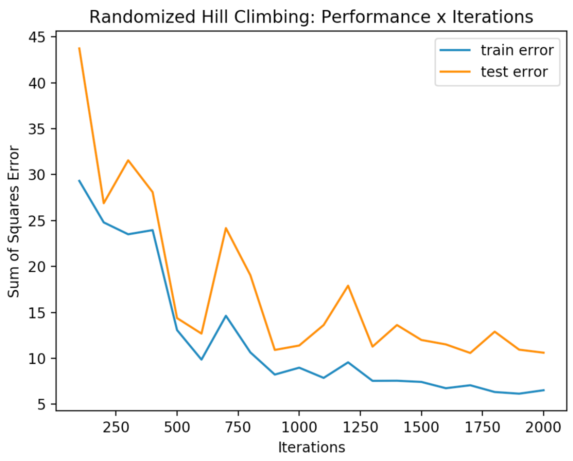
Randomized Optimization
Machine Learning Spring 2019
Sanjana Kadiveti

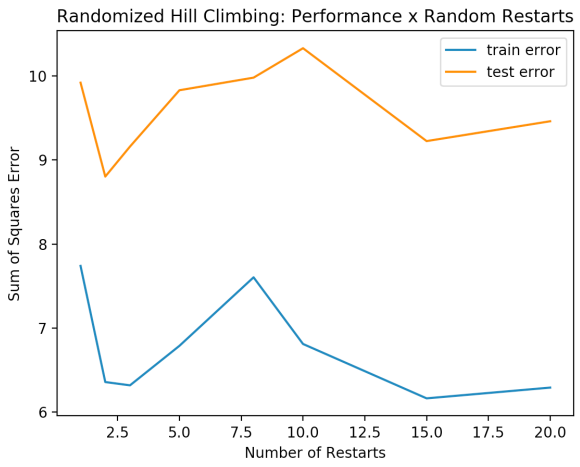
Neural Network Optimization

As a result of my Supervised Learning experiments, I wanted to learn how Randomized Optimization algorithms would perform on a Phishing dataset. This is a binary classification problem with 31 attributes and 11,055 instances. It had excellent performance using Supervised Learning methods, which is why I thought it would be interesting to explore what randomizing the search for a target function would do to performance on such a large dataset with many dimensions, each with fairly simple values.

Randomized Hill Climbing

For the randomized hill climbing algorithm, I split the Phishing dataset into a train and a test set of 70% and 30% of the data respectively. I conducted two experiments, one against iterations and one against random restarts, with a neural network of 2 hidden layers of 14 neurons each. I used this network configuration because it is what I found to be most appropriate in my last paper. With continuous values, RHC starts off by assigning weights uniformly. With each iteration, a neighbor is searched. This neighboring point is selected by varying a single weight's value by some random amount and finding a neighbor that has a higher fitness than the current point. In this case, the algorithm attempts to explore points with a decrease in sum of squared errors.





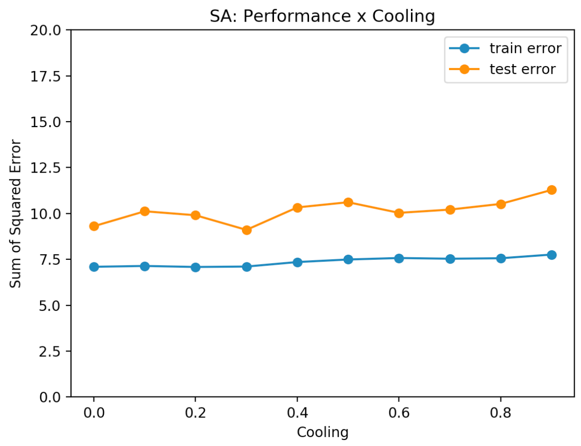
To the left is the graph of sum of squared error against number of iterations. An increase in iterations led to a steady decline in both training and testing error, and training converged to a value of around 7.4%. It is interesting to note that test error starts off with a steep decline, followed by a period of fluctuating error rates, and at around 1300 iterations we see signs of slow convergence. This is because, with the randomization involved, RHC doesn't always find the same optima, but past around 1300 iterations, the likelihood of it arriving or almost arriving at some optimal point is high. With this particular peak point, the model generalizes better.

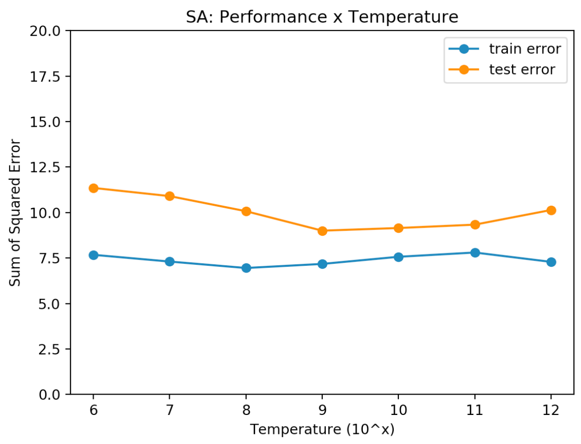
Neural networks with back propagation in my last report achieved a test accuracy of 95% with just 400 iterations, and from this graph we see that RHC takes much more iterations, and therefore much more time, to reach low error rates. If anything, the training convergence to 7.4% could mean it may never reach backprop's accuracy, because it's likely that the model has gotten stuck at a local optimum.

The graph above to the right shows how sum of square error changes with a different number of forced random restarts for RHC. We notice an unexpected jump in error from 5 to 7 random restarts, besides which there is a general trend towards lower error for more restarts. Perhaps the outlier can be explained by some region of the function where we see a much lower local optimum than we do at other regions, and we only reach it once because maybe its basin of attraction is relatively small. In general, we notice much better performance with restarts. We now see a test error as low as 9.4% with 15 random restarts. This is expected because as we know, restarts increase the algorithm's likelihood of finding the global optimum, and if not the global optimum, a better local optimum.

Simulated Annealing

Like RHC, simulated annealing selects a neighbor based on its relative fitness. The difference is, RHC is greedy, and thus assumes that the best point O is always reached on a positive trajectory from whatever current point we are currently on. SimA on the other hand, with a certain probability dependent on fitness and temperature, explores a neighbor with lower fitness if this probability



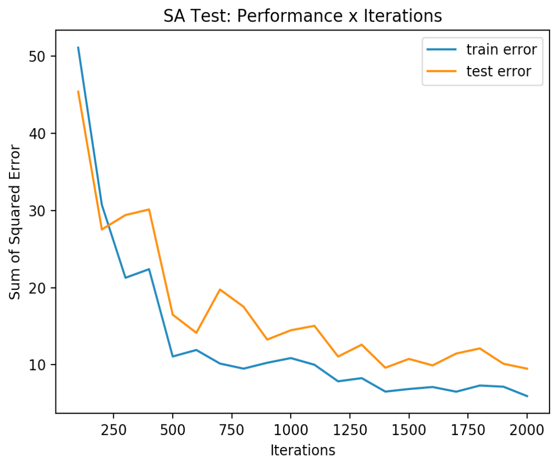


is greater than some threshold. Temperature controls how insistently the algorithm explores lower fitness points; high temperatures mean it will pick them more often. Cooling is what the temperature value is multiplied by at the end of every iteration; the algorithm essentially "cools" down by choosing to explore these points less and less until its end.

The figures above show how the algorithm performs as a factor of temperature and cooling. The temperatures used were 10^6, 10^8, 10^10, and 10^12, with a constant cooling rate of 0.75 and 1500 iterations. We see a decrease in error as temperature increases at first, followed by some stagnation. The test data shows a larger variance of results across the temperatures, which indicates variance in the dataset overall. This is likely a result of it having 31 attributes. Overall, a temperature of 10^9 performed the best, and I use it on the rest of my experiments.

With increasing cooling, error remains more or less the same, and towards the very end we see a slight increase in error. With a small cooling factor, temperature gets smaller more quickly, which means it explores worse points less and starts acting more like RHC. The increase in error with high cooling means exploring the search space isn't very beneficial given the number of iterations. This may give us some insight into what the dataset is like; perhaps there aren't many instances in which traversing seemingly worse paths lead to more optimal peaks.

An interesting difference between the SimA and RHC learning curves is, despite not really having better final error rates, SimA has fewer bumps and fluctuating error rates than RHC does. This is understandable due to SimA's inherent random exploration technique, which makes it more likely to see better results at fewer iterations and remain that way.

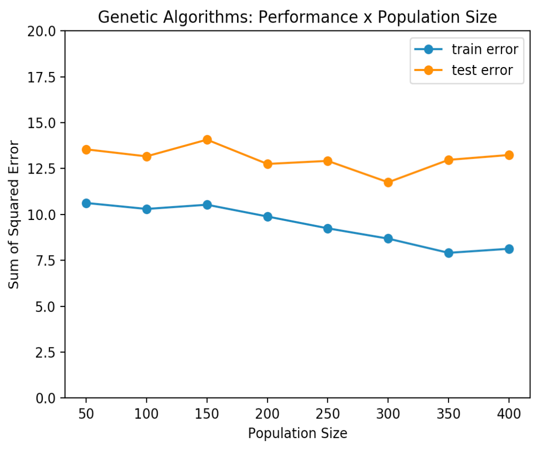


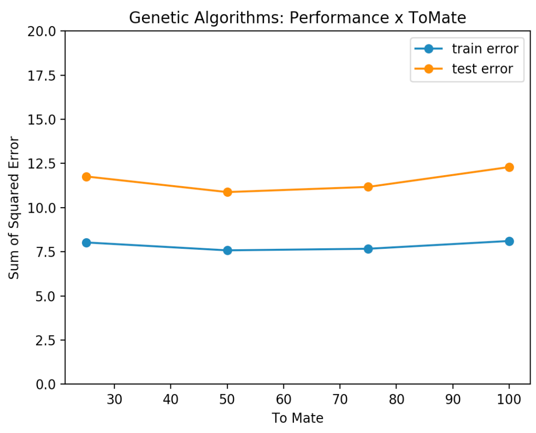
If I had to go back and test anything else on the last two algorithms, it would be the impact of temperature and cooling on performance on different numbers of iterations, because a large number of iterations would naturally aid both RHC and SimA in finding optimal points. I'm curious to know how temperature and cooling could step in and aid the algorithm if iterations are taken away, especially because in real life we wouldn't always have the time to run hundreds of thousands of iterations.

Genetic Algorithms

Genetic algorithms start with a population size and use mating and random mutation to alter the population and generate fitter points. Normalizing fitness scores allows the algorithm to generate a probability distribution. The mate factor determines how many times to crossover two models from the distribution to create a new one, and the mutation factor affects this new model by randomly picking a model and altering one of its weights.

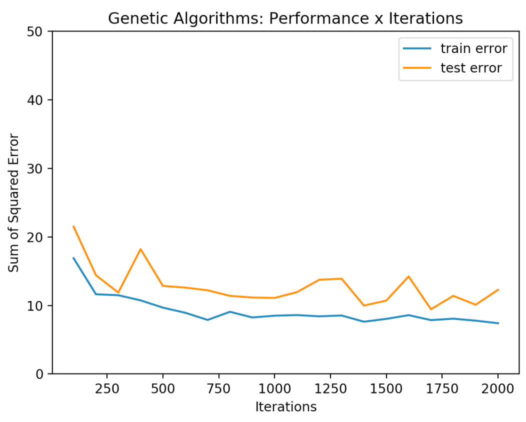
To perform my experiment, I tuned two parameters, initial population size and mate value, and plotted the algorithm's fitness over iterations using the optimal parameters found and a fixed value of 50 for mutate. While running the population size experiment, I used 50 for mate with 1500 iterations, and ran it over population sizes 50 to 400 in jumps of 50.





We notice from the graph on the left that GenAl generally performs better as size of the population increases. A small population may not be optimal because characteristics are more easily judged as poor and eradicated too quickly, leading to suboptimal crossover results. Too large of a population however, makes it difficult to get rid of undesirable characteristics because of the larger number and variety of solutions, which may not be compatible with each other. This is likely why we see an increase in error past population size 300, which is what I deemed to be the best population size for the dataset and is what I use for the rest of my runs.

I tested 25, 50, 75 and 100 as my mate values in my experiment over fitness. It appears to be that there is a happy middle level of mate that works best on the dataset, below and above which we see slightly higher error rates. This could be because with a lot of maters, there is a lot more variety, and since phishing is a binary classification dataset, the lower the variety the more likely it is we'll see better accuracy in classification. One the flip side, with very few maters and thus less crossover, the population is likely going to contain suboptimal individuals.



Increasing the mate ratio and population size increased the runtime, which is expected since more computation is required with more crossover and members of a population, but this could have been offset with an increasing mutate ratio, had I used one.

Overall, we see with the graph over iteration that GenAI accuracy can fluctuate even with the found optimal mate ratio and population size of 50 and 300 respectively. This is reasonable because there is still some randomness involved, especially with mutation being held at 50. In retrospect I wish I had optimized my results even further by tuning the mutate parameter, and that way could have made better comparisons with the other two algorithms. Nonetheless, we do still see a general trend towards lower error with more iterations.

Comparison

	Test Error	Train Error	Time (s)
RHC	10.6	6.5	36.98
SimA	9.46	5.94	37.35
GA	10.1	7.4	2984

Of all the algorithms, SimA with the right parameters learned the dataset the best, which we see from its low train and test errors. GenAI took the longest to run, which naturally is linked to the Phishing dataset's large size and feature space. Unfortunately, we don't see a final performance with GenAI that's better than SimA with many iterations. GenAI's real benefit comes with fewer iterations; unlike the other two, it doesn't need many of them to find fit solutions. However, RHC and SimA are still preferred in this case because they can find similar results in large numbers of iterations that take less time than GenAI does.

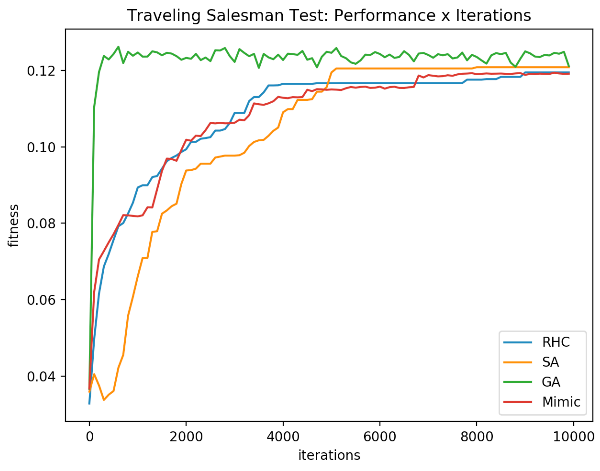
Interestingly, none of them were able to reach the 5% BackProp error rate. Backpropagation was also able to perform quicker than these algorithms, which may be because backpropagation corrects the weights at each instance or step rather than at the end of every iteration like these algorithms do.

Interesting Problems

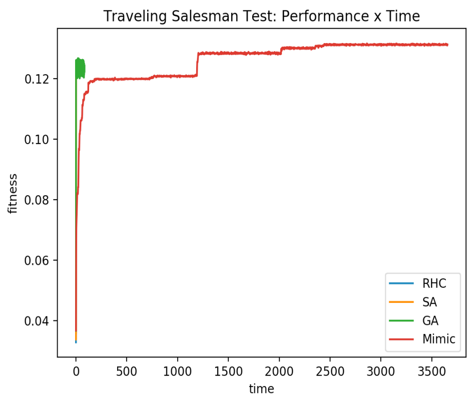
The following were performed with the following parameters:

```
SA: Temp = 10^11, Cooling = 0.95;
GA: population size = 200, mate = 100, mutate = 10;
MIMIC: samples = 200, toKeep = 20;
Iterations = 200,000
```

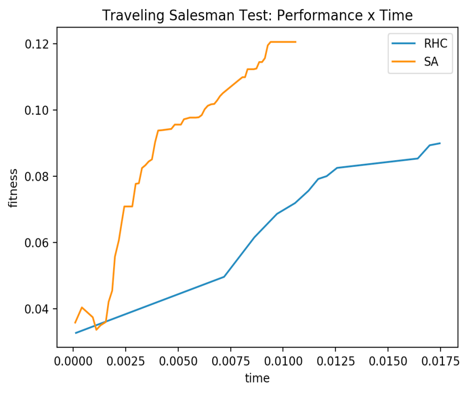
Traveling Salesman (Genetic Algorithms)



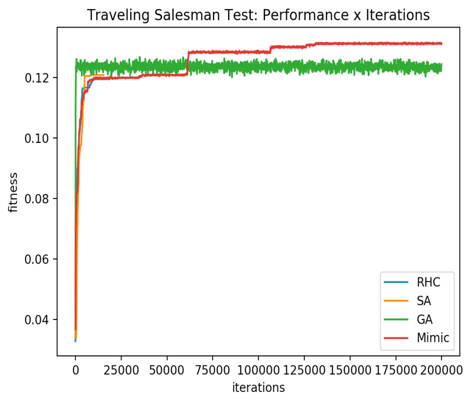
This problem deals with finding the shortest distance path a salesman could take through multiple cities. It's interesting because there are multiple solutions, some less optimal than others, which makes it a good example of when RHC and SA's "random walking" technique isn't as effective. I ran the problem first with 10,000 iterations using the 4 optimization algorithms on randomized graphs of size 50. The graph on the left shows the fitness curves over 10,000 iterations, from which we see that of all the optimizers, GenAl performs the best - best being defined in this case as the one with the highest fitness. This could be explained by the fact that GenAl, in its attempt to preserve diversity, becomes better at exploring functions and is suited for many frequently changing heuristics.



RHC and SimA converge at suboptimal solutions, and as expected, SimA took longer than RHC to converge. This could be explained by the fact that TSP favors and has multiple similar solutions, which makes it extremely difficult for blind fitness traversing algorithms like RHC and SimA to find more optimal ones among all the local minima. It also gives us a reason for why GenAl works so much better: its crossover function, un-fooled by the similarity between solutions, is able to find a fitter solution much quicker than any of the other optimizers.



The graphs to the right show time taken to reach fitness levels. To better zoom in on RHC and SimA's relative performances, I plot that on a separate graph. Mimic took magnitudes more time than the other algorithms, which can be explained by the problem's extremely large hypothesis space; there are a large number of paths joining a set of 50 points on a connected graph, which leads to longer running times due to Mimic's probabilistic sampling nature. It does however, continuously improve; it just needs more than 60,000 iterations to outperform the other algorithms, as can be seen on the graph on the left, which I plotted out of curiosity for at what point MIMIC defeats GenAl's fitness.

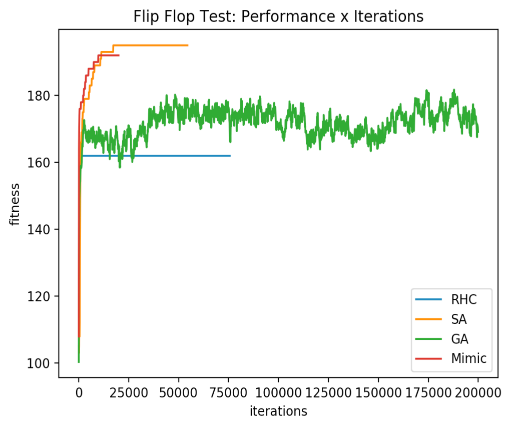


(all iterations)	Final Fitness	Time
RHC	0.119	0.45
SA	0.120	1.08
GenAl	0.125	8.17
Mimic	0.131	3655

Despite MIMIC's ultimately better fitness, I don't think its improvement of fitness over GenAl's makes up for the runtime and iterations it needs to get there. A way in which I could have made MIMIC's code run faster is by decreasing the number of points in the graph, or the number of samples MIMIC took, but a smaller sample size makes the optimizer less likely to converge to a global maximum.

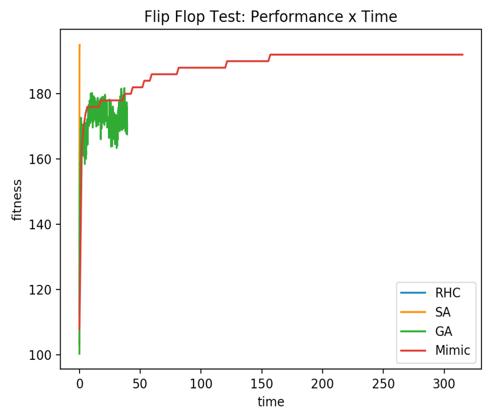
Therefore, I'd call GenAl the best option – this time, best being defined as a combination of fitness and time - but I acknowledge that relative to RHC and SimA, whose final fitness values aren't far off from GenAl's, GenAl takes relatively long to run. However, this is typical of genetic algorithms and could have been reduced by increasing the mate ratio, which would've led to faster convergence, but with some bias introduced.

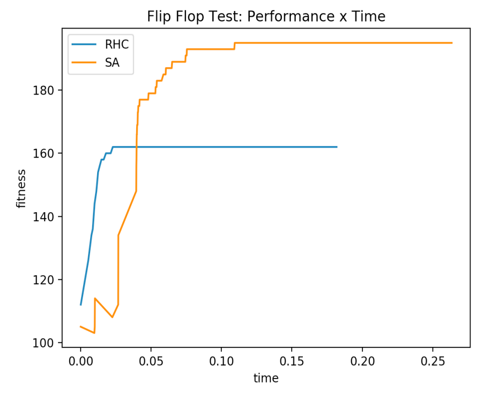
Flip Flop (Simulated Annealing)



The flip flop problem is a bit string one, and its fitness is determined by the frequency of seeing consecutive bits that alternate. The problem's function contains more suboptimal maxima, which makes it a good example to demonstrate the differences between RHC and Simulated Annealing. It only takes one input, the length of the bit string.

I ran the flip flop problem on randomized graphs of size 80. The graph above is of fitness with respect to iterations, and on the right are graphs of time taken for fitness levels and a table of final values at either convergence or 20,000 iterations. I plot a second one of RHC and SimA since they aren't visible in the first.



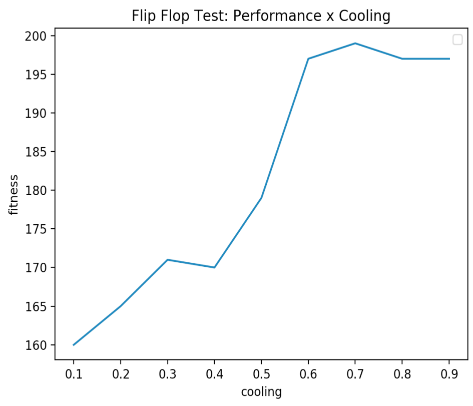


We notice that RHC quickly converges to a very suboptimal solution – this is an example of when traversing a seemingly worse path can lead to the highest peak. We notice that MIMIC performs really well at first, but doesn't converge to as optimal of a solution as SimA. This is likely because probabilistically estimating feature importance gets close to but doesn't completely capture the exact optimum point of perfectly alternating bits. The genetic algorithm doesn't converge or reach the optima, but it performs better than RHC does, and this can be explained by the fact that crossover may not work especially well in this case because two fit points may not necessarily be optimized independently. It's likely that they have the same kind of strengths and weaknesses (the latter being non alternating bits), and so combining two points wouldn't always lead to more consecutively alternating bits.

At around 18000 iterations, Simulated Annealing converges to an optimal solution that none of the other optimizers are able to achieve, all the while maintaining the second lowest runtime, which is why I think SimA is the best algorithm for the problem.

all iterations	Final Fitness	Time
RHC	162	0.45
SA	198	0.78
GenAl	176	39.32
Mimic	192	316.6

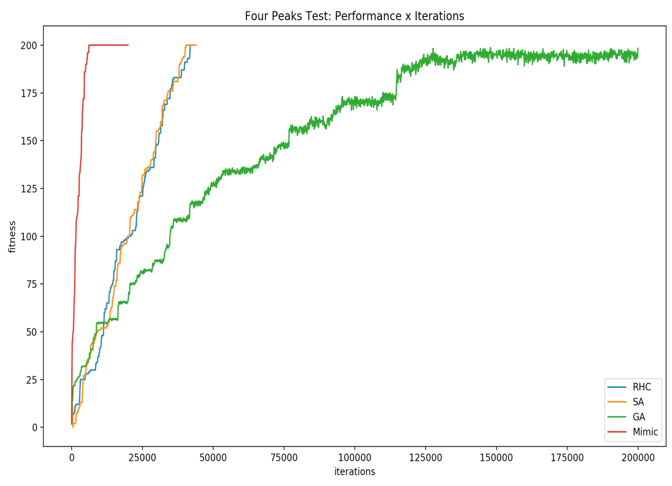
Curious if SimA's performance was impacted by cooling rate in any way, which for the experiment above I set to 0.95, I conducted an experiment varying the cooling rates to see impact on fitness. The graph to the right shoes us that Simulated Annealing fitness excels only past a certain cooling threshold, specifically around 0.8. Increased cooling is equivalent to maintaining the rate at which the optimizer explores seemingly worse paths. This means, in order to find optimal points in the search



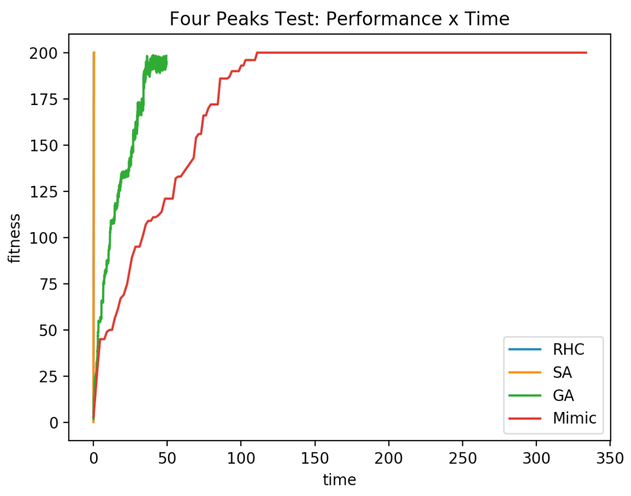
space, some exploration is necessary along with exploitation.

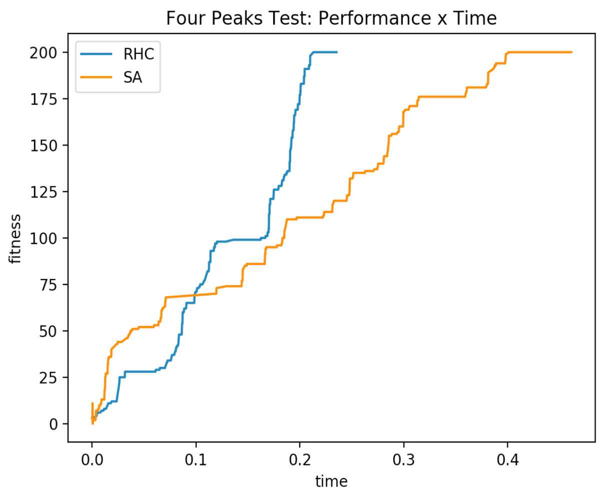
4 Peaks (MIMIC)

The purpose of the 4 peaks problem is to maximize a function with an N dimensional input vector. Its goal is to maximize the number of trailing 0s and leading 1s, and has a reward function that increases based on the values in relation to a threshold T.



From the graph on the left we see how each of the 4 algorithms perform in terms of iterations and time. We immediately notice Mimic's effectiveness; it required significantly less iterations to find the global optima. On the flip side, we notice that unlike the rest of the algorithms, GenAI wasn't able to find the optima within 20,000 iterations. I tuned its parameters and ended up using a population size of 200 (p/100 mate, p/20 mutate), and we see that it gets extremely close to the optima but still doesn't reach it.





The graphs above are of fitness in relation to time, the right one being a close up comparison of the RHC and SimA time. Naturally, SimA took a little longer which is expected given its nature of having searches additional to RHC's. Overall however, they performed almost equivalently well, both in terms of time and iterations. This means that the tendency to explore worse regions doesn't change the results much. I set my bit string length to 200, and perhaps that makes it a relatively straightforward path to traverse and an easy problem to solve. If the function isn't complex and there aren't many local optima, the extra search over neighborhood that SimA offers isn't required. With an equal fitness and lower time, RHC is the better option between the two for simple functions.

The structure of the problem is what makes MIMIC function so well. It's able to build off of information gained through every iteration, unlike the other algorithms, which step through without intelligently retaining information about the underlying distribution. As well as Mimic apparently performs however, we must acknowledge that it still took the longest time to find the optima. This is a tradeoff with Mimic in general – less iterations required because it learns structure along the way, but each iteration takes longer. For a problem as simple as Four Peaks, a blind function searcher like RHC is also efficient, but it sheds light on situations in which MIMIC is extremely beneficial, which are ones where the underlying distribution can be modelled.

Conclusion

Randomized optimization didn't perform quite as well as backpropagation did on the phishing dataset, as previously mentioned. BackProp corrects errors and propagates information, and so it works on the basis of learning. These optimizers on the other hand, are, for a lack of a better term, *random.* They pick values and hope for the best, clinging to what works and in some cases, throwing away what doesn't. This means they can take a lot longer to achieve the same results. In some of my experiments, perhaps there just weren't enough iterations performed for them to work as well.

From the problem section, we find that knowledge of the problem space is extremely beneficial when selecting which randomized algorithm to use to find an optimal solution. RHC and SA are better for distributions that don't appear to have any structure, for example. When there are many local optima with small basins of attraction, RHC and SA wouldn't perform as well, but MIMIC would not waste time exploring suboptimal pathways. With genetic algorithms, a degree of locality between features works better on its crossover function.