Randomized Optimization

Assignment 2

CS 7641

N. Ryan Karel

# Introduction

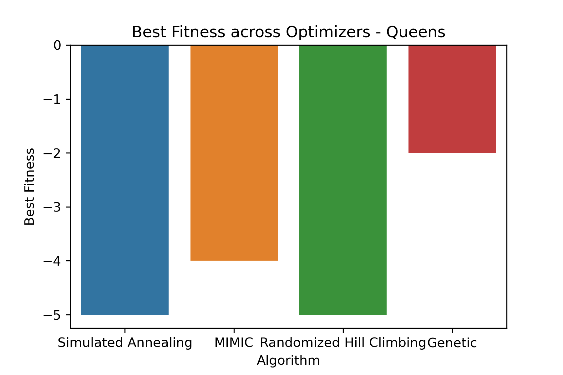
Here we investigate various randomized optimization algorithms as applied to a handful of problems that illustrate their strengths and weaknesses. We also compare the usage of randomized optimization in the context of neural network tuning against the previously explored backpropagation approach. To facilitate this, we use a dataset examined in the last assignment pertaining to wine quality. The vast majority of this assignment is performed using the mlrose library in Python (INSERT CITATION TO HAYES HERE).

# Three Illuminating Problems, Four Algorithms

We have three problems to tackle by employing four different randomized optimization (RO) algorithms: randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC (citation needed on MIMIC). Our first problem, “n-Queens” (genetic), is the classic optimization problem where better states have fewer “collisions” between queens on a chess board (citation needed from AI book). The second problem, “One-Max”, is an easier problem that all perform well on, but some are much faster than others (simulated annealing). Finally, the “4-Peaks” problem (MIMIC) that creates a context where there are known global and local optima, along with an emphasis on structure for the better states (INSERT CITATION TO ISBELL & VIOLA HERE).

## n-Queens Problem

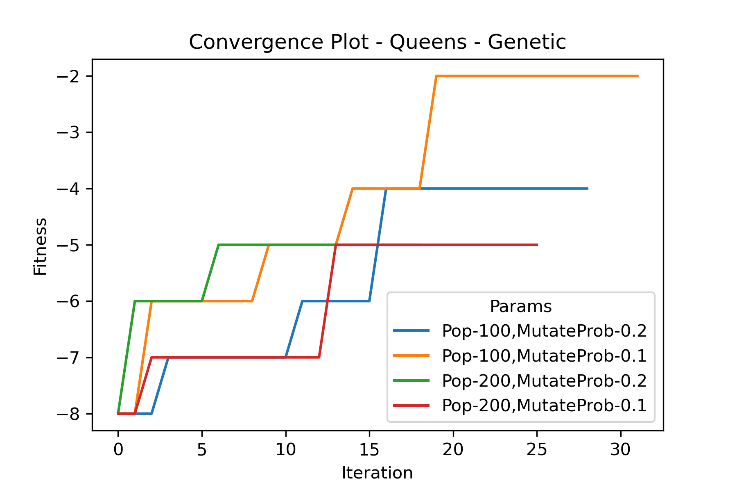
The n-Queens problem takes place on a hypothetical chess board where n queen chess pieces’ paths may intersect. Our goal is to reduce the number of possible collisions of queens by arranging the pieces optimally. Something interesting about this problem, aside from how easily visualized it is by a human, is that states that perform well globally will also perform well on “sub-boards”. For instance, if there are few collisions of queens across an entire board, it stands to reason that the left half of the board also has few collisions, as does the right side. This emphasis on locality may allow some algorithms to succeed where others do not, a prediction that we shall soon see realized by the genetic algorithm shortly. Note that although this is traditionally a minimization problem, we may negate the fitness values (measured as the number of possible collisions between queens on the board) to convert it into a maximization problem. Let us apply each of our available algorithms to the problem and see which performs the best.



Note that each of the algorithms have been tuned to this problem over several hyperparameter options. For the sake of brevity, we will only examine the hyperparameter tuning process of the selected algorithm for each problem. Furthermore, note that the standard bit string size we use for determining which algorithms perform best is 16, though we do examine the impact of problem size on performance, too.

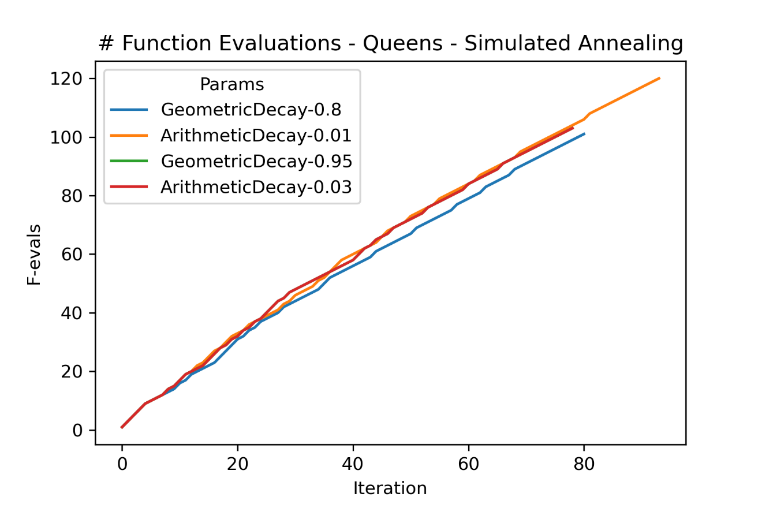
After all of this tuning, we see that the genetic algorithm has the highest score, which is shown here as the least negative score. This stands to reason as we contemplate the algorithm, wherein states are “bred” with one another to produce new states. Recall the example we gave earlier in this section about how a strong whole board may be decomposed into a strong left half sub-board and a strong right half sub-board. Theoretically, the process could work well in reverse, the order in which the algorithm may take strong components and breed them together to form an even stronger whole.

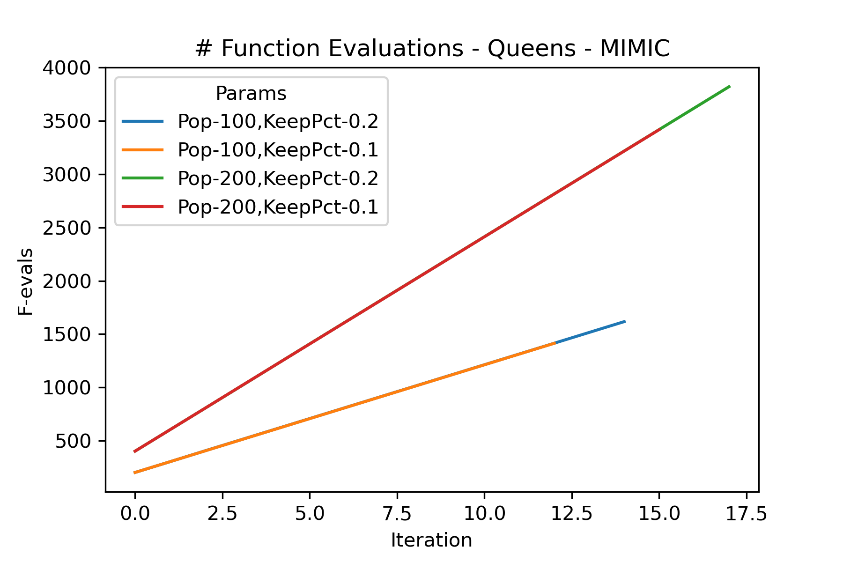
Let us examine the convergence curves for the various hyperparameter options we have for the genetic algorithm.

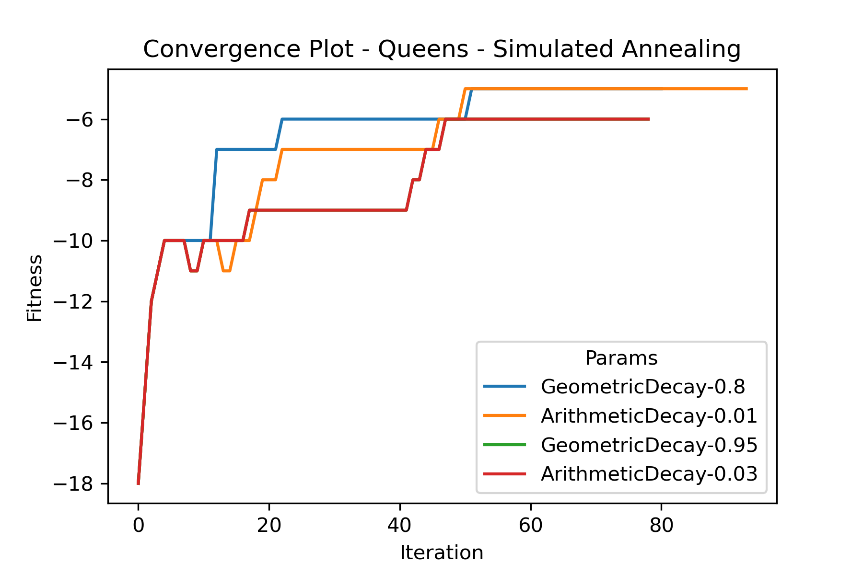
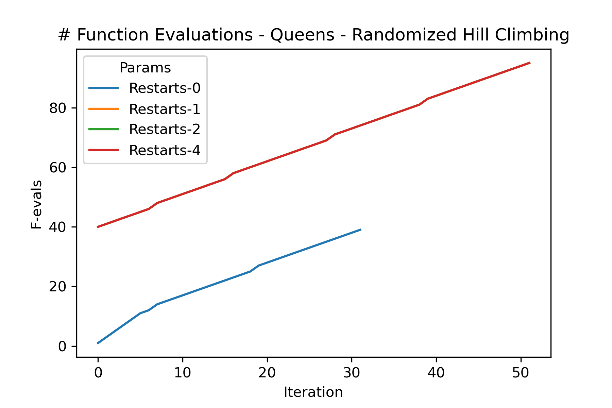
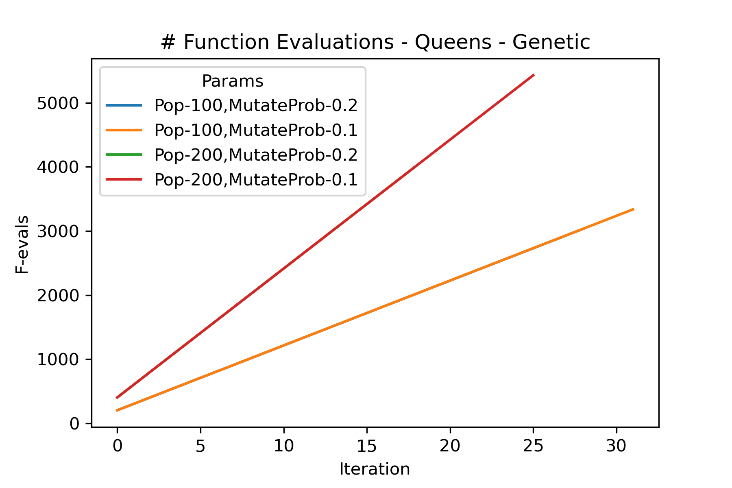
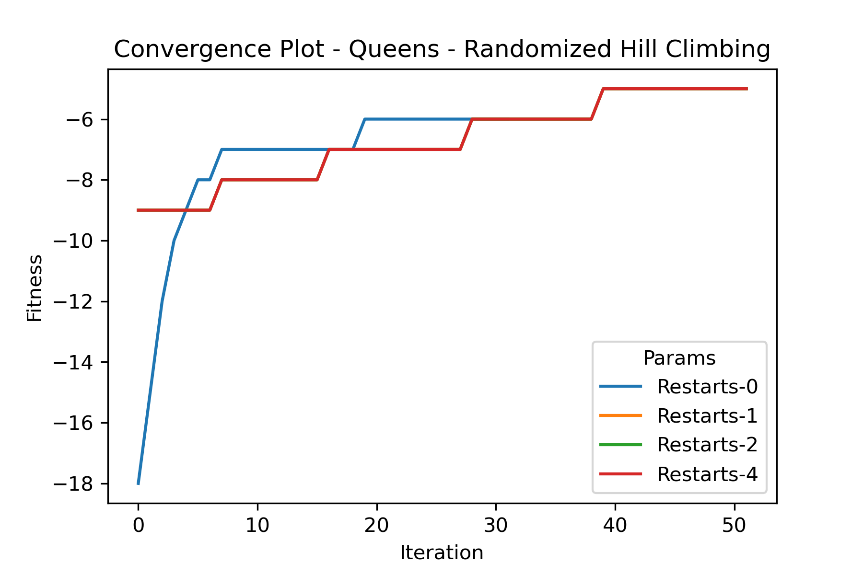
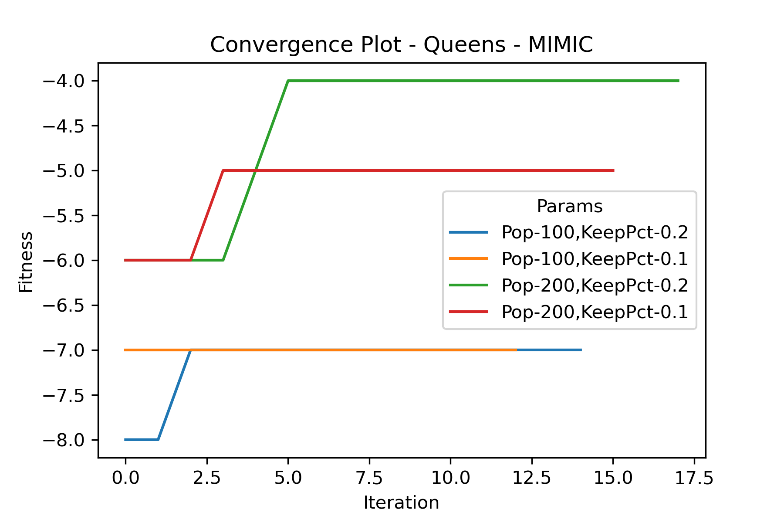


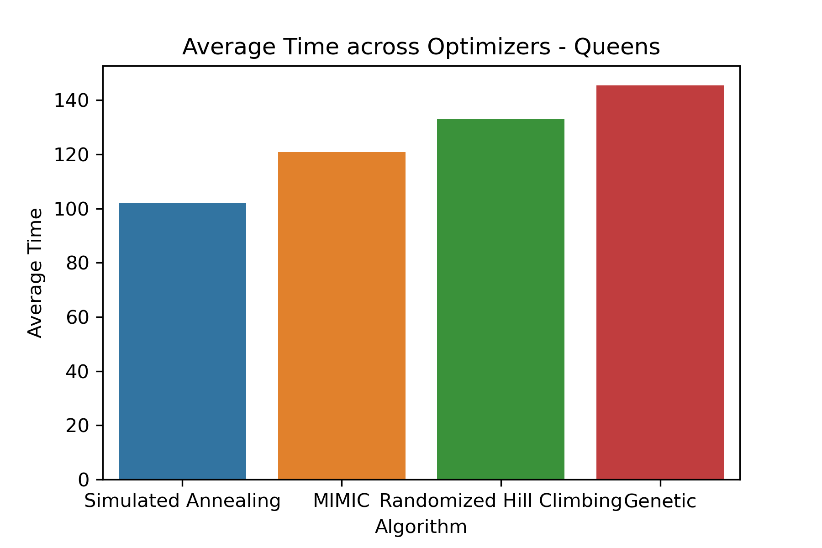
We see that the smaller population (100) and smaller mutation rate (0.1) tunings won in the end. The smaller mutation rate is likely due to the fact that strong sub-boards will generally not be made stronger (even when combined with another strong sub-board) by changing some of its own components, likely weakening the fitness of the sub-board. Instead, it appears there was a greater emphasis on simply retaining and breeding together good sub-boards. We assumed a 75% breed percentage for all the possible tunings.

Note that although the genetic algorithm performed best, it also took the longest (determined by an average across the different hyperparameter values). We will find a clue for why this was the case by reviewing the convergence plots for the other algorithms. Although the total number of iterations for RHC was higher than for genetic, the algorithm in general requires less time per iteration given the substantially smaller number of function evaluations required. There were a couple different orders of magnitude of evaluations required for RHC (under 100) than for the genetic algorithm (thousands). This is expected when we consider how many candidates the genetic algorithm is considering at any iteration compared to the RHC algorithm. The MIMIC algorithm, on the other hand, had a similar number of function evaluations but far fewer iterations. This also aligns with our prior beliefs, which state that the MIMIC algorithm generally has more computationally intensive iterations, but fewer of them. We also see this born out by the MIMIC algorithm finding its local optima earlier than its peers, at least in terms of earlier iterations. Lastly, we also see a small number of function evaluations and iterations for the simulated annealing algorithm, explanations that, when combined together, illustrate why this algorithm had the smallest mean time to convergence.

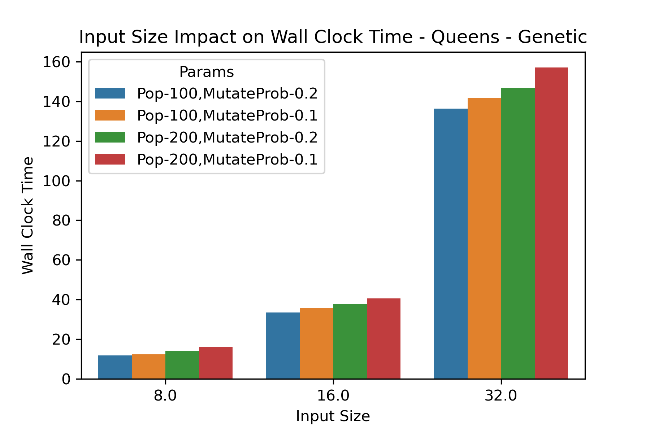
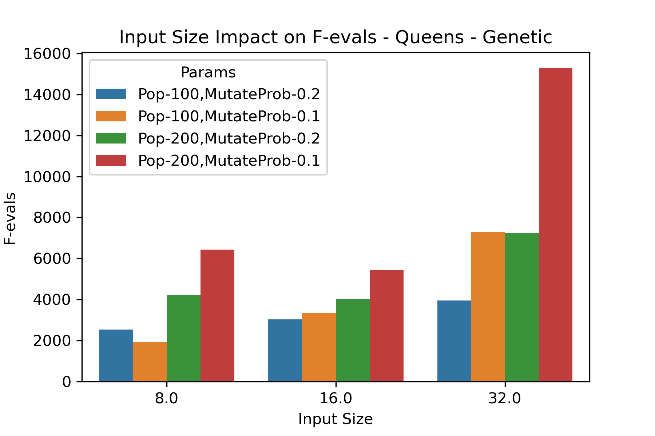
q

.

.

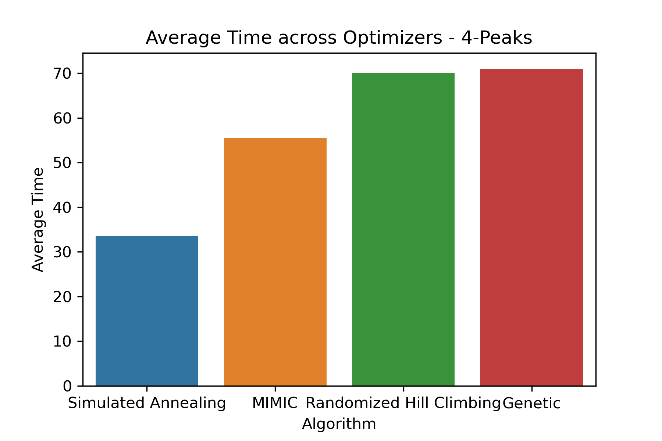
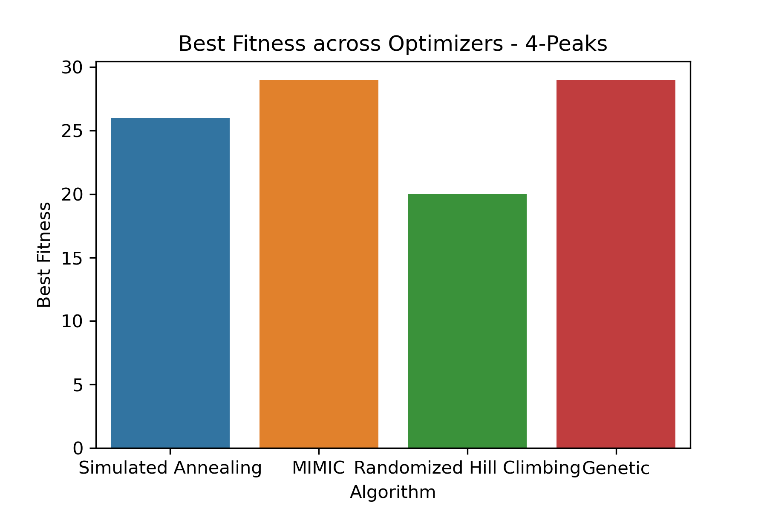


It may also be instructive to view the number of function evaluations as a function of input size into this algorithm. Recall that, because we are in the n-Queens problem domain, the states themselves are not bit strings but rather sequences of numbers between 0 and n-1, inclusive. Given the larger state spaces at play, we might expect the increases in input size to be smaller (in terms of proportions) than the required function evaluations to search the space. However, when we review the data seen below, we find this not the be the case. This is likely due to the fact that there is a fixed population size for a given set of hyperparameter options, not one that also grows in proportion to input size. As such, it appears that the genetic algorithm is able to grow its required function evaluations roughly linearly as a function of input size. We do however see the trend expected when reviewing wall clock time instead of function evaluations.

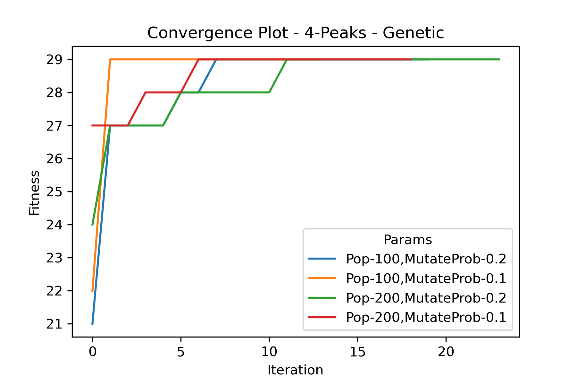
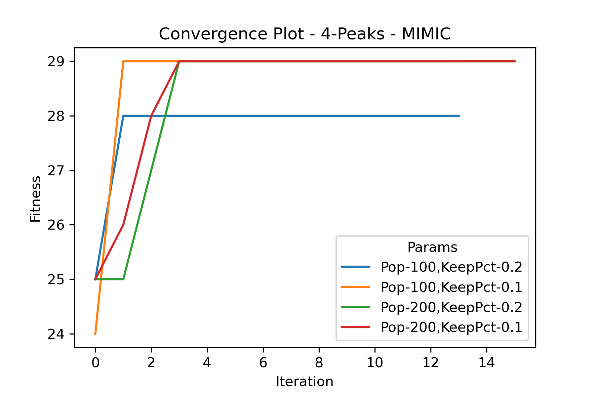
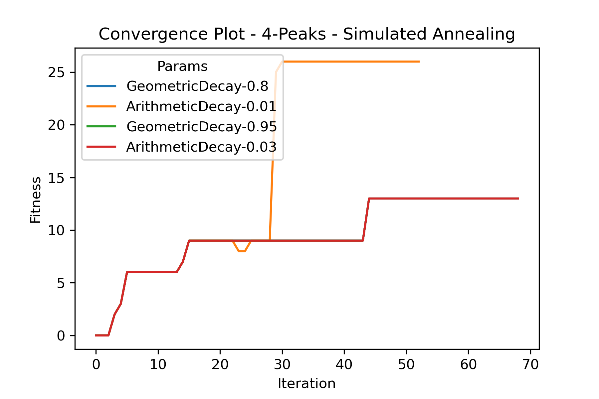
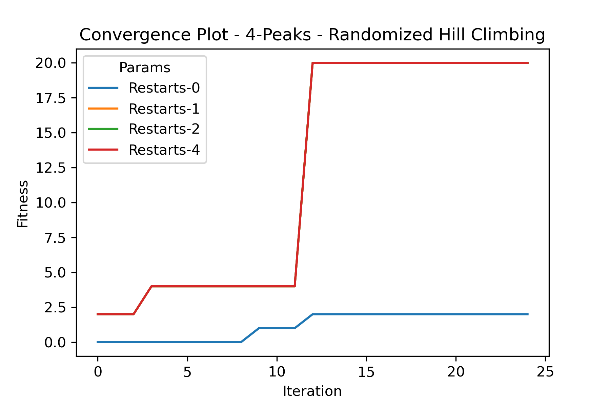


## 4-Peaks

The 4-Peaks problem is constructed intentionally such that there are known global and sub-optimal local optima known in advance. For this particular discussion, we use a T set to be 0.1 of our selected bit string size, following the parameters selected in Professor Isbell’s paper (INSERT CITATION HERE). First things first, let us see which algorithms perform best on this particular task. We can see that both the MIMIC and genetic algorithms perform best for this problem, though when we review the average time plot we can see that MIMIC reaches its verdict faster than the genetic algorithm.

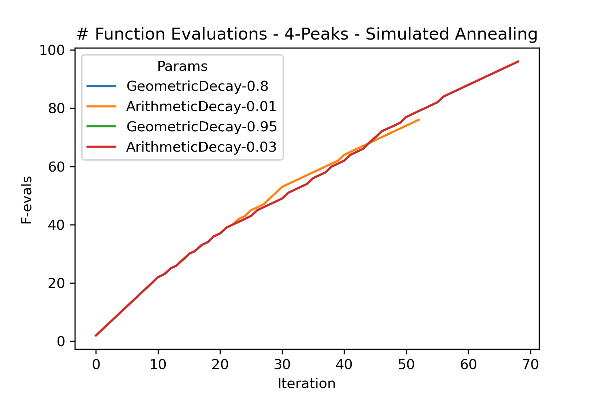
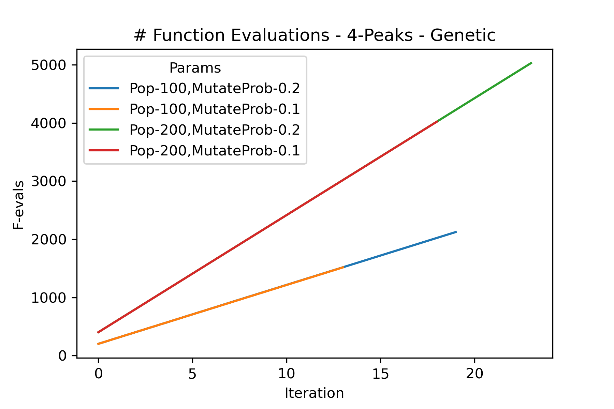
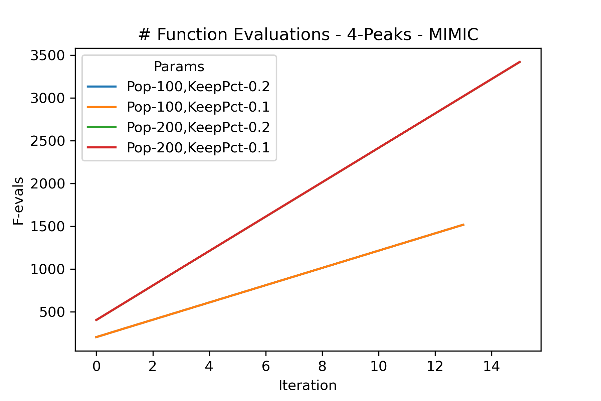
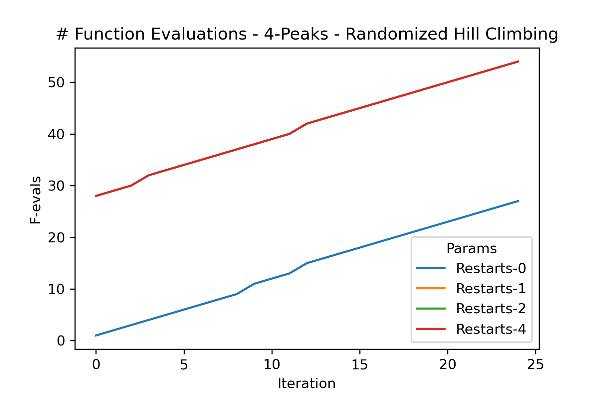


Why might the MIMIC algorithm be better suited to this task than its peers? Recall the structure of this particular problem. The goal is to realize a particular structure of the state space wherein there is a certain number of bits of the same value followed entirely by bits of the other value. The more highly rated state instances follow a particular structure, so we need an algorithm that can capture this structure and then optimize within that context. The MIMIC algorithm was designed to do just this, specifically through its ability to articulate some estimate of probability density at various points in the state space. As the algorithm explores this space, it begins to quantify the impact of interactions amongst state dimensions against the likelihood of optimality. We see its performance at correspondingly accurate and fast levels. Let’s examine the convergence plot for the candidate algorithms.

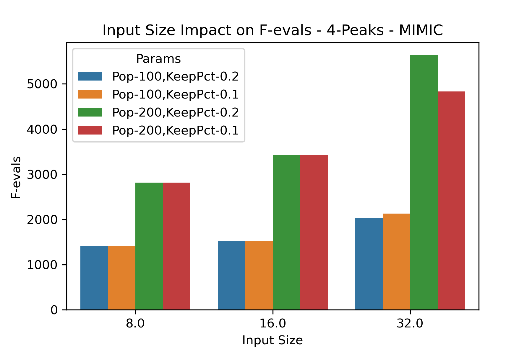
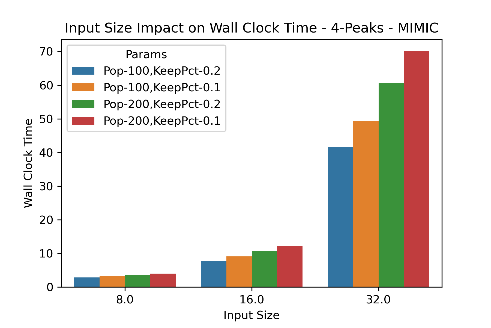


Similar to some of the behavior observed back in the Queens problem, we see a relatively small number of iterations for MIMIC, but each more computationally intensive than the iterations of its peers, excluding the genetic algorithm. There is a great deal learned at each iteration of the MIMIC algorithm, which we would expect given the large number of function evaluations even in the early iterations. MIMIC requires these evaluations to begin to model the relationships between the input dimensions. Returning to the convergence plot for MIMIC, amongst three of the hyperparameter tunings we can observe that the information learned in the first one-to-two iterations is enough to take the fitness levels up to the highest values achieved by any of the algorithms. In terms of the best hyperparameters, the similar performance levels of the values shown here cause us to defer to the wall clock times and function evaluations graphs to make our final decision.

As for the other algorithms, the genetic algorithm is also quick to achieve the same level of fitness as the MIMIC algorithm, but not quite as quick. It does receive some of the same benefits as we observed back in the Queens example, where locality was values highly. Having that said, the ability of MIMIC to hone in on structure allowed it to out-perform MIMIC here. As for randomized hill climbing and simulated annealing, we can clearly see from the convergence plots the difference in how much information is gleaned from the samples from the state space in MIMIC versus these competitors. Both of these algorithms have reasonable parameterizations that are leaping in their fitness values (as a function of iterations) fairly late in their runs. This speaks to the substantially smaller “memory” retained by these algorithms.

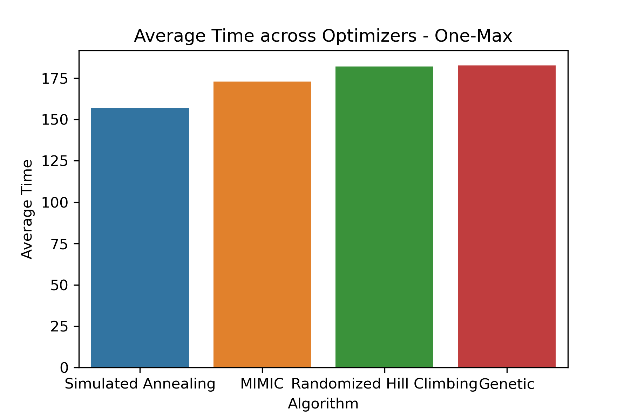
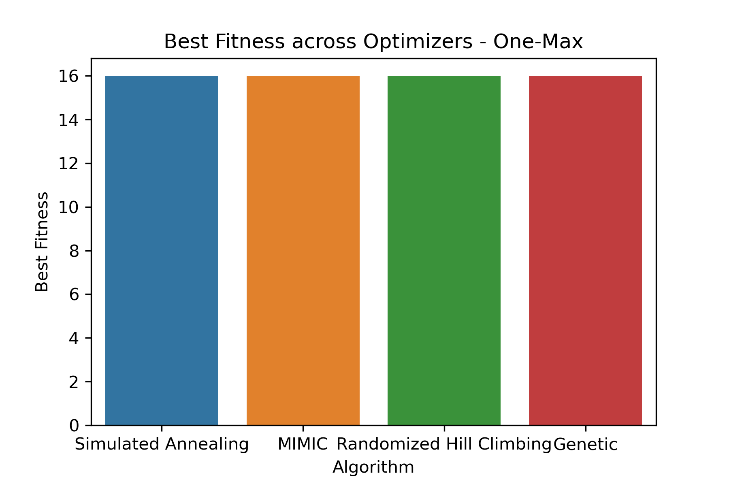


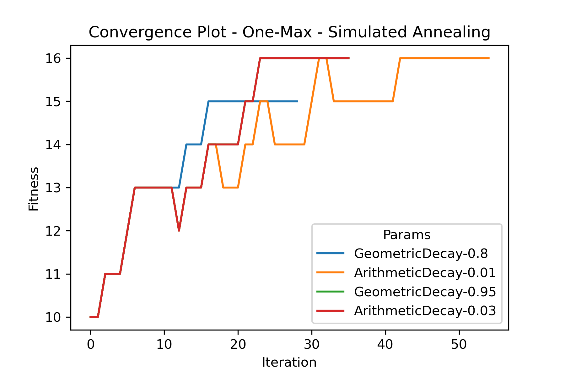
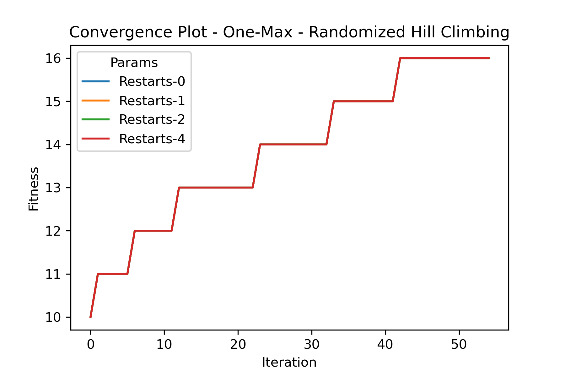
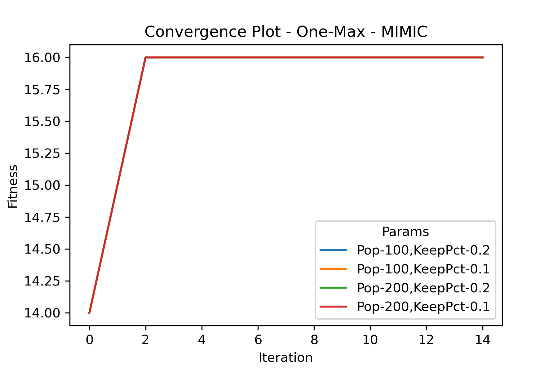
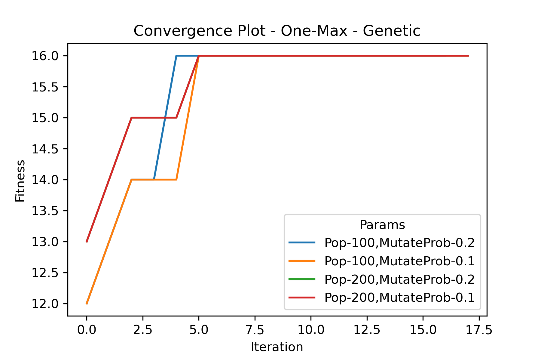
Let us review the wall clock time and function evaluations required for the MIMIC algorithm as applied to the 4-Peaks problem. Naturally, we see a correspondence between the hyperparameters that were fastest to converge to their optima and the fastest wall clock times amongst the smaller population hyperparameter selection. For this particular problem, it appears that the 100 value is to be preferred over the 200 value. Perhaps the relatively simple structure of the optimal states lends itself to more targeted retained populations (lower population size, lower keep-percentage). Using the convergence graphs and input size impact graphs below, we can confidently say that population 100 and a 10% keep percent is the optimal tuning for the 4-peaks problem, at least amongst the options we explore here.

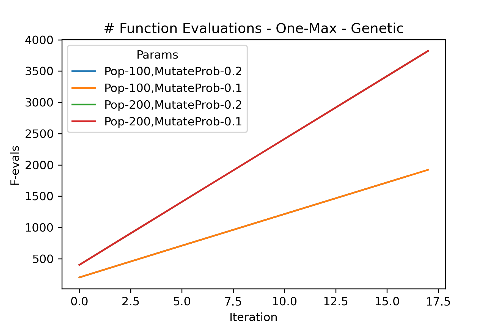
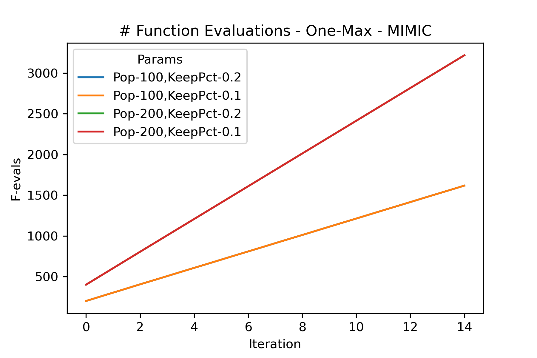
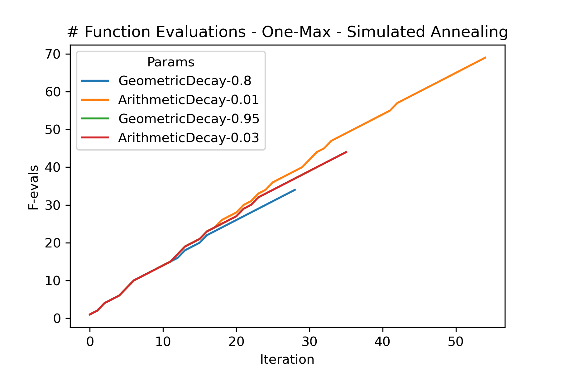
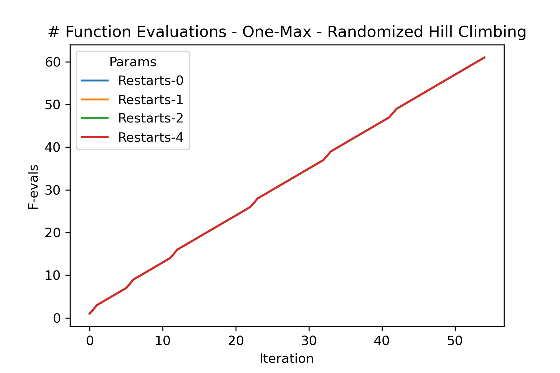


## One-Max

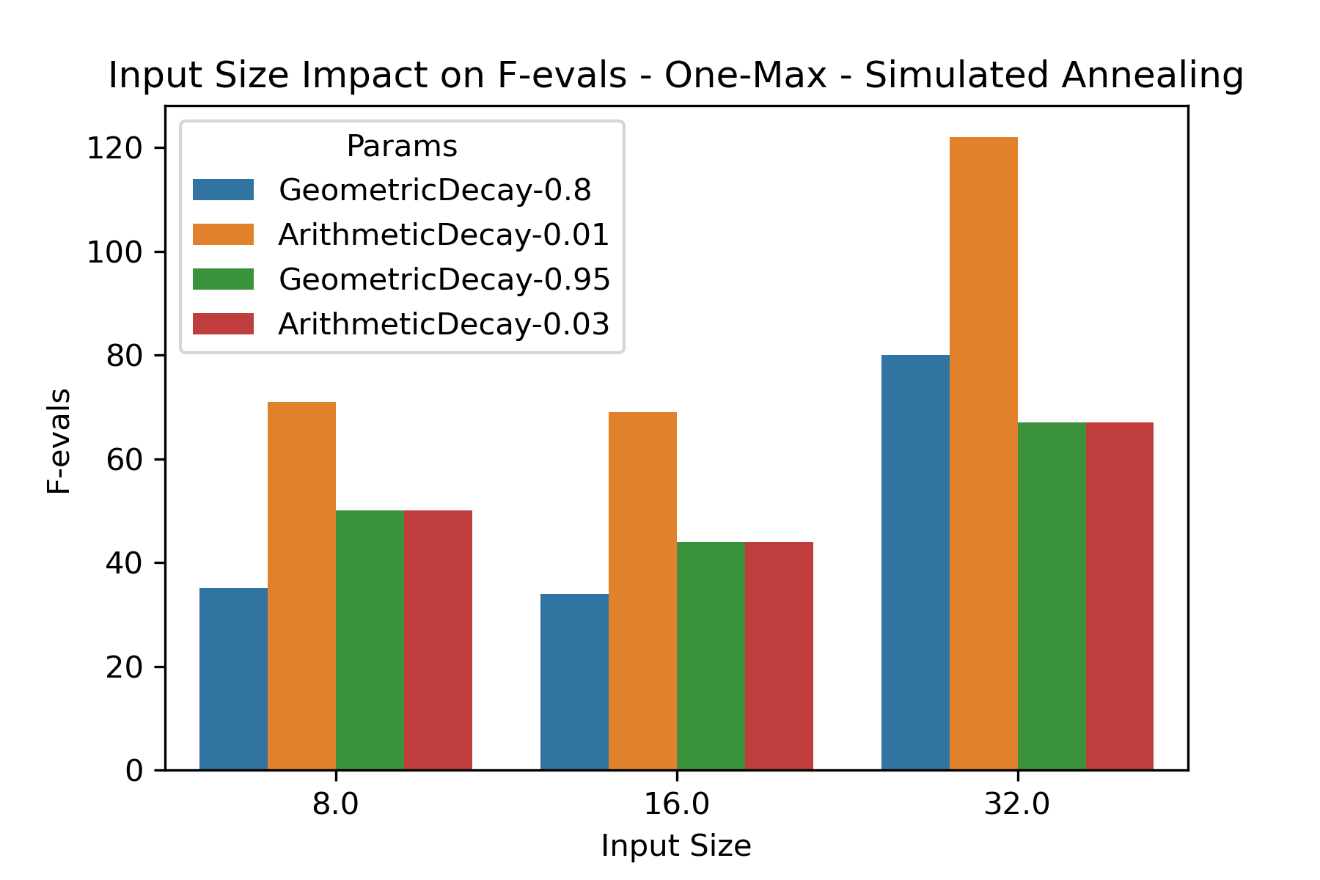
We now look at a more straightforward problem, the One-Max problem. This problem is interesting because it could hardly be more different from the n-Queens and 4-Peaks algorithms, where the genetic and MIMIC algorithms shined with their ability to emphasize locality and structure, capturing and retaining relationships amongst the dimensions of a state space. Here, the One-Max problem will differentiate the candidates not by seeing which can determine the global optimum (all can do this), but by seeing which one does so most efficiently. Right off the bat, we see simulated annealing emerge as our winner, though not by a massive margin. Even still, simulated annealing is the right tool for this job. Let us examine plots below to uncover why it is the more efficient choice.

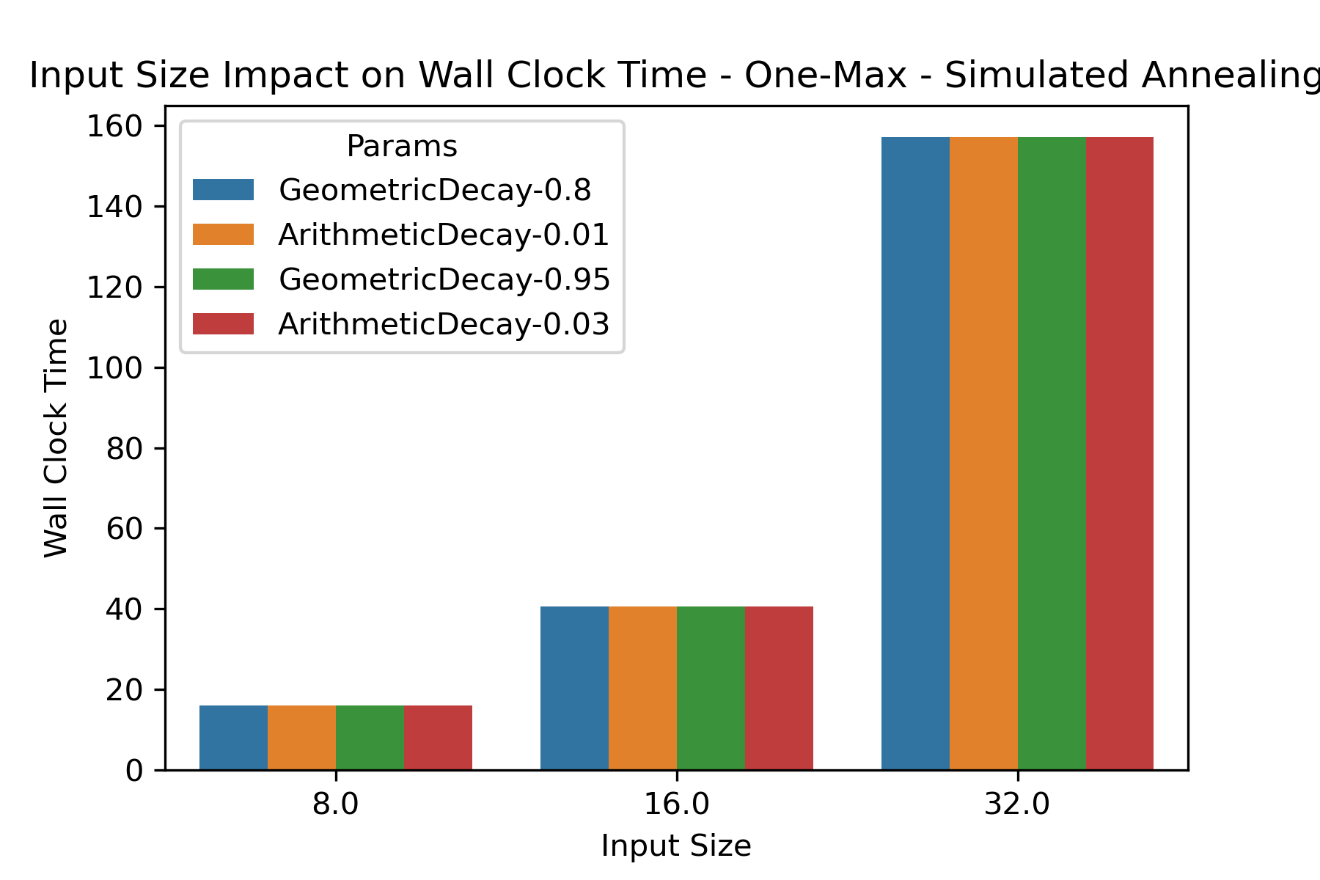






Though it requires more iterations than the genetic and MIMIC algorithms, the simulated annealing approach requires far fewer function evaluations than the algorithms that out-performed it on the prior problems. This makes sense within the context of both the problem itself and the SA algorithm itself. Firstly, the problem can be attacked without considering any relationships amongst the input variables. Instead, each algorithm learns at varying speeds that ultimately it just needs to flip each bit to 1 to be successful for this problem. The simulated annealing algorithm requires no population of states at each iteration to learn this. Note that using the information in the SA convergence plot, we can see that the lower temperatures (higher decay values) are the right choice for this problem. With no other local optima to deceive the optimizer, it can take the improvements with little need for detour and little need for a larger scale memory.





We also encounter a similar set of trends on the function evaluations and wall clock time as seen for earlier problems and their corresponding best performers. There are not that many more function evaluations required as input size grows, but wall clock time grows at a rate higher than linearly, perhaps quadratically as viewed in the charts.

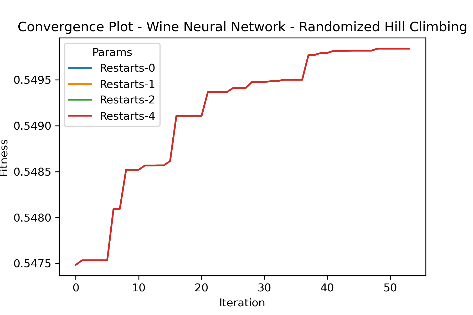
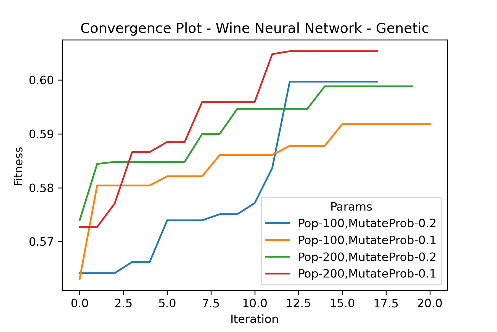
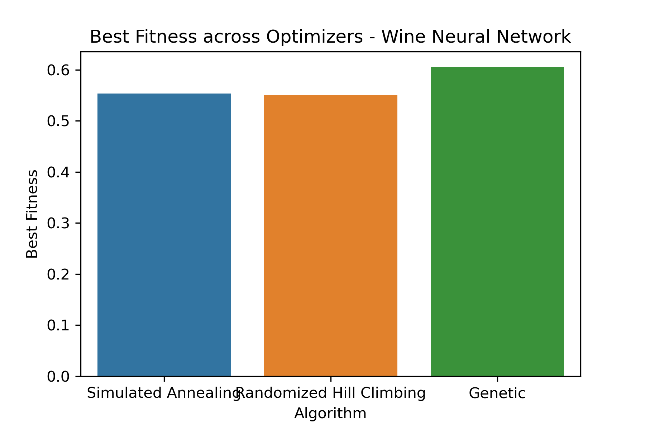
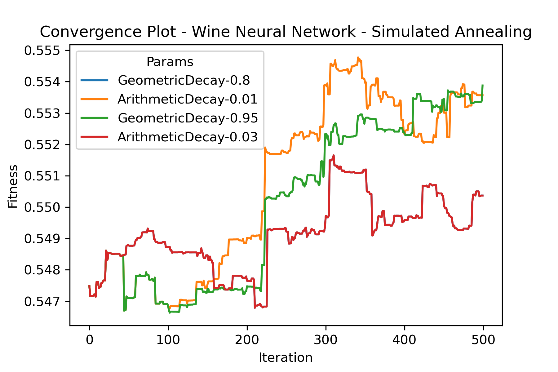
# Neural Network Tuning

We revive a dataset and model from the prior assignment to see how the techniques discussed above (excluding MIMIC) fare in a more challenging context: tuning neural network weights. First, we must apply these techniques to a new continuous context, leaving the discrete world behind. Another added complexity is the size of state space. Whereas before we worked in terms of bit strings sized 8, 16, or 32 in length, we now graduate to hundreds of parameters to tune. For the sake of contrast, we will compare the resulting neural networks derived from our randomized optimization techniques with the MLPClassifier (INSERT CITATION TO SKLEARN HERE) object learned using backpropagation. Similar to Assignment #1, the ultimate test between the approaches will be performance on a holdout set, where performance will be measured using the ROC AUC OvO metric described in my prior report (CITATION NEEDED?).

When training the models using the randomized optimization techniques above, I tried to match the same hyperparameters used in the prior assignment. There we used an alpha value of 0, the logistic activation function, a constant learning rate of 0.001, and a single hidden layer of size 50. Fortunately for myself, we can access the coefficients and biases (or intercepts) of the scikit-learn implementation directly and update those using the mlrose algorithms, making a one-to-one comparison.

With the RO algorithms fit to the training data, we can see lackluster performance from randomized hill climbing and simulated annealing. The genetic algorithm is definitely the winner amongst these, though it pales in comparison to the original backpropagation algorithm. To understand why, the notion of locality may play another larger role here. The 1-d array that stores the state of the algorithm has the coefficients next to each other at the beginning of the array and the biases next to one another at the end. Presumably, an algorithm like the genetic algorithm which breeds portions of the state with one another may be able to combine strong coefficients with strong biases to form a stronger whole.

For the final test, we compare holdout set ROC AUC OvO using the old algorithm with that of the best performing randomized optimization approach, which is the genetic algorithm with population 200 and mutation probability 10%. The backpropagation approach, which leverages a precise knowledge of how each coefficient interacts with another, outperforms the genetic algorithm by a longshot. The genetic algorithm scores a 60% on the holdout set, and the backpropagation approach scores a 76%. It appears that the supervised approach provides a substantial amount of information that the learner can use, whereas the genetic approach is too lacking. Still, this was an interesting exercise in seeing some alternative approaches in tuning a neural network.

 Chart, line chart

Description automatically generated