CS 7641 Assignment 2

Introduction

This assignment explores four different optimization algorithms and their efficacy at different types of problems. The purpose of the experiments is to see which algorithms performs best at which types of problems, and to confirm the implications of the No Free Lunch theorem, that there isn't a general purpose algorithm that is best for all types of problems. Three toy problems were tested using the ABAGAIL library using three of the four algorithms. Then, all four algorithms were tested on the optimal Neural Network structure resulting from a dataset in Assignment 1.

Algorithms

Randomized Hill-Climbing (RHC)

Randomized Hill-Climbing is a straightforward optimization method where a random starting point is picked at start, then its neighbors are evaluated to see if any improves the fitness function. If any of the neighbors have better fitness than the current point, that neighbor then becomes the current point. This process is repeated until the current point has better fitness than all of its neighbors, at which point the search is restarted with another random point. Restarts keep occurring until the search budget (i.e. total number of iterations) is spent. In terms of exploration vs. exploitation, this algorithm heavily leans toward exploitation since it always looks for an improvement. There were no special parameters that were used for the implementation of this algorithm.

Simulated Annealing (SA)

Simulated Annealing is a optimization algorithm that is inspired from metallurgy where metals were repeated heated and cooled, allowing its molecules to fall into their optimal structure. The algorithm has a set temperature schedule and the tradeoff of exploration vs. exploitation is dependent on the temperature. The jump from one point to another is done probabilistically based on the difference in fitness as well as the temperature. At the beginning of the algorithm, the temperature is set to high, allowing points to jump to other points even if their fitness is worse, resembling a random walk. As the temperature decreases, the algorithm is less inclined to jump to points with worse fitness and is instead doing more exploitation. As the temperature approaches zero, the algorithm is essentially doing hill climbing until it reaches the final peak. The two main parameters tested for this algorithm were the initial temperature and the temperature decay rate (a multiplicative factor between 0 and 1).

Genetic Algorithm (GA)

Genetic Algorithm is inspired from evolutionary biology and taking many inspirations from its mechanics. It works by having a starting population and then evolving that population by only selecting the ones with some percentile fitness, then breeding them via crossover, and mutating them. Exploration is done via the crossover mechanism and exploitation is generally done at the mutation step in order to increase the fitness of the instance. Every iteration, a new generation of the population is created and would have inherited many of the "traits" from the best performing instances of the

previous generation. The GA implementation takes in three main parameters: the sample population size, the number of instances to mate, and the number of instances to mutate (at each iteration). This is an interesting algorithm because it is different from the hill climbing based algorithms explained above and is an entirely different approach to randomized optimization.

Mutual-Information-Maximizing Input Clustering (MIMIC)

MIMIC is an algorithm that tries to capture the structure behind the problem using a candidate distribution at each iteration. It starts by randomly selecting a population, then filtering the population for some percentile. The remaining points are then used to construct a candidate distribution. This is done is by representing the structure of the distribution as a Dependency Tree, which is the simplest structure that still captures dependencies between different features. The Kullback–Leibler (KL) Divergence between the candidate distribution and the sample distribution is then minimized, which translates to maximizing the Mutual Information between each node in the Dependency Tree and its parent. This is solved using a Minimum Spanning Tree (MST) algorithm (such as Prim's) with the weights in the tree negated, producing a Maximum Spanning Tree. On the next iteration of the algorithm, points are sampled from the new candidate distribution and the process is repeated. This algorithm spends more time per iteration than other algorithms mentioned since it needs to create the candidate distribution but benefits by having more of the problem structure captured on each iteration. This algorithm represents a different class of algorithms that tries to represent structure and not just the current point and its fitness. The main parameters for this algorithm's implementation are the total sample size, the sample size kept, and the smoothing coefficient m, also called equivalent sample size or pseudocount.

Problem Results

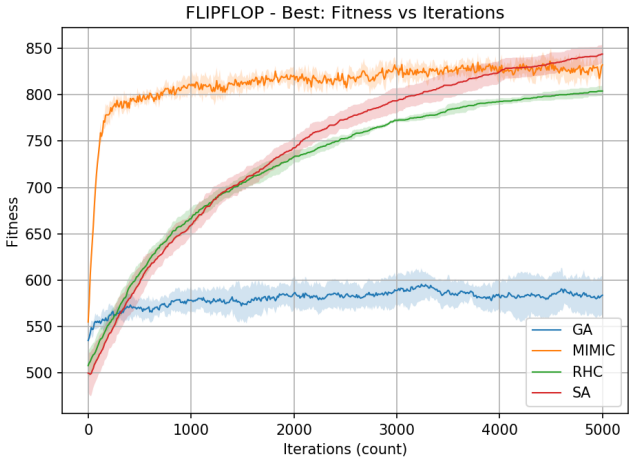
Three toy problems were tested with the four optimization algorithms mentioned previously. Those problems are Flip Flop, Continuous Peaks, and Traveling Salesman. For each problem, 5 iterations were run using each of the four algorithms to get a sense of variance for their performance. Flip Flop and Continuous Peaks were run for 5,000 iterations each whereas Traveling Salesman was only run for 3,000 iterations due to time constraints.

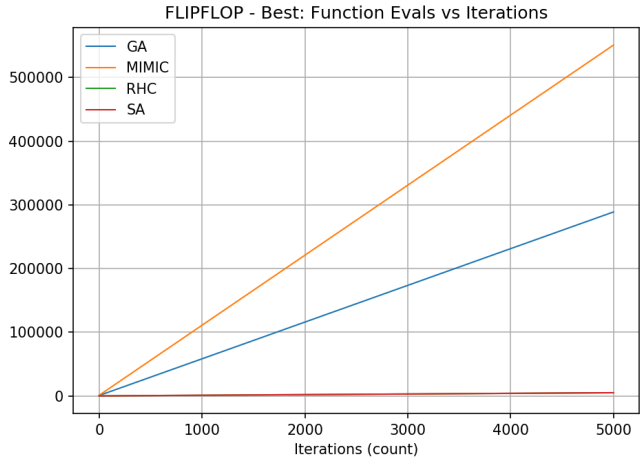
Flip Flop

Flip Flop is a simple problem that was mentioned in the lectures. The goal of the problem is to maximize the number of alternations in a bit sequence, i.e. to maximize the number of times a `0` bit is adjacent to a `1` bit. This problem was run for 5,000 iterations for the three algorithms tested and used a 1,000 length bit string. This is a particularly interesting problem for contrasting the different algorithms because it has fairly simple structure, two global optimums that are reachable from any point, and has no local optimums. This will showcase the strength of greedy / exploitation-based algorithms, as well as algorithms that can sufficiently capture its structure.

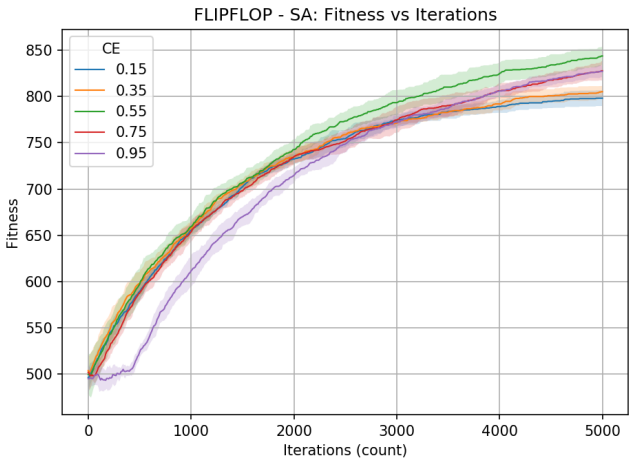
The fitness curve for the problem is shown on the bottom left. MIMIC starts off strong as it's able to learn the structure of the problem very quickly but eventually plateaus as perhaps the method of generating candidate distributions missed certain parts of the sequence in its representation. GA performs consistently poorly as I believe the uniform crossover mechanism is not suited for this type of problem as it randomly assigns each bit to one of the two parents, not knowing the relationship between each bit and the adjacent ones matter. RHC and SA both start off very poor but improves gradually until they surpass (or will surpass) MIMIC. The reason for this is because greedy search is quite suited for this problem; you can get to the global optimum from any point just by continuously searching neighbors. For that reason, both will continue to improve and eventually reaching global optimum. The Function Eval chart on the bottom right shows that, as expected, MIMIC has the highest number of function evaluations per iteration, followed by GA, since both must maintain a population of points instead of having one current one like RHC and SA. This is further reinforced in the total run duration (5,000 iterations) for each algorithm: GA ~0.9 seconds, MIMIC ~2,500 seconds, RHC ~0.02

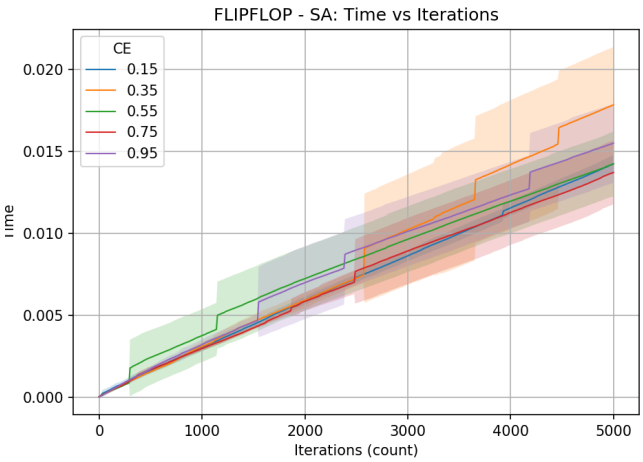
seconds, SA ~0.015 seconds. MIMIC is much slower than GA because, in addition to maintaining a population, it also needs to run the Mutual Information maximization algorithm and construct the Dependency Tree.



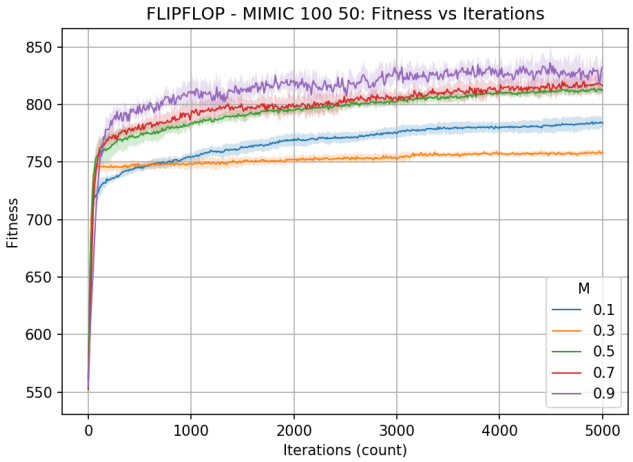


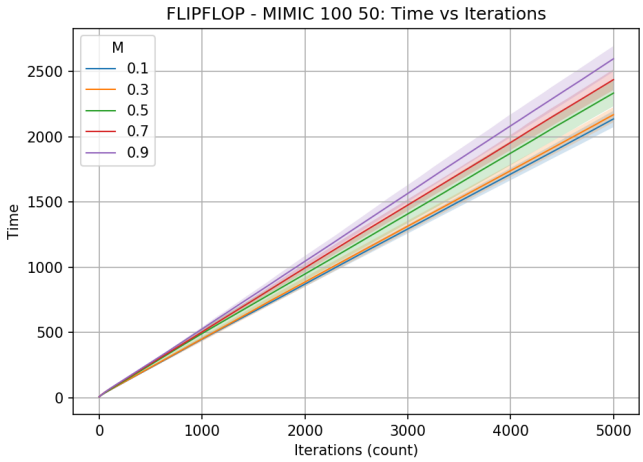
Now we take a closer look at the SA algorithm and how its performance changes with the different parameters. The two main parameters with initial temperature and the cooling rate. The initial temperature did not make a significant impact on the overall performance, given that it is sufficiently high, so I'm only showing results for initial temperature of 10^{10}. The cooling exponent (CE) parameter is the multiplier that decreases the temperature on each iteration. Both the performance and the timing was not very sensitive to the cooling parameter. One interesting observation is that the least aggressive cooling schedule (CE = 0.95) starts off with very poor fitness. This is likely because with a slower cooling schedule, the algorithm spends more iterations at the beginning exploring rather than exploiting / climbing.





Now we take a closer look at the MIMIC algorithm and how its performance changes with the different parameters. The population size and samples generated were set at 100 and 50 respectively, which were the largest parameters that would run in a reasonable amount of time. With a larger population and more samples, the algorithm generally performed better as it was able to construct better candidate distributions. The third parameter tested was the m equivalent sample size, which is a smoothing factor that helps mitigate zeros or near-zeros in the estimation of Bayesian probabilities. This helps the numerical and sampling stability of the candidate distribution. The charts below show that with more smoothing, the algorithm performs better. This perhaps indicates that the population size / to-keep size is not adequate for numerical stability. With more time, I would increase the population and keep sizes and re-run the same experiment to see its effects. In terms of calculation time, there is a slight increase as the smoothing coefficient increases but for the most part it is quite negligible: about 100 seconds increase for each 0.2 increase in the parameter.

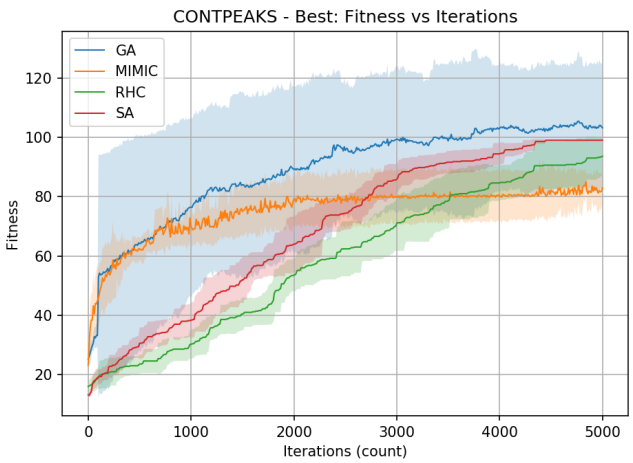


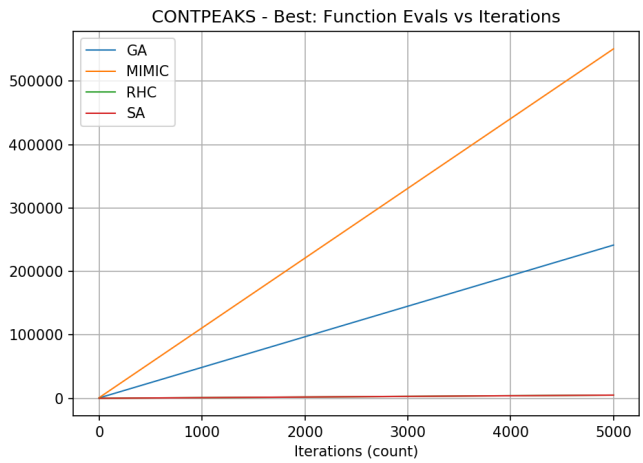


Continuous Peaks

The Continuous Peaks problem is an extension of the 4-peak and 6-peak problems, where there are numerous non-global optimums and the algorithm must find the one global optimum. This problem is interesting because there are so many local optimums and it will test the algorithm's ability to get out of those effectively and search through the space in a effective way.

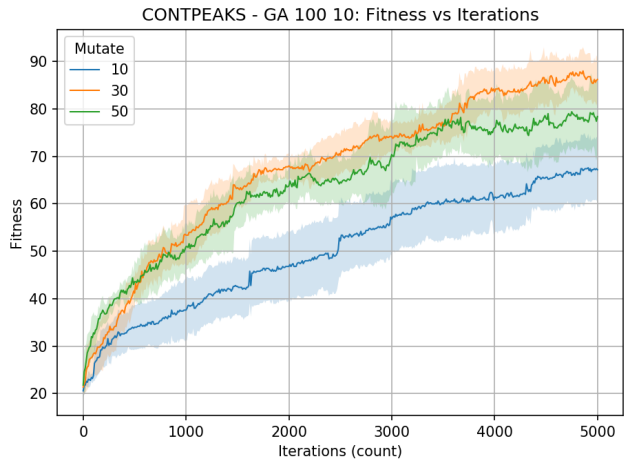
In terms of overall performance, GA was able to find the peaks with the highest fitness although there is a large range, indicating the parameters of the algorithm have a large impact on its performance. MIMIC learned extremely quickly but then began to plateau with very little improvement. This is likely because at some point the sampling distribution no longer includes basins of higher peaks, so it only produces points from a few local peaks or even just one local peak. RHA and SA are understandably slow since they don't retain any information on the structure of the problem, nonetheless, they continue to grow and show little signs of plateauing. This is because they still have the ability to explore after many iterations, and are not constrained to some sampling distribution / prior belief. RHC has the most consistent gains and show the least sign of plateauing because it's sampling uniformly after each local optimum and eventually it will hit every optimum, including the global one. In terms of number of evaluations per iteration, it's similar to the flip flop problem.

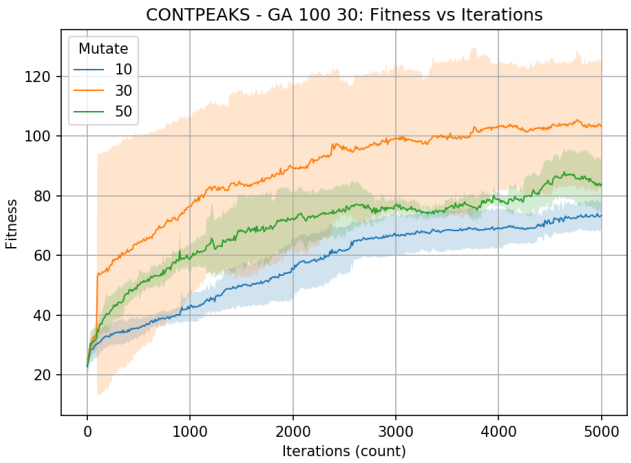


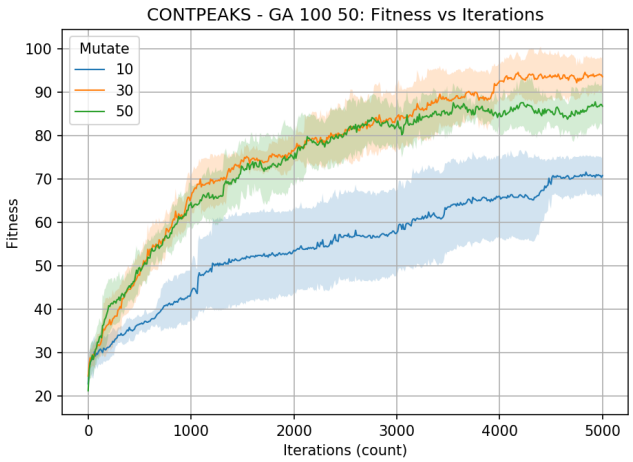


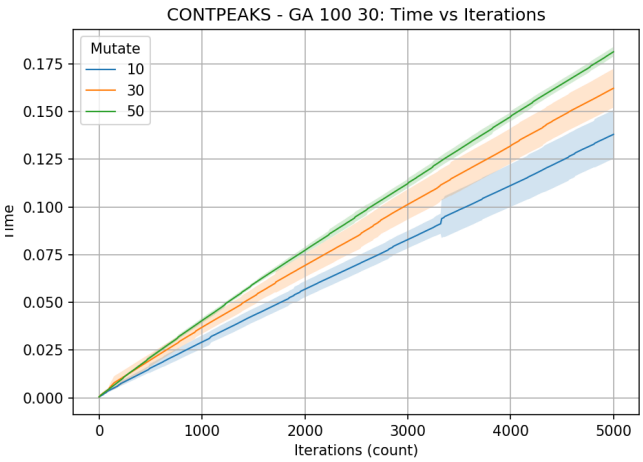
Taking a closer look at the GA performance, we can see the influence of the different parameters. The population size is the number of instances that the algorithm starts with and ends with at each iteration. The number of instances to mate is how much of the population undergoes crossover to produce new instances. The mutate parameter specifies how many of the instances undergoes mutation, or exploitation in the general optimization context. The population was set at 100 since that was the largest population that could be run in a reasonable amount of time. From the fitness curves below, we see that the middle of the range for both mutate and mate parameters (30) produced the best results. This is interesting as

one would usually expect the more mating and more mutating the better the performance. This result may be because with too much mating and mutating, some of the structural information from prior generations may be lost on every iteration; about 1/3 of the population would be a good balance between generating new instances and remembering patterns from old instances. Notice that the variance is also incredibly high for the best setup, indicating it's not the most reliable setup (i.e. does not always have good candidates in its population) but on average it produces the best results. The fourth chart below shows the time it takes per GA iteration for the mate parameter of 30. As expected, the time per iteration is higher with more mutations and is able a constant time increase.

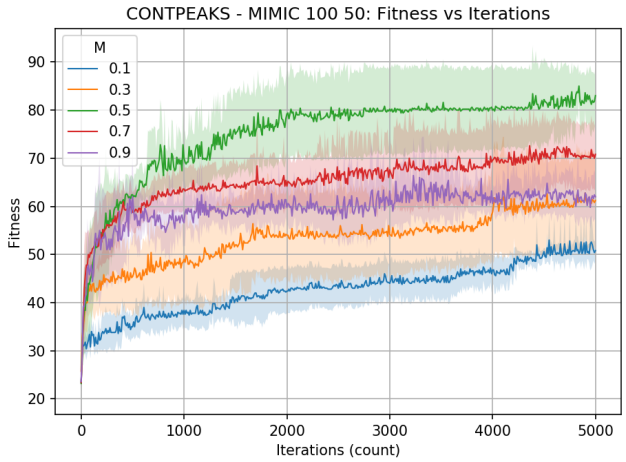


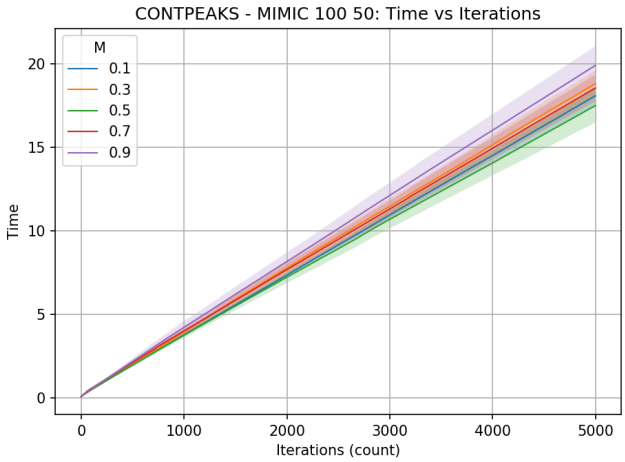






Now we take a closer look at the effects of different parameters on the MIMIC algorithm. The smoothing parameter behave differently than it did in the Flip Flop problem: it's no longer the case that the more smoothing the better. The optimal smoothing is about a 0.5 pseudocount, exactly in the middle. The reason for this is that there is more structure to capture in the dependency tree and so the signal from each estimator is fairly important and could be drowned out if the smoothing factor is too large. In the future, it would be interesting to re-run this experiment with a more complex graph structure than a Dependency Tree to see if it can perform better on the Continuous Peaks problem. In terms of timing, we see the same phenomenon as in the Flip Flop problem: more mutations taking longer but negligibly longer.

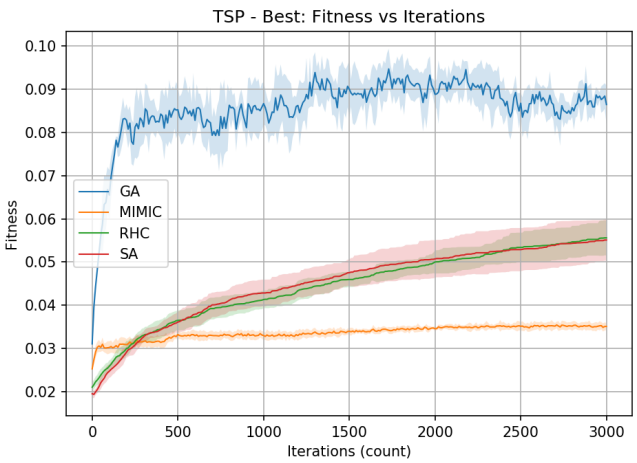


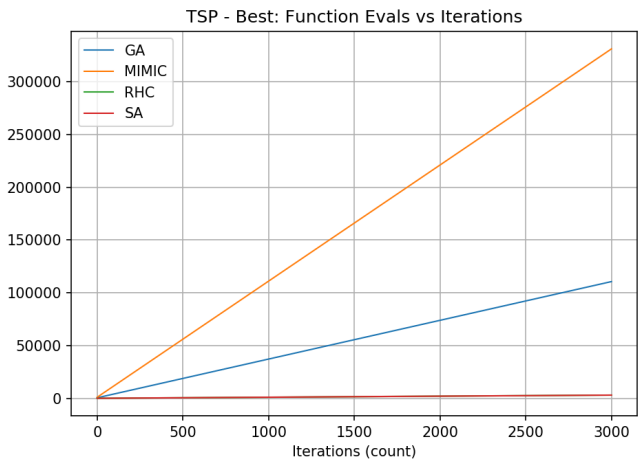


Traveling Salesman

The Traveling Salesman problem is a classic combinatorial optimization problem that is known to be unsolvable in polynomial time. The problem statement is, given a set of points and weights edges between those points, what is the fastest route that visits each point and returns to the original point. This will be an interesting challenge as there is definitely structure in the problem but not easily modelled. Also, the fitness function for this problem is extremely complex as it's based on a densely connected graph.

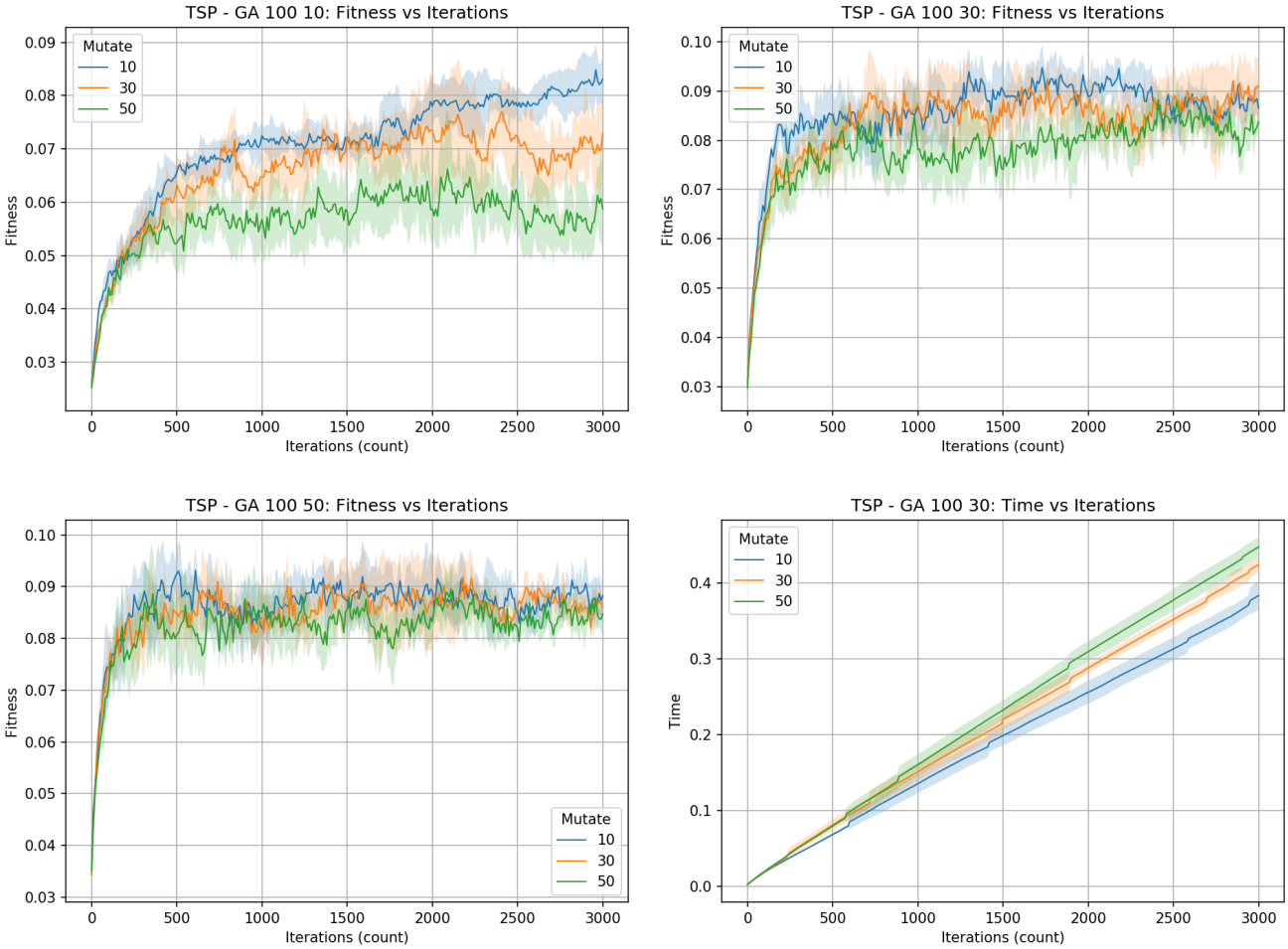
Overall, GA proves to be by far the best algorithm for this problem, and MIMIC the worst. Intuitively, GA's stellar performance makes sense as the crossover mechanism will essentially combine together good routes on one part of the graph with good parts of another part of the graph. RHC and SA has a slow and steady performance as always. MIMIC does surprisingly poorly on this problem. This is likely because the structure of problem, being a dense graph, is difficult for the MIMIC dependency tree algorithm to capture properly, giving a poor approximation of the true distribution. In the future, it would be interesting to use a different representation for the structure, such as a more dense connected graph. This will take much longer for the algorithm to run but it would likely capture more of the characteristics of the problem on each iteration.





Taking a closer look at the GA performance, we can see the influence of the different parameters. For this problem, the maximum mating size of 50 seemed to have performed the best (vs. the 30 in the Flip Flop problem). This is likely because with a more complex structure / fitness function, a higher mating population would be required to ensure at least some decently improving candidates are in the next generation. However, the mutating population parameter does not seem to have a large impact on the results. One explanation for this is that the crossover mechanism is the main

driver for fitness improvement in this particular problem. In terms of timing, more mutations generally more time, as shown in bottom right chart, but not significantly more time.



Neural Network Analysis

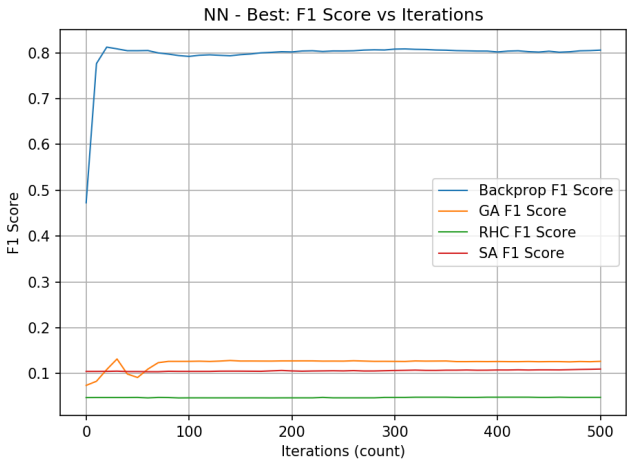
This part of the report analyzes the performance of three of the four algorithms (RHC, SA, and GA) on finding optimal weights for a Neural Network. This poses an interesting problem as there is a well defined but complex structure in a fully connected network. The performance of these algorithms will be benchmarked against the standard Neural Network learning algorithm: Gradient Descent with Backpropagation.

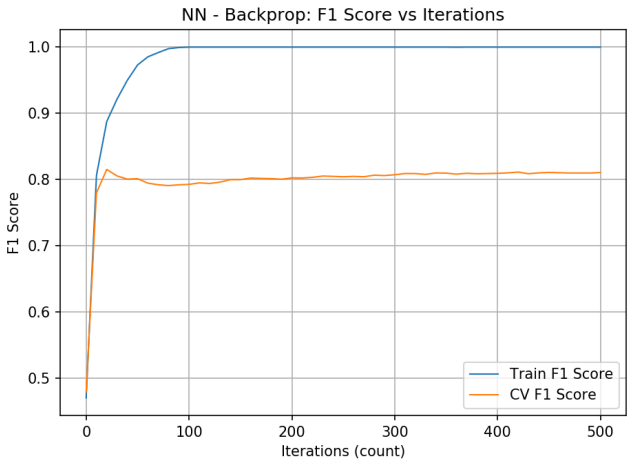
Data and Network Architecture

The dataset I've chosen is from Assignment 1, and is the Fashion pictures dataset. This dataset contains 10,000 pictures of size 28 by 28, and are labelled zero to ten, which corresponds to the type of clothing that is in the picture. I picked this dataset as Neural Networks performed quite well on this dataset and could sufficiently learn the underlying mapping. To assess the results, I used the average F1 Score across all 10 labels of the dataset. This metric is well suited for this problem as it is the harmonic mean of both precision and recall, making it fairly holistic. The setup for training and testing is similar to Assignment 1: 20% of the total dataset is used for testing, and then 20% of the remaining data is held for validation, both stratified by label. The network architecture is taken as the best performing Neural Network setup from Assignment 1. This is a fully connected network with 784 input nodes (pixels), one hidden layer of 784 nodes, and an output layer of 10 nodes using ReLU and Softmax activation functions. This is a fairly complex network with more than 600,000 parameters so it will be a challenge for the optimization algorithms to find decent solutions.

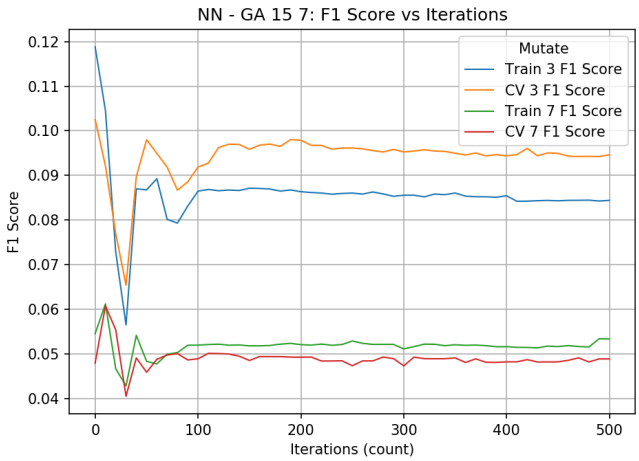
Performance Results

Unsurprisingly, backpropagation is the undisputedly the best algorithm for optimizing Neural Network weights, with none of the other algorithms even coming close to its performance. In fact, most of the optimization algorithms can barely do better than randomly guessing the labels, which has 10% accuracy and F1 Score. This is generally because the parameter space is so large in this network that exploitation works extremely slowly since it's usually dependent on neighbors. In other words, changing the parameter set of ~600,000 one parameter at a time is very slow. This raises an interesting point about these optimization methods: that in high dimensions, perhaps a better suited neighborhood function could be defined. The bottom right chart shows the performance of backpropagation. It optimizes extremely quickly and reduces training error to 0% at around 100 iterations, at which point the learning rate plateaus and both the cross validated and training F1 Score flattens. The final F1 Score for this algorithm was **81.3%**.

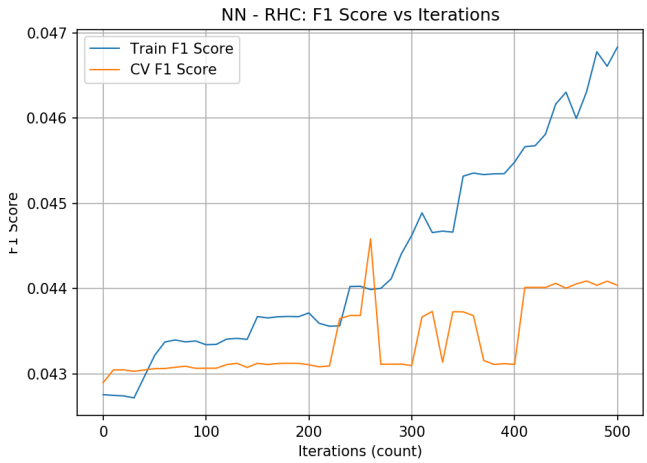




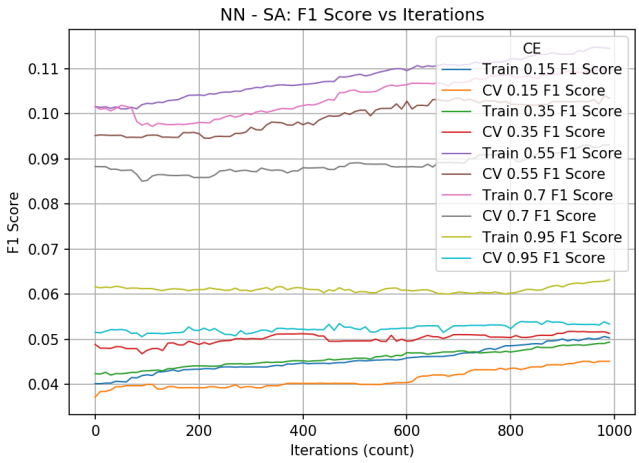
Taking a closer look at the GA performance, we see that it does the best out of the 3 randomized optimization algorithms tested, with a final testing F1 Score of **13.2%**. Since the network was so complicated, I was only able to test with parameters of 15 population, 3-7 mate population and 3-7 mutate population. Parameters of larger sizes usually caused a memory error or the runtime would be unreasonably long, likely because each instance of the population take a lot of memory to store and their neighbor sets are also quite large. The best performing parameters had 3 mate population and 7 mutate population. The differences between the mutate population setups are shown in the graph below. We see that although there are differences, the algorithm generally does not generally perform better than their starting points, meaning its performance is heavily dependent on the starting random parameters. This shows the inability for GA to actually get better fitness on this particular problem, which is due to the extreme complexity of a 10-label fully connected network with 784 hidden nodes. After a certain point, the mutation is not powerful enough to change the course of the optimization which means the performance more or less plateaus.



RHC performs a better in the sense that it is at least improving the training error each iteration. This is because RHC is a purely exploitation based algorithm and will always try to go to a neighbor that has better fitness. Unfortunately, moving through neighbors is extremely slow as it's only changing one of the ~600,000 parameters at a time. The improvement in training score also translates somewhat into validation score. Given enough time and resources, it would be interesting to see how well it can do with 50,000+ iterations. From this we can see that RHC works theoretically but is just extremely slow when the parameter space is large.



Simulated Annealing performs similarly to RHC in that we do see slow and steady progress as the number of iterations increases. However, much like RHC and GA, it's not significant enough so ultimately the performance is determined by the random assignment that happens at the beginning of the run. SA performs slightly worse than RHC and does not have generally have any advantages over RHC for this particular problem. The reason is because for Neural Networks, there are many optimums, even local ones, that can sufficiently fit the data and we only need to find one of these. This is one of the main assumptions behind the Gradient Descent algorithm. This means that the "exploration" phase of the SA algorithm does not add much benefit, when in fact always exploiting will generally produce better results.



Conclusion

Through these experiments we've explored four distinct optimization algorithms and their performance on both toy problems and on Neural Network weight optimization. MIMIC performed quite well on Flip Flop where there was a simple structure to the problem that the dependency trees could capture effectively. RHC and SA were robust optimization methods that didn't exhibit plateauing behavior common to GA and MIMIC. GA performed best on the Traveling Salesman problem, which allowed GA to make use of the crossover mechanic the best. In terms of Neural

Network weight optimization, Backpropagation was far and away the best algorithm. RHC was the best of the randomized algorithms for that experiment, since it behaves similar to Gradient Descent in the sense that it does immediate exploitation. The big difference is that it needs to scan through all its neighbors, all of whom only differ by one parameter, to find a better set of parameters. This is an extremely slow learning process unfortunately, but could potentially be fixed with a more aggressive neighborhood function. Overall, we see that different types of algorithms work best for different types of problems and that there isn't one type of algorithm that seems to do the best for everything. This perfectly exemplifies the implications of the No Free Lunch theorem in optimization.