William Xia
Wxia33
CS4641 Machine Learning

Randomized Optimization

Abstract

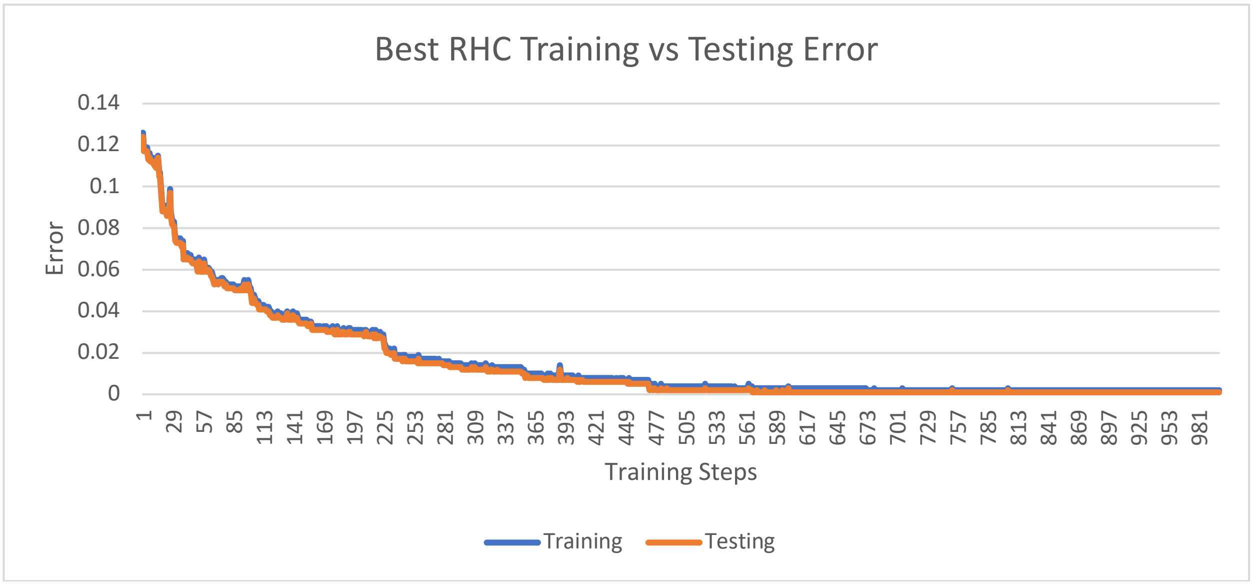
This assignment explores various randomized optimization algorithms by comparing and contrasting their properties while optimizing the weights of a neural network training on the UCI Wine Quality dataset. The performance of Randomized Hill Climbing, Genetic Algorithm, and Simulated Annealing on the task of optimizing the weights for a neural network are compared. Next, MIMIC is included with the other three randomized optimization algorithms and used to optimize solutions for Traveling Salesman, Knapsack, and Flip flop problems.

Background

From the experiments run on the previous Supervised Learning project on the wine dataset, the optimal neural network layers were determined to be 6. The data were split into 70-30 ratio to train and test on. Each algorithm tested and compared different hyperparameters. For hyperparameter choice, we are ideally looking for a smooth curve that reaches a bias point relatively quickly. The smoothness is desired since we would like an algorithm to display low variance across different problem types. Fast convergence is also desired, since such a property indicates the parameter allows the algorithm to learn good solutions quickly.

Randomized Hill Climbing

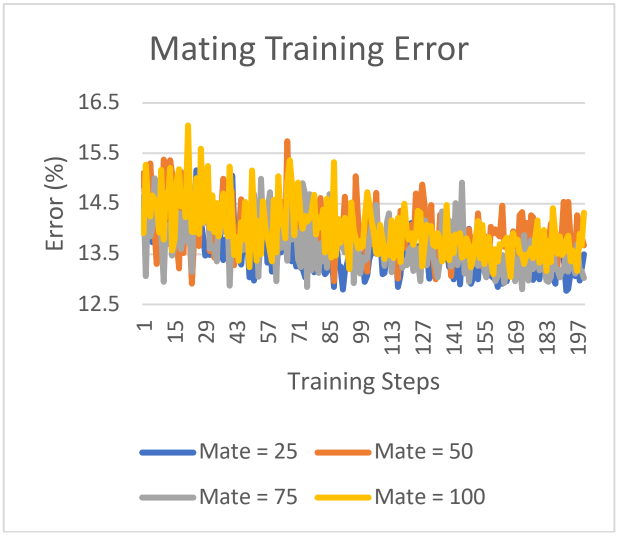
Randomized Hill Climbing randomly chooses points and finds the local maxima. Since the only hyperparameter is the number of times the algorithm is run, the best fit graph was chosen from the most optimally fitted run.

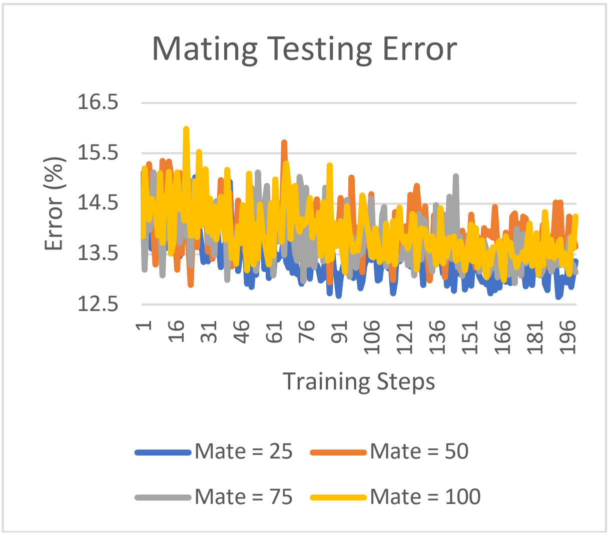


The above chart matches how the algorithm is expected to perform. The algorithm initially starts on a random point in the hypothesis space, and then iteratively finds the maxima point. The beginning of the graph is quite erratic, hinting at the random nature of the algorithm process, but as the algorithm proceeds to find the maxima, the error rate slowly decreases, and the performance becomes more predictable. The algorithm seems to converge to a low bias value, since the algorithm performs similarly to gradient descent, albeit with some randomness.

Genetic Algorithm

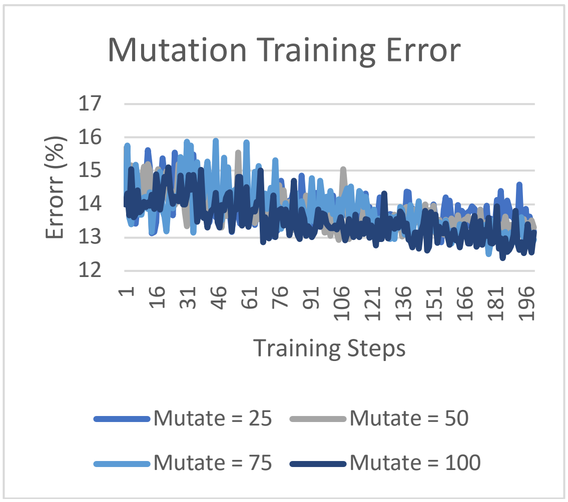
Genetic algorithm is based off of the theory of evolution; a population of configurations is generated, among which the best configurations are chosen to move and cross breed with each other, and the process repeated. For genetic algorithm we have two hyperparameters: mating and mutation. The mating parameter controls how many members of the population mate with other members, such parameter attempts to simulate the process of reproduction in biology, with the idea being that more fit individuals will pass on genes to other fit individuals and eventually converge on children which perform even better than the individuals did. The mutation parameter controls how much the population mutates, or spontaneously changes, for each training iteration. This parameter is intended to help individuals avoid local optima, in which random changes are injected into the individual in hopes of improving the overall fitness of the population.

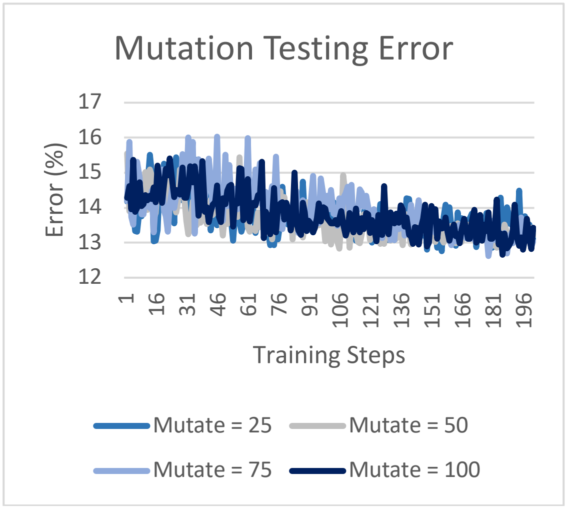




A consistent behavior across all parameters is that the variance of the error reduces as the number of training steps increases. This behavior may be caused by the converging nature of the algorithm; since as time increases the probability of converging on an optimal solution increases, the more variation in the population the more likely the genetic algorithm will get close to the optimal weights in the neural network. Hence, more of the population will be similar to each other and variance will be reduced. However, it seems that the lowest mate parameter, 25, performs the best for this problem. While high mating may help the algorithm produce individuals which are "on the right track" for the optimal solution, so to speak, the high mating also reduces the chance of the individual keeping the same traits that may allow it to perform better in the future. A balance towards the lower end of mating is therefore preferred for this problem.

Final Hyperparameter Choice: Mate = 25





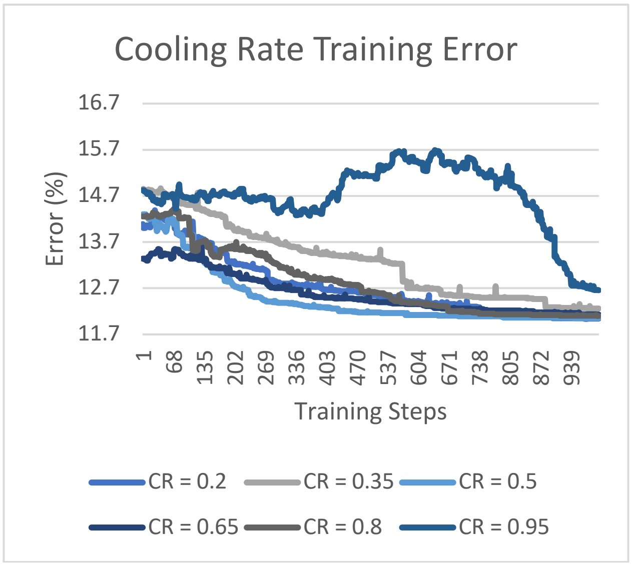
Similar behavior can be seen as with the mating graph, in that the variance decreases as training steps increases, albeit more slowly. High mutation parameter allows the algorithm to randomize the weights much more frequently, but since the population is culled based off of the performance, the general trend of the population's error will decrease. Mutations can only change the weights so much, and with the evolutionary pressure to decrease error, the algorithm produces the curve above. High variation in the individual itself, also known as mutation, allows the individual to explore ways to optimize the performance, given that the individual already performs well enough in the population. A high mutation parameter is essentially a randomized version of depth first search, since the individual explores its own hypothesis space much more. For this problem a higher mutation produces consistent and good performance across the training steps.

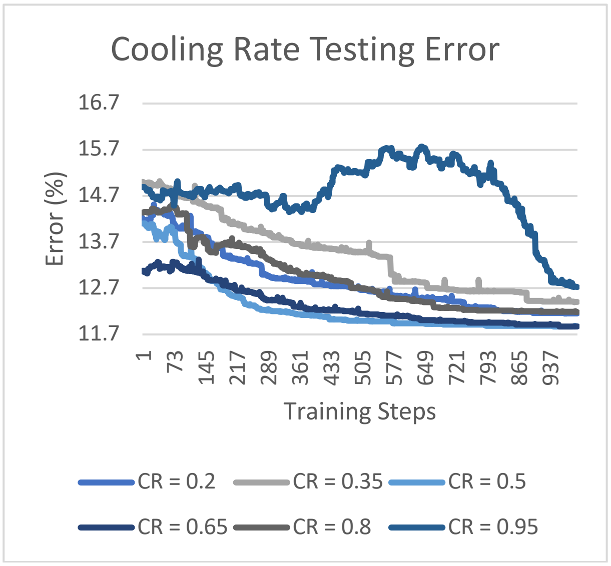
Final Hyperparameter Choice: Mutate = 100

Simulated Annealing

Simulated Annealing is based off of metallurgy; a high temperature is chosen which creates a high variance in the configuration space, and the temperature is slowly reduced which causes the algorithm to converge on a single maximum value. The initially high variance causes the algorithm to avoid local maxima and eventually converge on locations which are generally the global maxima of a hypothesis space. For simulated annealing there are two major hyperparameters to control: the cooling rate and temperature. The cooling rate of the algorithm controls how fast the algorithm decreases the temperature, much like how real metallurgists control the cooling rate to give metals different properties, the cooling rate affects the weights the algorithm converges on. The temperature parameter controls the initial starting temperature. As we will see with the graphs below, the random nature of simulated annealing means that, in the context of this problem the starting temperature will not matter, since high enough temperatures are essentially "random enough".

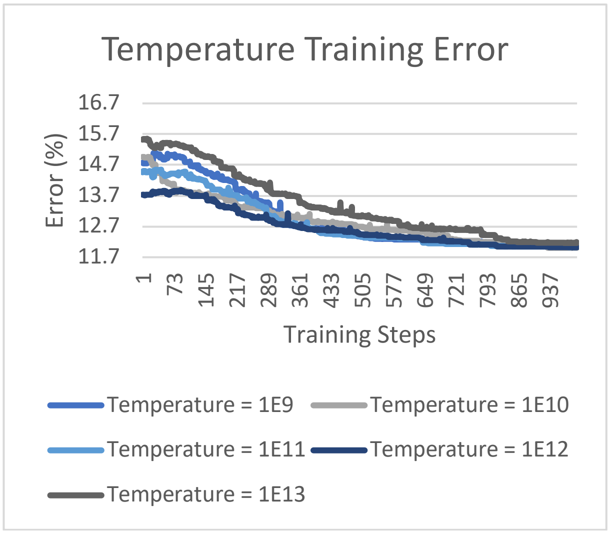
William Xia
Wxia33
CS4641 Machine Learning

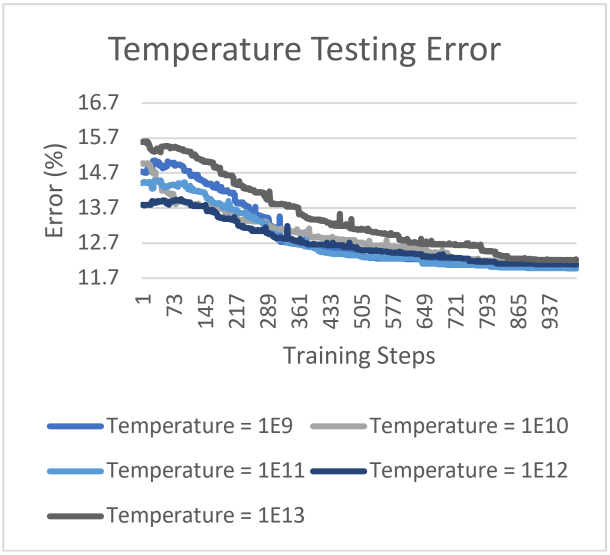




Generally, the algorithm produces error which decreases over time, with the exception of CR = 0.95 which produces a curve at around training step 400. Since higher cooling rates cool slower, the greater variance of the algorithm causes the curve to reach points which may potentially be worse than the iteration before, but eventually, all cooling rates converge to a low error rate. We see that a faster cooling rate causes the algorithm to converge much more quickly to a better solution. The algorithm with the lower cooling rate starts off with a high bias, but as the training steps increase there exists little variance compared to the other cooling rates. While the other cooling parameters do converge to about the same error rates, the performance is highly variable across the training steps. Thus, for this problem lower cooling rates are preferred.

Final Hyperparameter Choice: Cooling Rate = 0.2





William Xia
Wxia33
CS4641 Machine Learning

Temperature seems more consistent compared to cooling rate. The performance of simulated annealing is more or less the same across all temperatures. Since the idea of the algorithm is to start from a higher temperature and decrease to a lower temperature, as long as the temperature is high enough then the curves for all the graphs will be similar. Since the cooling rate for these sets of algorithms is held constant throughout the different runs, the variance is not significantly different from each other. The bias of each parameter starts out high, but over time converges to a stable value. Despite the similar performance, the temperature 1E12 seems to perform consistently the best, and so will be chosen as the hyperparameter of choice.

Final Hyperparameter Choice = 1E12

The two graphs above demonstrate that the only parameter that significantly influences the algorithm is the cooling rate.

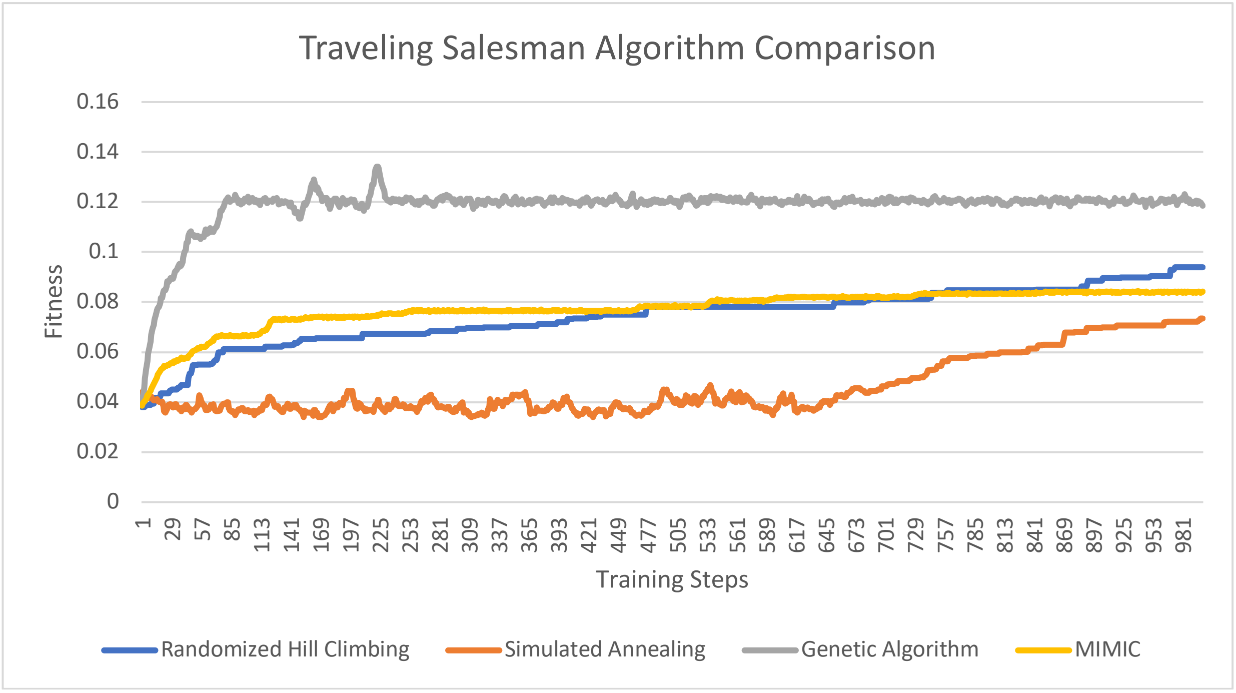
William Xia
Wxia33
CS4641 Machine Learning

Interesting Problems

Randomized Hill Climbing, Genetic Algorithm, and Simulated Annealing will be used to optimize solutions for three different interesting problems. A fourth algorithm, MIMIC, will be included for analysis as well. All the algorithms used the same experimentally determined parameters as described above.

Traveling Salesman

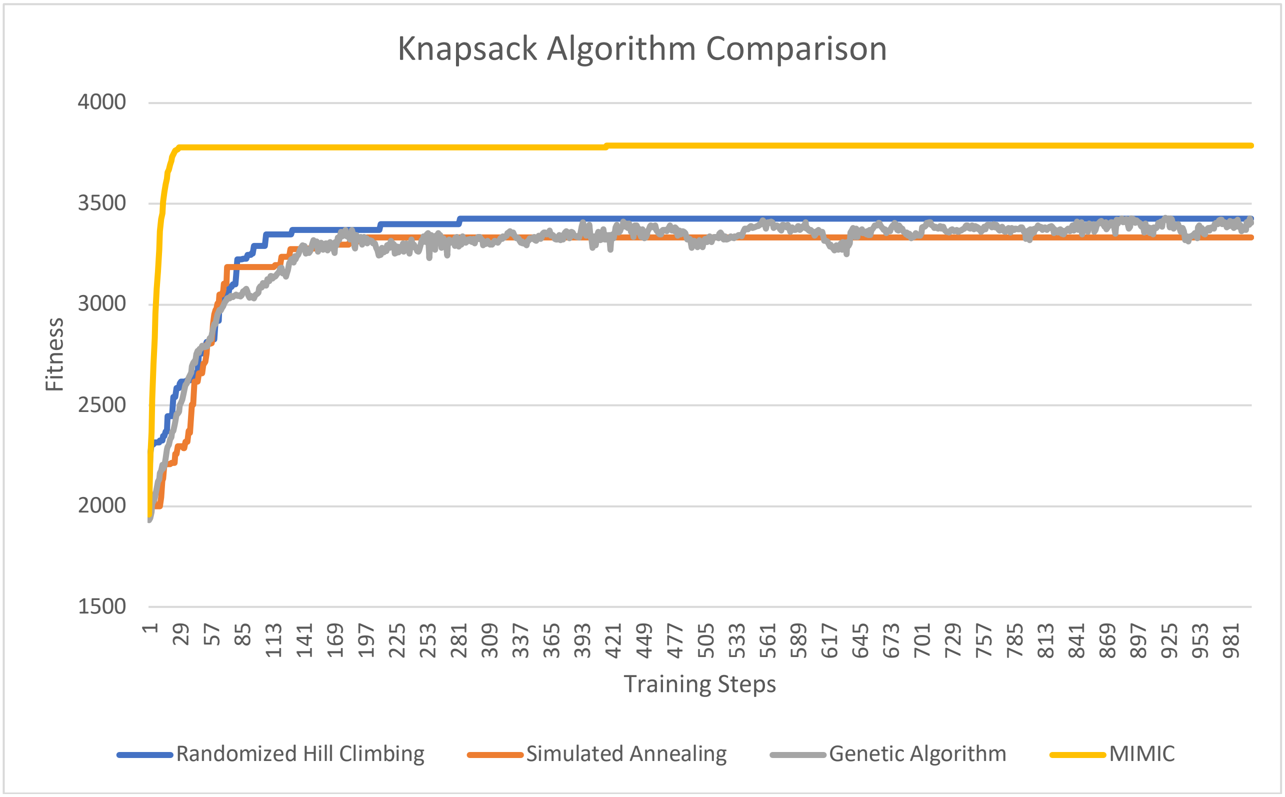
The Traveling Salesman is an NP-hard problem that has been studied extensively. The problem asks one to find the shortest possible path that visits all nodes in a graph and returns to the original starting node.



Here, we see that genetic algorithms outperform all the other algorithms significantly. Randomized Hill Climbing and MIMIC both produce similar results, while simulated annealing is the worst out of all the algorithms. The traveling salesman problem seems to benefit from a breadth-first search approach, in that searching many solutions in a shallow manner rather than one solution in a deep manner yields the best results. Once a path is deemed short enough for a population, then more of the population will derive similar solutions from the initial candidate path. However, it seems there is a plateau in the solution, since the fitness exceeds 0.12 at some points in the graph, but the trend remains the same throughout. This indicates that there still exists some bias in the problem, as once a genetic algorithm finds a good enough solution, it is difficult to exceed such a solution using genetic algorithm. MIMIC and Randomized Hill Climbing use similar methodologies, in that both algorithms begin with a uniform assumption of the distribution of hypothesis, and iteratively updates the solution as training increases. Randomized Hill Climbing performs slightly better in the end, as the algorithm tends to move towards maxima points in general. Simulated annealing does not perform nearly as well, since the algorithm randomly searches the traveling salesman problem space, which is quite massive. Simulated annealing does eventually find a way to increase fitness but compared to the other algorithms simulated annealing performs poorly due to the slow convergence, but the convergence was quite fast in terms of real-time, as simulated annealing performed up to 100x faster than the other algorithms.

Knapsack

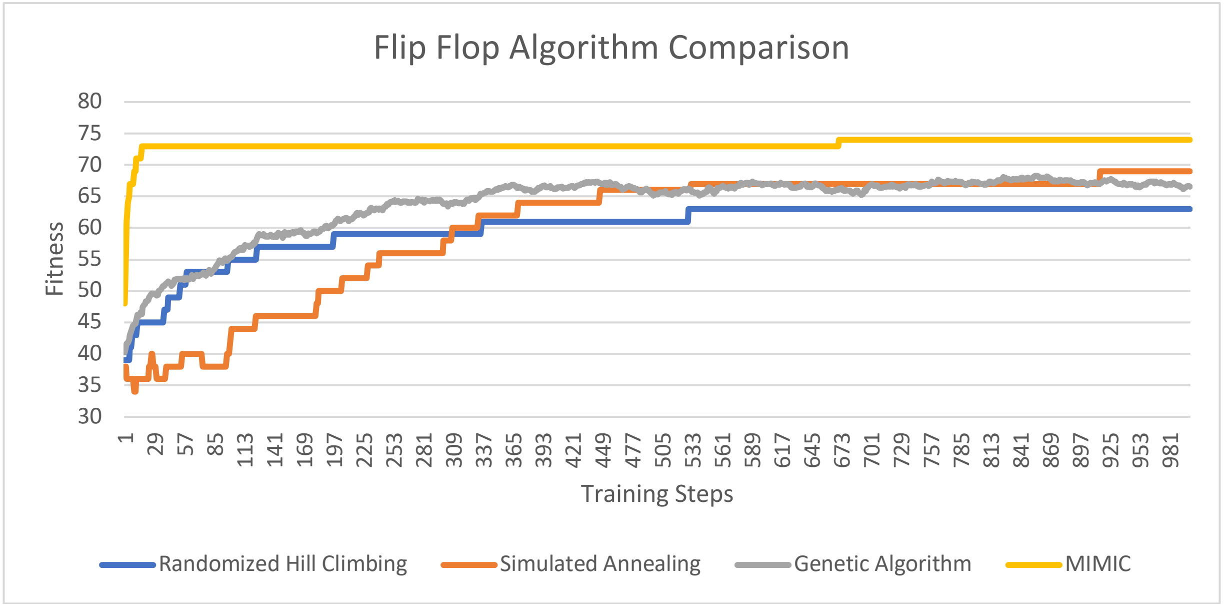
Knapsack is another classic problem that is usually introduced to people learning dynamic programming. The problem is: given a set of items with associated weight and value, what is the maximum value of a subset of items that weighs less than or equal to a specified limit?



Here, we see that MIMIC performs the best out of all the other algorithms. Randomized Hill Climbing, Simulated Annealing, and Genetic Algorithms produce more or less the same curves and fitness, with genetic algorithm displaying notably more variance. While MIMIC produces the best results quickly, the run time is quite significant, sometimes taking up to 100x as long as simulated annealing, introducing a significant cost constraint when one is trying to find an optimal solution to this problem. The problem is unique in that different objects have different effects on the dependencies of the solution, and so probabilistic algorithms such as MIMIC are well-suited for this type of problem, since it accounts for how different objects interact with each other. The other three algorithms managed to find a solution to converge upon, but since the algorithms do not take into account how one item may be more optimal to include than another, there exists some bias as evident in the graph. Genetic Algorithm is notable here in that there is a wide variance of values notable as the training steps increase. The mating and mutating of the population produces random changes even when the solution is close to the bias plateau; but similar to what occurred in travelling salesman, the algorithm does not improve further, since optimal solutions are not intelligently explored further.

Flip Flop

The flip flop problem asks how many times bits have been flipped within one string.



MIMIC performs the best out of all the algorithms, with randomized hill climbing and genetic algorithms producing similar performance, and simulated annealing starting out with bad results but slowly converging to be second best. This problem counts the number of bits which have been flipped, which seems to have a degree of statistical dependency on other bits. MIMIC is able to quickly determine such dependencies within few training steps, but again as with the other problems MIMIC performs the slowest regarding real-time duration. Simulated annealing happens to be the fastest in real-time but converges the slowest for each iteration. Early iterations have higher temperatures, which causes the fitness to be generally low for most runs, but as the temperature is cooled down, the fitness increases rapidly, as optimal solutions begin to appear. Randomized Hill Climbing and Genetic Algorithm are both in the middle of the four algorithms in terms of real-time performance, with genetic algorithms being the slower of the two, but produce consistent results across all training steps. The random nature of genetic algorithms causes some fluctuation in the final plateau of fitness, while randomized hill climbing converges on the error value in a predictable manner.

William Xia
Wxia33
CS4641 Machine Learning

Conclusion

Randomized optimization provides another way to optimize a problem space, given that one does not know an optimal approach to solving a problem. As shown with the above graphs, different algorithms and different hyperparameters can produce radically different results for each respective problem. Genetic Algorithms display breadth-first search like properties, in that many different candidates are selected across the hypothesis space. Simulated annealing generally performs worse for the first few iterations, but eventually converges on a similar fitness value as the other 3 algorithms. Simulated Annealing also performed the fastest across nearly all problem spaces, so Simulated Annealing is preferred when efficiency is a concern. Randomized Hill climbing produces results much like gradient descent, since the algorithm is essentially gradient descent but with random restarts. MIMIC shows its strength in problems that contain probabilistic dependencies but is costly to run and is not always suitable for certain problem types. Thus MIMIC should only be used when one has knowledge of a problem space and has the available resources to run the algorithm.