iterations may therefore be appropriate to reduce overfitting and improve generalization ability.

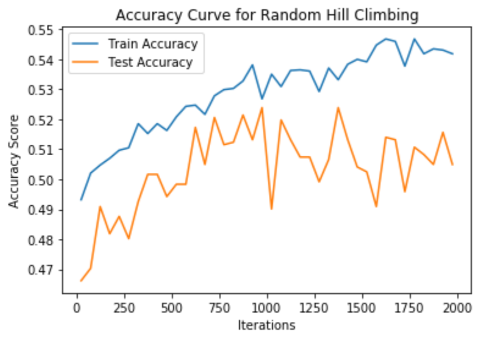

Figure 29

The confusion matrix for the out-of-sample predictions is shown in Figure 30. The optimized NN achieved an overall accuracy of 50.4% when trained using RHC. The classifier predicts a team will cover the spread around 60% of the time and as a result produces many false positives. These false positives, however, aren't a big issue in the scope of the problem; when betting Vegas lines, we are more concerned about overall accuracy and are indifferent to both false positives and false negatives.

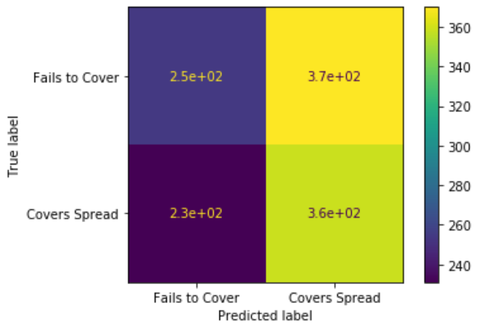

Figure 30

B. Simulated Annealing

SA was next used to minimize the loss of our NN and was given 2,000 iterations. Random search was used to identify the optimum starting temperature, decay rate, minimum temperature and step size which were found to be 0.899, 0.902, 0.007 and 0.145 respectively. The loss curve for the optimized NN with these parameters is shown in Figure 31. While the curve exhibits a general downward trend, there are several periods where the loss spikes in value. These increases in loss are expected as with high temperatures the algorithm will act more like a random walk and potentially take steps away from the local minimum in attempt to find a better overall solution. With more iterations (over 1,300) the temperature cools and we see a more monotonic decrease in the loss curve. SA, however, doesn't appear to have converged to an optimum set of weights even after 2,000 iterations.

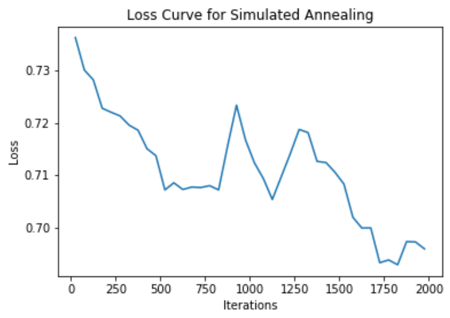

Figure 31

The NN optimized using SA was next tested in and out-of-sample. Figure 32 shows the training and testing accuracies of the optimized classifier. While training accuracy sees modest improvement with more iterations, test accuracy improves significantly. The test accuracy finishes around 54% which is much higher than the test accuracy observed form RHC. The training accuracy, however, is lower than that of RHC, which may be another indication of RHC overfitting. Overall, with less overfitting than RHC, the NN with weights trained through SA appears to have better generalization ability.

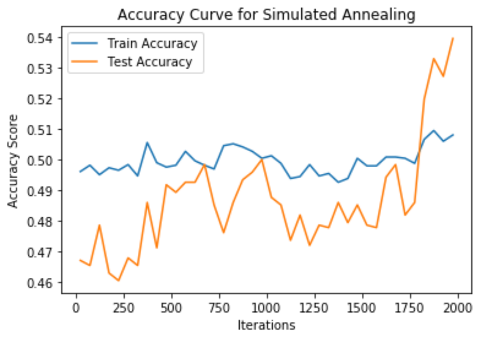

Figure 32

The confusion matrix for the predictions made by the classifier are shown in Figure 33. As can be seen the algorithm predicts a team will fail to cover the spread around 2 times more frequently than predicting a team will cover. However, in both cases, the algorithm is quite efficient. When predicting a team will fail to cover it is correct 54.3% of the time and when it predicts a team will cover it is correct 53.7% of the time. Both of these accuracies exceed the target accuracy of 52.4% which is necessary to break-even on Vegas spread bets. More test data however is still necessary to ensure these results aren't spurious and due to chance.

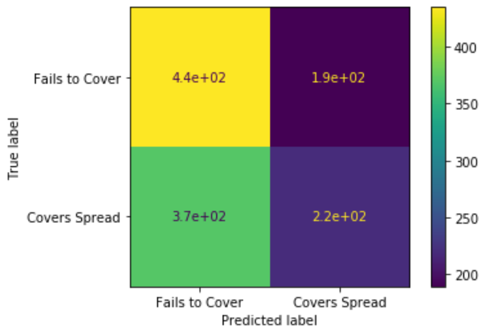

Figure 333

C. Genetic Algorithms

A GA was given 200 iterations to converge and was used to tune the weights of the NN. 100 attempts of random search were used, and it was found that the population size, mutation probability and step size that simultaneously minimize loss were 66, 7.66% and 0.0193 respectively. Figure 34 shows the loss curve of a NN with weights trained using these parameters. While the loss curve exhibits a general downward trend, it's also discontinuous and resembles a step function. The flat regions of the loss curve indicate periods in which cross-over and mutation were unable to improve on the current best solution. For example, from 250 iterations to 750 iterations, the same optimal solution for the algorithm was maintained and thus the loss remained unchanged. After 2,000 iterations, the loss remains orders of magnitude higher than the previous algorithms suggesting either slow convergence of the algorithm or that GA is converging on an inferior local minimum.

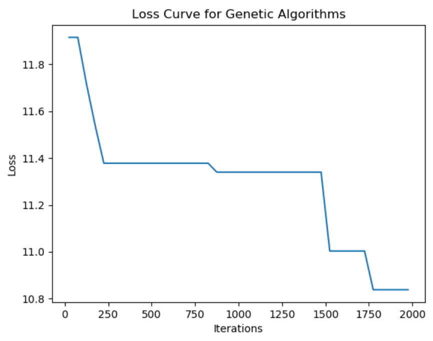

Figure 34

The NN optimized with GA, was tested out-of-sample to evaluate its ability to generalize. Figure 35 shows the train and test accuracies produced by the NN on different iterations. The test accuracy of the algorithm dominates the train accuracy at any given iteration, which indicates underfitting. However, even with underfitting, the NN as able to generalize with accuracy comparable to RHC. This suggests there's likely a lot of noise in the dataset; thus, overfitting to the training set will result in overfitting to the noise and a model with poor generalization ability.

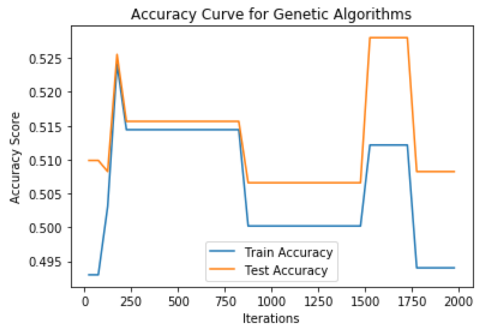

Figure 35

Figure 36 shows the confusion matrix of the predictions made by the classifier. The final model achieved an overall accuracy of 50.8%. Additionally, the algorithm produced many false positives and when predicating a team will cover a spread, it was wrong with these predictions more than half the time. When predicting a team will fail to cover however, the model does much better, producing an accuracy of 52.8%.

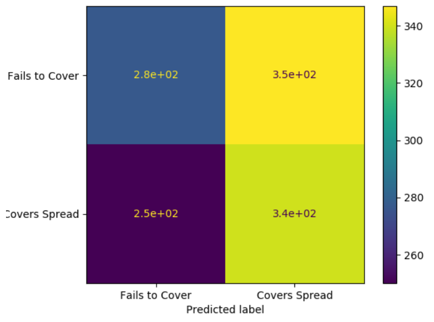

Figure 36

D. Gradient Descent

We finally analyze the performance of a NN with weights trained using gradient decent. Using random search, the learning rate that optimizes in-sample training accuracy was found to be 0.0047. The loss curve for training of our NN using this learning rate is shown in Figure 37. As can be seen, the loss curve converges smoothly in around 1,100 iterations, much quicker than all of the previous algorithms. Additionally, the loss curve contains fewer kinks and is able to efficiently identify a local minimum.

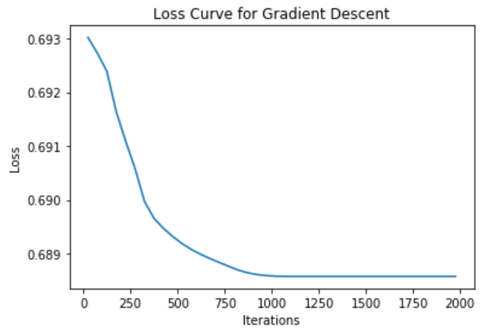

Figure 37

The NN was next tested out-of-sample. Figure 38 shows the training and testing accuracies at each iteration. Both training and testing accuracy appear to increase up until 200 iterations. At this point, the algorithm begins to overfit as seen by the increasing training accuracy and decreasing testing accuracy. This overfitting may be curtailed by early stopping. Another important note in Figure 38 is that both training and testing curves plateau after 1,100 iterations; this indication that the algorithm converged to a local minimum and thus, the weights of the NN don't change as iterations are increased beyond 1,100.

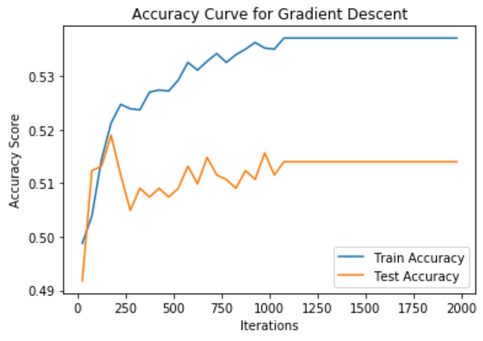

Figure 38

The confusion matrix for gradient descent is shown in Figure 39. The model performs reasonably and achieves an accuracy over 51%. In addition, the model appears to produce more accurate results when predicting a team won't cover; it is correct 54.1% of the time when making these predictions. Since only 52.4% accuracy is necessary to break even, one potential strategy may be to only bet when the model predicts a team won't cover. More data, however, would still be useful to confirm the relative efficiency the algorithm has in predicting true negatives.

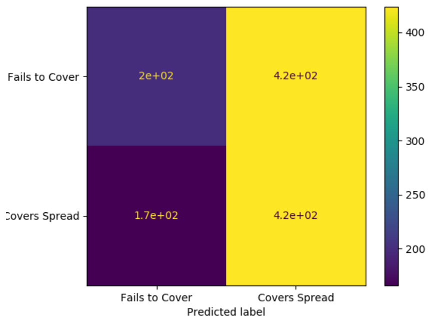

Figure 39

E. Algorithm Comparison

Figure 40 shows a comparison of the loss curves produced from RHC, SA and gradient descent. The loss curve for GA wasn't included in this plot as it was orders of magnitude higher than the other algorithms. From the loss curves we note that after 2,000 iterations, all algorithms had a similar loss value of around 0.70. RHC and SA, however, never completely converge and have larger loss values in early iterations.

Gradient descent produces the smoothest most consistent loss curve. This makes sense since gradient descent is always adjusting the weights in the direction of steepest descent; that is, the algorithm simultaneously adjusts all weights such that the reduction in the cost function is maximized. In contrast, RHC and SA rely on randomness to adjust the weights. RHC randomly makes adjustments and greedily accepts them if they reduce loss and SA similarly makes random adjustments but accepts candidate solutions less greedily. Further, while gradient descent uses backpropagation to simultaneously tune all weights, RHC and SA consider modifying only one weight with each iteration. Moreover, by calculating gradients and simultaneously modifying all weights, gradient descent achieves a quicker and more reliable convergence path to a local minimum.

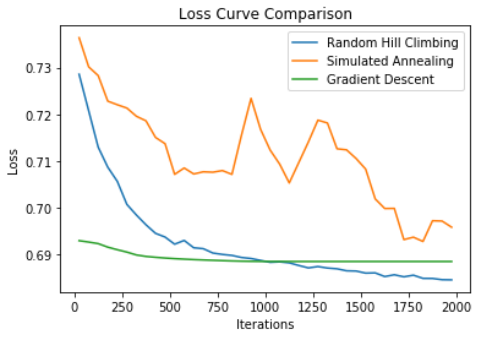

Figure 40

The training time for each iteration is shown in Figure 41; GA again was excluded from the comparison as the algorithm produced train times significantly larger than the other algorithms. While RHC appears to be much slower than SA and gradient descent, this is because 10 restarts with 2,000 iterations per restart were used. Moreover, the algorithm completed 20,000 iterations in total, while SA and gradient descent only finished 2,000 iterations; this explains why RHC takes roughly 10 times longer than SA. After adjusting for the number of iterations, RHC, SA and gradient descent all complete in a similar amount of time. Gradient descent takes a bit longer than the other two algorithms due to the calculation of the gradient at each iteration. While this speedup is marginal, our NN only contains a single hidden layer of size 3. Increasing the size of the network would likely increase gradient descent's train time as more weights would need to be tuned in each iteration and thus partial derivative calculations.

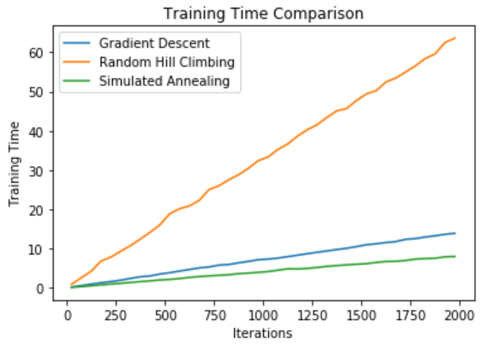

Figure 41

While gradient descent is efficient in finding a local minimum and does so in a reasonable amount of time, test accuracy also needs to be considered. Figure 42 shows a comparison of test accuracy for each algorithm. As can be seen, SA performs the best achieving an accuracy score of 54%. This likely occurs due to the fact that the other algorithms are overfitting. Gradient descent and RHC in particular both aggressively search for the global minimum. Gradient descent always takes greedy steps in the direction of steepest descent while RHC will only accept a candidate solution that is an improvement over the current solution. The greediness of each of these algorithms results in overfitting to the training set and a model that doesn't generalize well out-of-sample. While SA has a choppy loss function, it also exhibits less overfitting. Moreover, when using gradient descent or RHC to train NN weights on a noisy dataset, early stopping should be considered to prevent overfitting as such may impair the generalization ability of the network.

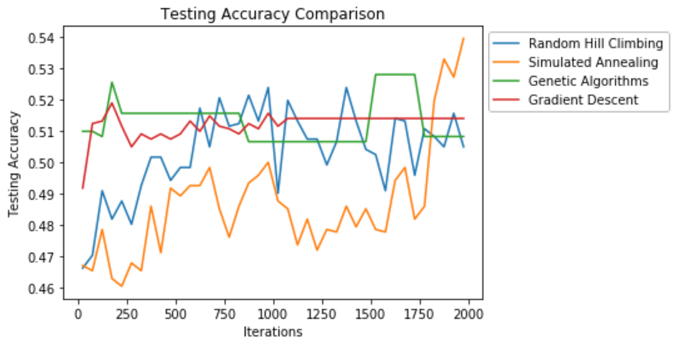

Figure 42

VI. Conclusion

Based on the performance of each randomized optimization algorithm, it's clear that different algorithms may perform better on different problems. For simple optimization problems with little structural dependencies, SA is fast and efficient. SA's performance, however, appears inversely related to the complexity of the problem; as problems become more complex with more interdependencies and local minimum, the algorithm's performance deteriorates. Moreover, with more complex problems, either MIMC or GA should be considered. MIMC specifically attempts to model structure using dependency trees and thus is most appropriate when variables exhibit conditional independence given another variable. This was present in the Flip Flop problem where the best value for x_{i+1} is conditionally independent of all values except x_i. GA may also be used to model complex relationships and is most useful when a problem can be broken down into several smaller and independent optimization problems.

While RHC, SA and GA are discrete optimization techniques, they can be applied to continuous domains by considering discrete step sizes when modifying continuous variables. When applied to the continuous domain of adjusting NN weights, gradient descent was most efficient identifying a local minimum. By adjusting all weights simultaneously in the direction that most reduces the loss function, gradient descent was able to converge on a local minimum loss comparable to that found by RHC and SA in around half as many iterations. While gradient descent is efficient and guaranteed to find a local minimum, it is also susceptible to overfitting, which is a particularly important concern when working with noisy datasets. By greedily tuning weights to minimize in-sample loss, the NN as a whole may lose its ability to generalize out-of-sample. While the addition of more data may reduce this overfitting, collecting more data often is not feasible. Moreover, with limited data, early stopping or training network weights with less greedy optimization algorithms such as SA or GA may reduce overfitting and improve generalization ability.

References

[1] Crabtree, T. (2020). NFL Scores and Betting Data. Retrieved from https://www.kaggle.com/tobycrabtree/nfl-scores-and-betting-data

[2] Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and Search package for Python. https://github.com/gkhayes/mlrose. Accessed: 09 October 2020.