Analysis of Unsupervised Learning for Randomized Optimization

Andrew Shi

CS 7641

October 15, 2023

I. Introduction

In this assignment, multiple simple techniques involving unsupervised learning with respect to randomized optimization will be explored in depth. To understand these simple algorithms, it is important to understand how they behave under a variety of circumstances. The following techniques will be the focal point of this exploratory analysis: random hill climbing, simulated annealing, genetic algorithms, and MIMIC. Because the analysis focuses on the behavior of these algorithms, their implementation will be scarcely discussed. There is ample documentation online explaining the underlying theory if the interest of the reader is piqued.

To analyze how these algorithms work under certain circumstances, this paper will use them in a variety of optimization problems, which describe a "fitness function" that the algorithm attempts to maximize (or minimize) through calculation. The optimization problems considered are One Max, 4-peaks, and Travelling Salesman. In One Max, an initial state of a string of "bits" is given, and the fitness is maximized by maximizing the number of ones in the string. In 4-peaks, a string of "bits" is given, and fitness is determined by the maximum between the number of trailing "0's" and leading "1's". In Travelling Salesman, a list of coordinates is given, and the fitness is maximized when the path that traverses the coordinates is minimized.

II. Setup

This project made extensive use of the following libraries: time, numpy, mlrose_hiive (Machine Learning, Randomized Optimization, and SEarch). The time library was used to calculate the equilibrium times for different fitness functions of all algorithms. mlrose was developed for instructional teaching of randomized optimization for Georgia Tech's CS 7641 and contains all of the aforementioned optimization algorithms, as well as a number of fitness functions, with capacity to introduce custom fitness function definitions. This analysis made use of discrete optimization problem class inherent in mlrose, which takes in a function as an input, and can be applied to any of the algorithms.

For all tests unless otherwise specified, the following parameters were used:

Parameter	Value
Max Iterations	10000
Max Attempts	200
Population Size (GA and MIMIC)	100
Restarts (RHC)	10
Mutation Probability (GA)	0.1
Keep Percentage (MIMIC)	0.2
Random Seed	1

The notebook – and this analysis – contain several plots for various reasons. For consistency, the algorithms are color-coded in every plot as such: Hill Climbing – Blue; Simulated Annealing – Orange; Genetic Algorithm – Green; MIMIC – Red.

III. One Max

Perhaps the simplest of the fitness functions, One Max returns the count of ones in a given bitstring, and seeks to maximize that count. When run using the four algorithms, the following plot is produced for the fitness and number of function evaluations by

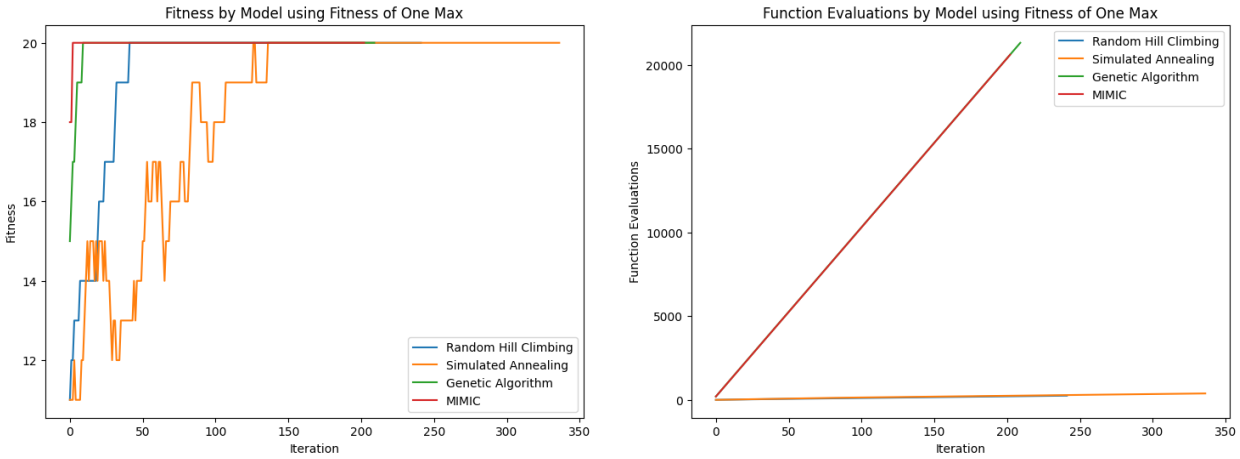


Figure 1. Fitness and function evaluations by iteration of with One Max fitness function.

In this problem, a bitstring input of size 20 was used. Preliminary analysis yields the fact that all of the of the algorithms reach a perfect fitness function after a number of iterations. Furthermore, the number of function evaluations for MIMIC and GA far exceed those of SA and RHC. Note that the convergence for each function occurs when a function achieves the same fitness for an extended period, exactly 200 consecutive iterations, our maximum attempts parameter.

Because of the iterative sampling nature of GA and MIMIC, where a number of samples are generated, then a subset of the samples is taken to form a second generation, these algorithms converge on the perfect fitness function quickly. This rate at which they converge is proportional to the population size, which is set to 200 samples per iteration. Because the keep percentage of MIMIC is set to 0.2, the best fifth of samples is used as a threshold, and only better samples will be generated with at each iteration, leading to a quick increase in fitness, even faster than genetic algorithms, which tracks point estimates instead of population densities. The downside, however, is that because these two keep track of large populations, is that each iteration requires a large number of function evaluations. Because each tracks a population of 100 points, each point found in each iteration must be evaluated using the fitness function, resulting in a slope of 100 evals per iteration. This will be a trend that follows for the following fitness problems.

When examining RHC, the function seems to reach its maximum within 50 iterations. The fitness of the RHC plot can be described as monotonically increasing, as a change is only made when a better state is found. In the first few iterations, large changes are made, and RHC quickly comes near the maximum fitness, since more favorable states are available. As the fitness approaches 20, the more iterations are needed to find a favorable state. The number of function evaluations that this function uses increases linearly as well, only needing a single evaluation to determine if a new state is favorable. In each iteration, a single point is sampled and judged on whether it is to be accepted, giving rise to a linear increase in total function evaluations.

SA, however, seems to require far longer to stabilize at the maximum fitness function with the same rate of function evaluations as RHC. Note that SA begins by moving at a similar rate as RHC, but stagnates with consecutive or large negative jumps. One can imagine that SA performs extremely poorly, especially from the perspective of number of iterations run for the One Max function because of the lack of local minima in the problem space. Therefore, SA often chooses poor states when completely unnecessary, and does poorly in this problem.

The following plot shows the fitness achieved as a function of input size, starting from four and traversing through forty.

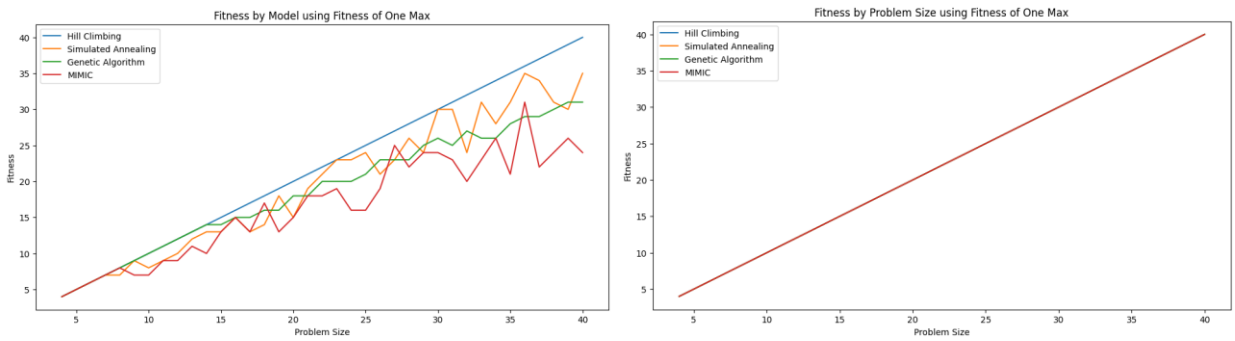


Figure 2. Fitness given by the One Max fitness function as a function of input size. Left shows 1000 iterations with population size 10. Right shows normal conditions.

This test was run under two separate conditions. Under normal conditions, all algorithms reach perfect fitness values. To get discernable results, the input parameters were changed. Because, the fitness function for One Max is a purely convex function, as there is only one critical point – the global maximum – when the input bitstring is all ones. Any favorable step, then, would force the RHC to converge on this global maximum until the end state is reach. As RHC implemented is set to iterate 1000 times, it has ample time to reach the maximum fitness function, and make no further changes.

Note that the other algorithms perform similarly well, with a moderate degree of variation. All algorithms are perfect up to a problem size of 7. GA seem to increase fitness more consistently than SA and MIMIC, which have large variations in output fitness. GA seems to be far more consistent because of the nature of offspring generation that the algorithm utilizes. Once a population is generated, and fitness values are found for each sample, the best of the population are selected (through a probabilistic distribution proportional to fitness). A crossover is performed on the most fit to produce the next generation of offspring. The nature of this generation algorithm tends to work well with problems that represent mutually independent features, as One Max does. The decrease in population size plays a large role of these algorithms to converge on the maximum value, and limited iterations further decreases their chances of reaching maximum. Less samples examined in each iteration translates to less increase in the threshold at each iteration. When 10 samples are found in a space of 2^{40} points, increasing the sample space is very difficult, which results in the low and inconsistent fitness of MIMIC.

SA demonstrate great variability in fitness as problem size increases when given less iterations and less population size. One can understand from Figure 1 that SA depends largely on having a large number of iterations, and when capped at 1000, a “hardened” state is not reached and variance is common.

IV. Four Peaks

In this section, the Four Peaks problem will be analyzed through the lens of the four fitness functions. The Four Peaks problem is an optimization problem where the fitness is the maximum between the number of leading ones or trailing zeros in a bitstring. Additionally, if both counts are greater than a certain threshold, a bonus on n is added to the fitness value, where n is the length of the bitstring. For this problem, the threshold is defined to be 10 percent of the overall bitstring. Running this algorithm under our defined conditions for a bitstring of size 10 yields the following plots.

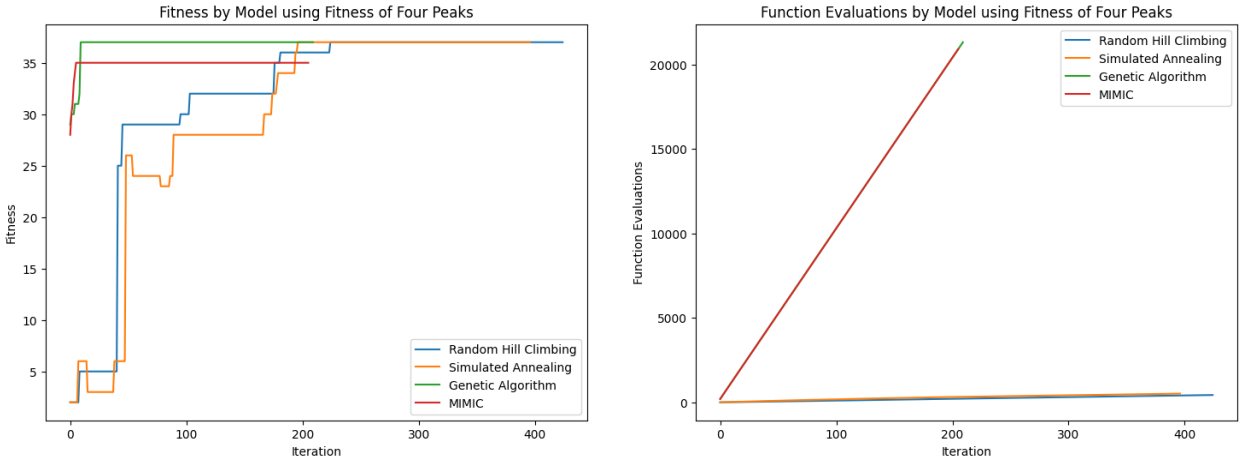


Figure 3. Fitness and function evaluations by iteration of with Four Peaks fitness function.

After about 400 iterations, all functions converge. Again, MIMIC and GA tend to find solutions extremely quickly, within the first 20 iterations, and then accepted in converged. Looking at the number of function evaluations, MIMIC and GA again run 100 evaluations per iteration, one evaluation for each point in the population, as with One Max, resulting in about 20000 total function evals taken. Note that the MIMIC algorithm is the only one to not find the maximum fitness of 37. In the following 200 attempts, no improvements are made, and the algorithm converges. Because of the high attempt count, the probability distribution created from the first iteration may be inaccurate leading to a nonoptimal fitness.

Again, RHC and SA call one function evaluation per iteration, resulting in shadowed function evaluation plots. RHC again slowly converges on the final maximum fitness value. Note that around 45, there exists a large jump in fitness, of exactly 20. This is due to a random neighbor state being able to fit the bonus function. A similar jump exists in that of SA, at around iteration 50. Note that SA will almost never make a jump down by 20, as this a highly, highly unfavorable action. Therefore, once the bonus function is "found", the bonus will not be dropped again. Because of negative jumps, SA is also the slowest to reach maximum fitness. It is important to note that the maximum number of attempts parameter is crucial in determining the performance of SA and RHC. If set too low, these algorithms risk having their runs be cut short due to iteration periods where no improvements are made, indicated by stagnant fitness with respect to iteration.

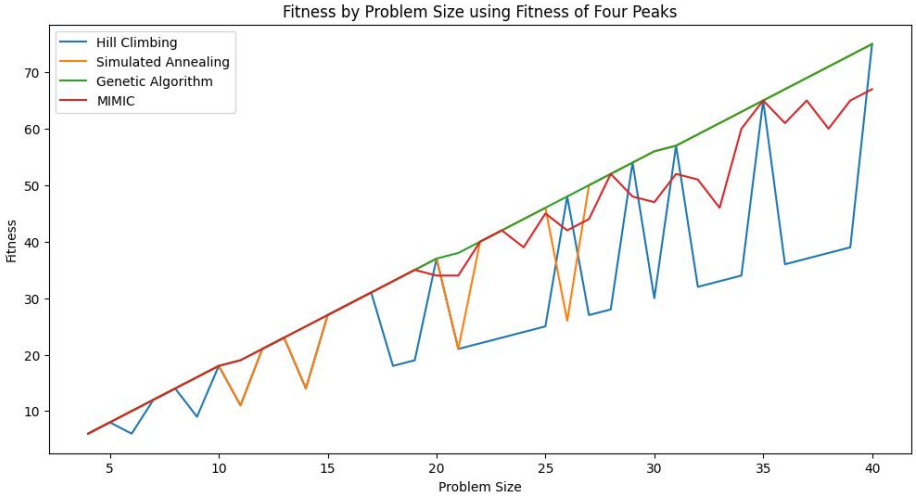


Figure 4. Fitness given by the One Max fitness function as a function of input size.

Upon inspection, two “bounds” seem to appear, most notably by GA and RHC. This wedge shape describes the possibly of finding or not finding the bonus function, and is notably highlighted by the “bouncing” by RHC, as sometimes, the random search stumbles upon the bonus, and sometimes does not. To a lesser extent, SA also bounces, on four occasions not being able to find the bonus. Why do these tend to not find the bonus, as opposed to GA and MIMIC? It is because of the nature of neighborhood searches used in RHC and SA. When a state is found that has great fitness score without bonus, say, a string of all ones, it is highly unlikely that a neighbor that is, say, 111…0000, will be selected from the random pool of potential neighbors, and will not be accepted. This rate of course decreases with input size, and hence RHC does poorly on larger bitstrings. Of course, SA always has potential to select an unfavorable state transition, and that is why it tends to find bonuses more often.

When analyzing the upper bound, noted by GA, the function seems to do perfectly. One might notice the slight notches that occur periodically at problem sizes that are multiples of 10. Of course, this can be explained by the bonus threshold parameter; that is, because the threshold is set to 10 percent, when the bitstring size increases from 19 to 20, an extra trailing zero or leading one is necessary. MIMIC tends to follow this upper bound decently, as least for low problem size. However, as problem size increases, MIMIC starts to perform poorly, yet with less variance than RHC, as it finds the bonus function in every problem. Even so, it often fails to find the optimal state, possibly from a lack of large population size.

V. Travelling Salesman

As per the famous problem, the travelling salesman optimization problem presents a list of points which must be ordered such that their round-trip traversal is minimized. For this specific problem, points are represented by two-dimensional Cartesian coordinates, and the distance metric used between points is simply the Euclidean distance. As opposed to bitstring problems, greater restrictions on the input strings allowed; because each point must be visited exactly once, the only valid states are those that are a permutation of the input points. This results in a far more difficult problem to solve, space grows factorially, while bitstring space grows exponentially. For this specific problem, however, mlrose offers an optimizer specifically for the travelling salesman problem, as the discrete optimizer used in other sections does not work. The final fitness value returned is the total Euclidean distance generated from the list of points. Note that the problem seeks to minimize the fitness.

When run with the four algorithms, the following plot is produced:

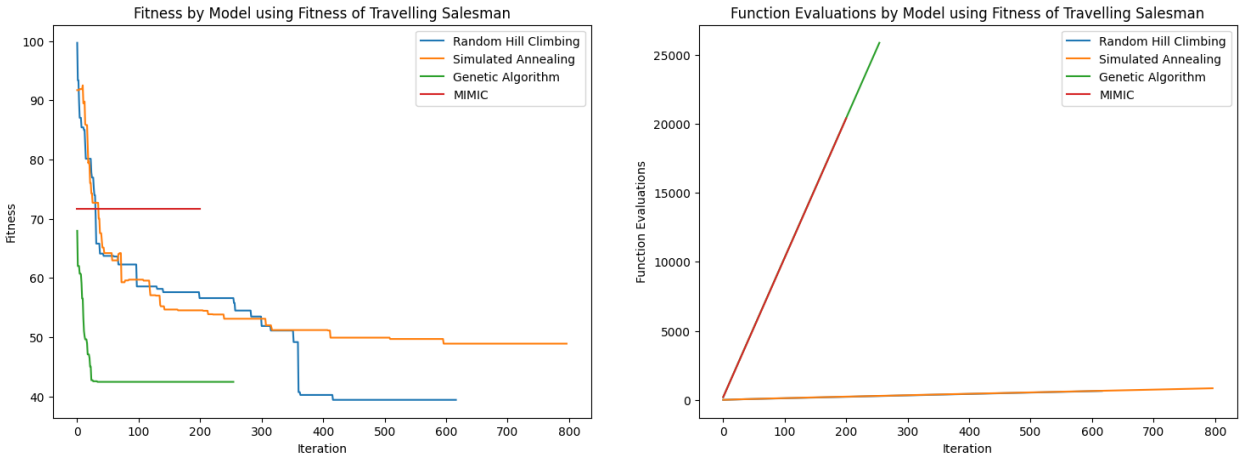


Figure 5. Fitness and function evaluations by iteration of with Travelling Salesman fitness function.

Again, note the quick increase in number of function evaluations done by MIMIC and GA, proportional to the population size. However, in this case, MIMIC converges on a solution immediately and refuses to search further. MIMIC seems to perform poorly at this task, likely due to the extremely complex solution space its attempting to model. To compensate for this fact, MIMIC would need a significant increase in population size, to be able to sample a larger population for a more accurate grasp of the solution space.

On the other hand, the other algorithms – especially RHC and SA – perform a good amount of searching before convergence, 600 and 800 iterations respectively. Note that in this run, very few unfavorable jumps were made in the SA model. This is likely because of the larger scale of fitness that exist on this problem. A negative jump, therefore, which is calculated from the difference of fitness, is therefore less likely. The lack of unfavorable jumps renders this algorithm to behave very similar to RHC, and makes it susceptible to local minima, which it seems to do in this example. However, this should be explored deeper with different random seeds. Around iteration 370, RHC seems to make a large jump to drop the fitness by about 10. This is likely due simply to random chance. Again, this should be analyzed further with more random seeds.

GA tends to quickly solve its problem in terms of iterations, finding a solution to converge on at 50 iterations. The downside of this however is the large number of function evaluations it takes to keep track of a large population of points.

Below is a plot showing the fitness as a function of problem size, ranging from four to 20. A list of 20 points was generated randomly, and the first *k* points are used at each experiment.

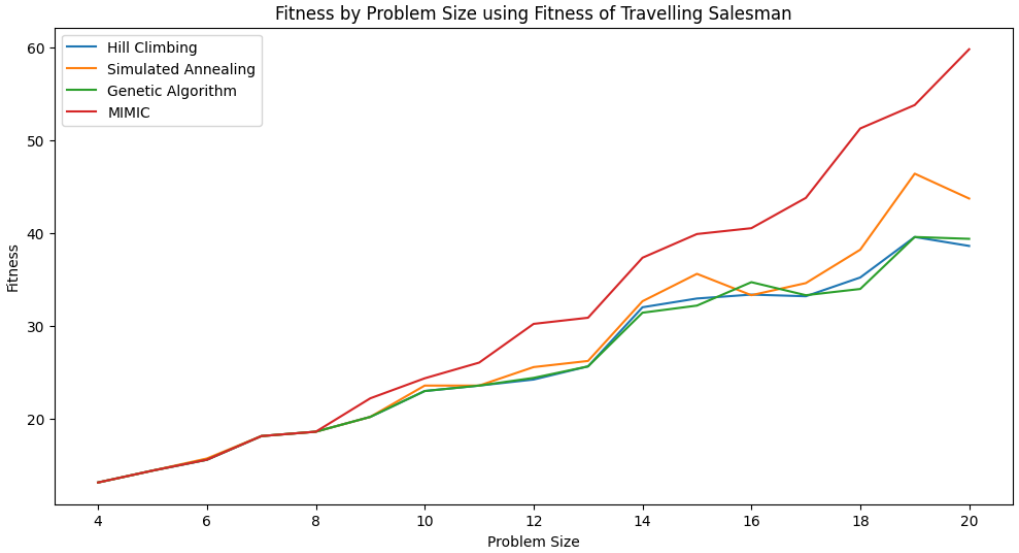


Figure 6. Fitness given by the Travelling Salesman fitness function as a function of input size.

RHC and GA seem to be consistently the best at this task, performing similarly (within 2) at all problem sizes. Note that all algorithms perform similarly until a problem size of 8, at which point MIMIC returns a slightly highly fitness than the other algorithms, as stated previously. On the other hand, GA, which also tracks a population, outperforms MIMIC because of the nature of legal crossovers in the Travelling Salesman problem. Because only k-permutations are legal states, GA is far more constricted in the offspring generation strategy; GA must use some type of random swaps or shuffling between two parents in a way that preserves the legality of the state. This strategy approximates an RHC strategy, where random swaps are made, explaining their similar behavior.

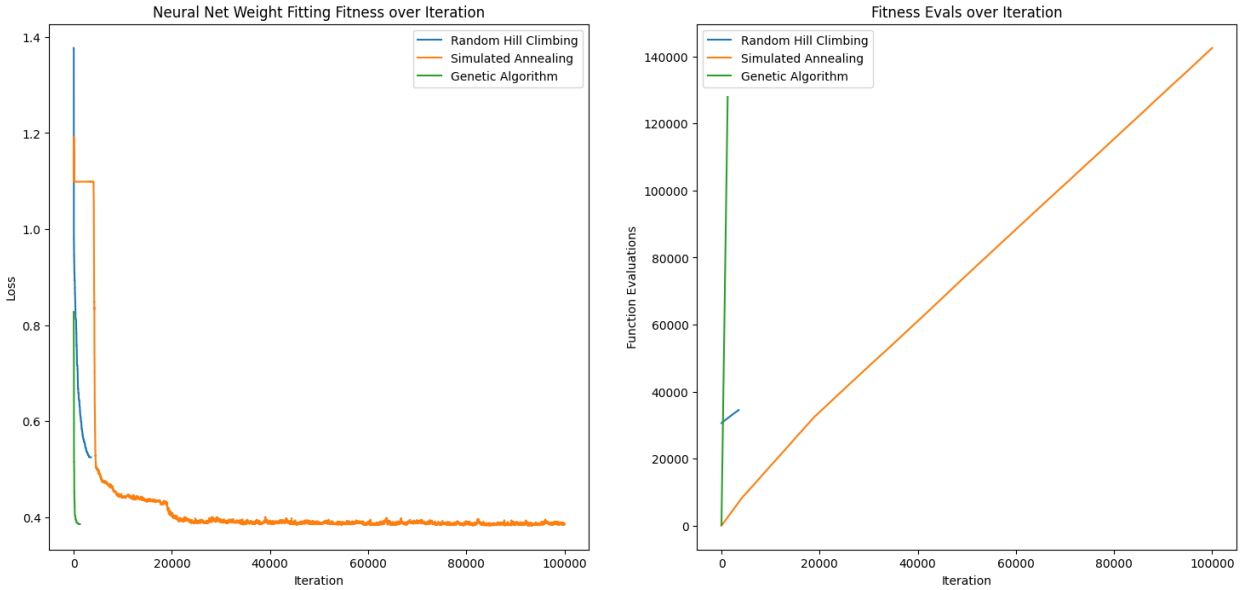
Again, SA seems to perform similarly to RHC and GA until higher problem sizes are reached for reasons discussed in the previous examination. This causes RHC to behave like a slower version of GA, where not many negative jumps are made, limited by either highly unfavorable jumps or low annealing temperature. SA would benefit greatly from both a slower decay schedule and an increase in the number of iterations run. Again, the stochastic nature of SA makes it susceptible to randomness, and more random seeds should be used to run this analysis.

VI. Neural Network Fittings

In assignment one, a neural network was trained over a dataset to produce an accuracy of about 72% using backpropagation. This value will be used as a baseline for the analysis in the section.

The dataset used in this section is the sklearn iris data. This data contains 150 datapoints each with features represented by four floating point value real number and one target classifier. However, the data used in assignment 1 was truncated to its first two features only for visualization purposes. The dataset contains three targets representing three flowers. This dataset is quite a simple one, and was selected to run a comparable analysis with previous results. For consistency and comparison purposes, the same truncated data will be used to train the models in this section.

In this experiment, the 20% of the dataset was taken as a testing set, and the rest was used to train models using RHC, SA, and GA to the standard conditions. The learning rates for each was set to 0.1, and a relu activation function was used. Each model was fitted with a single layer of four hidden nodes. RHC is allowed 10 restarts.



Algorithm	Accuracy
Backpropagation	72%
RHC	80%
SA	80%
GA	83.33%

Figure 7. Learning curve and number of evaluations over iterations. The final accuracy on the test set is also shown.

In the above figure, the loss for each model is plotted as a function of iteration. Also shown is the testing accuracy of each model in a table. Note that all models outperform the backpropagation optimization performed in assignment one. RHC and SA perform equally well, with accuracy scores of 80%, while GA slightly outperforms these, with a score of 83.33%.

For RHC, it seems that the algorithm gets stuck in some type of local minima, as convergence is quickly reached, and any random movement is not accepted. The loss at this point over the training set is 0.5251, while still offering an accuracy on the training set of 80%. Increasing the number of restarts from 10 to 100 seems to decrease the loss slightly to 0.5177 with a training accuracy of 83.33%.

In SA, the loss if much more finely tuned, though at the detriment of about 20 times as many iterations as the others. The result however, is a loss over the training set of 0.3865, much lower than that of RHC, though starting at a loss of about 1.2, the highest of all algorithms. SA also runs far more function evaluations, due to its high number of iterations. The high number of iterations are a result of the annealing properties. Because this problem, unlike previous optimization problems, is a continuous optimization problem and seeks to optimize weights over continuous values, far more possibility of losses can be found, many of which are slightly higher than the current fitness. Therefore, SA is likely to continue running for very long times, as it is cut off by the maximum number of iterations in this problem.

When using GA to determine neural network weights, it again seems to find a near optimal solution quickly, as it does with the discrete optimization problems. Again, it still uses a large population and therefore runs more function evaluations per iteration. The final loss of GA is 0.3857, which is quite good for the number of iterations run. Increasing the population size per generation to 200 samples reduces the loss to 0.3715 and increases the accuracy to 86.67%, both of which are only slight improvements due to the ability to find larger volume in the solution space.

VII. Runtime Considerations

For optimization problems, it is helpful to see the runtimes associated with the size of the problem. The following figure shows the wall clock runtimes of our problems as a function of problem and problem size.

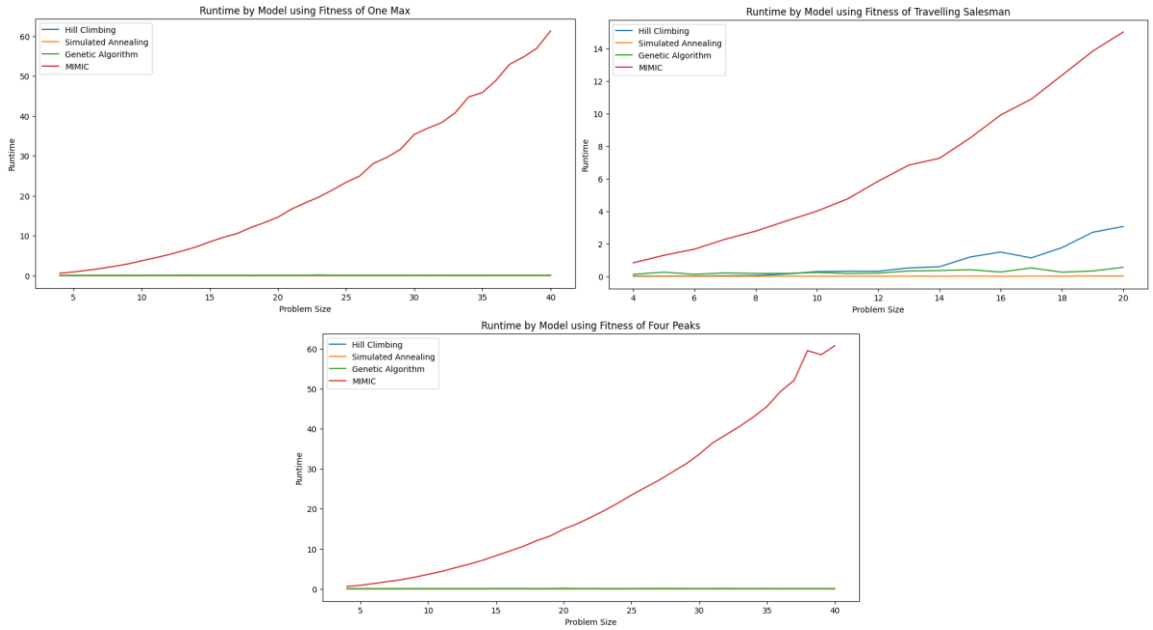


Figure 8. Runtimes by model for all three problems.

For most of the models – RHC, SA, GA – the runtimes seem to be unaffected by problem size, almost negligent in every case. At least for problems of bitstring, all runs seem to not depend on problem size at all. In the Travelling Salesman problem, RHC does seem to increase its runtime by a noticeable margin. Because the number of iterations of RHC is proportional to the number of iterations, this means that RHC runs far more iterations for Travelling Salesman than other problems. This is likely because of the number of permutations available for a factorial-scaled problem like Travelling Salesman. However, because the runtime of SA is limited by an annealing factor that ensures a convergence, its runtime is guaranteed to stay within a given bound. GA also seem relatively unphased by the problem size, as its search time runs more directly with its sampling size than the problem size.

As for MIMIC, it is the only algorithm that is greatly affected by problem size, displaying strong quadratic behavior, more or less. Because it is known that the number of function evaluations is constant per iteration, due to the population size, the limiting factor on the runtime of MIMIC is the construction of probability densities to model the problem and the increased number of iterations. Specifically, MIMIC constructs a dependency tree to characterize a Bayesian network represent an estimated distribution.

VIII. Conclusion

Four different optimization problems were examined under different optimization scenarios in this exploratory analysis. In the One Max optimization scenario, randomized hill climbing excelled, as the lack of local maxima in the problem means that every expected step is the optimal step for finding maxima. Simulated annealing, however, seemed to excel when many local optima detract from the global optima. This is evident in the case of Four Peaks, where simulated annealing almost always finds the optimum state, and only missing if the bonus is not found. Genetic algorithms do quite well on Four Peaks as well, as the crossover implemented preserves good parts of states in Four Peaks, keeping leading ones and trailing zeros together, at the detriment of many function evaluations. Genetic algorithms also perform well in continuous classification problems like that of using neural network, able to take large populations to combine good weights from one sample with good weights from another to produce ideal states. With the Travelling Salesman problem, however, all algorithms seem to have a difficult time, due to the expanded problem space not simply being a discrete optimization problem. MIMIC, especially had a difficult time representing the probability distribution. MIMIC seems to do well in discrete problems that have rulesets and bonuses that are unclear and difficult to find, as shown by its ability to find the bonus in every run of Four Peaks with respect to problem size. In future iterations, tuning of hyperparameters and more random runs can produce more results about the algorithms. Nonetheless, it is clear that each algorithm has use cases that highlight each one's utility.