Randomized Optimization - CS 4641

Omar Shaikh

March 3, 2019

Abstract

In this report, I cover applications of the following randomized optimization techniques: Random Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA), and MIMIC. To do this, I explore these algorithms in two distinct domains. My first domain involved testing the first three optimization techniques on a neural network. The second domain involves testing optimization problems on all the techniques, while highlighting strengths and weaknesses of each. In the conclusion, I provide a quick recap on general situations where these algorithms perform best.

1 Optimizing Loss for Neural Networks

1.1 Recap from Supervised Learning

In project 1, I used two datasets to run my experiments. For this section, I used only the Credit Card Fraud classification dataset. Recall that the dataset was highly imbalanced. For the sake of brevity, this paper will only focus on the undersampled dataset, as training on project 1 only occurred on the undersampled dataset. With gradient descent, our optimal hyperparameters were as follows: [learningRate = .001, l1 = .1, epochs = 25, layers = 1, nodesHiddenLayer = 32].

Our cross validation accuracy on the undersampled dataset reached .949; however, recall that almost every variation of the neural networks tested converged with similar accuracy (margin of -+ .001%). This makes me suspect that there exists an obvious global minima that gradient descent found immediately. The main objective of this subsection, then, is to see which algorithm converges the fastest. Because of how obvious this minima must be, I hypothesized that RHC would be the best – there's no need to get out of a local minimum because the attraction basin for the global minima appears to be very large. Figure 1 highlights what I suspect my search space looks like. Although Figure 1 highlights a maximization problem and we attempt to minimize the loss function, the same idea applies.

1.2 Procedure for Finding Optimal Hyperparameters

For each of the following algorithm subsections, here are the common procedural precautions and notes.

- Normalized the Credit Card Fraud dataset using sklearn's Robust Scaler, which scales each feature according to its quantile range. In my case, I set the new range to be (-1, 1).
- Analyzed only the undersampled dataset, because of this paper's goal (comparing optimization algorithms).
- Limited number of optimization iterations to 2000, to keep things fair.
- For all mentions of CV, assume I'm using 5 folds on the train data, with an 80-20 split between test and train.
- Used binary cross entropy as the loss function, like in Assignment 1.
- Used a personally edited version of mlrose,[1] a machine learning and random optimization package for Python.

Also, note that for the following subsections iterations and attempts are slightly different hyperparameters. Here's the distinction: if the best neighbor is an improvement or our algorithm allows us to explore and not exploit, then move to that state and reset the max attempts counter; however, keep increasing iterations regardless.

1. G Hayes, *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python*, `https://github.com/gkhayes/mlrose`, Accessed: 22 February 2019, 2019.

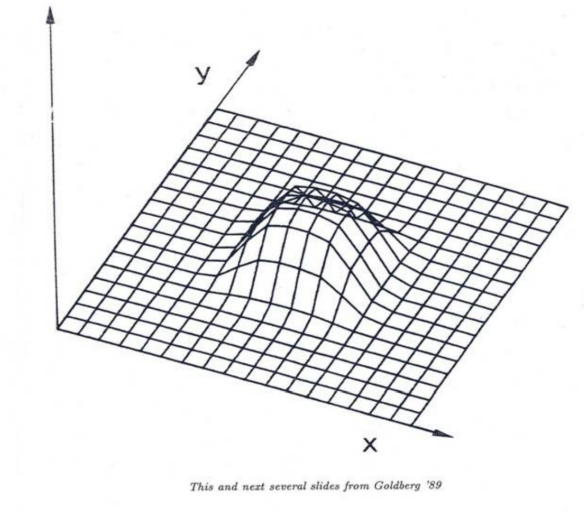

Figure 1: Hypothesized Problem Space for NN

1.3 Randomized Hill Climbing

The only hyperparameters that need tuning are the number of restarts and the learning rate, where the learning rate is the change in the direction towards the optimal neighbour. To find the optimal hyperparameters, I used random search over the following search spaces: $learningRate = rand(.01...1)$ and $restarts = \{rand(1...100)\}$. The best parameters were restarts = 78 and learningRate = .27, with a cross fold accuracy of .914.

Figure 2 highlights the cross validated percent accuracy over these random hyperparameter search spaces. It appears that only the learning rate has an effect on our accuracy; the outliers in the max attempts and restart graphs are the same points with low RHC learning rates. This makes sense, because our learning rate dictates how fast we climb this simple search space. Our attempts don't matter, because we always find a better neighbour until we converge. Finally, our basin of attraction is large, so the number of restarts doesn't matter either. Figure 4 confirms that RHC works remarkably well; given 2000 iterations, the best restart (78) settles in on what appears to be the global minima.

Upon inspection, all restarts remarkably well, with a small difference in final loss (less than .01). Because of how simple the search space is, RHC performs nearly as well as gradient descent. Due to the performance similarities, I'm tempted to say that the difference in weights between RHC and gradient descent may just be random noise. Also note that because of the search space's simplicity, RHC with few restarts will prove to be very fast in the comparative analysis section.

1.4 Simulated Annealing

For simulated annealing, I tuned the decay schedule and the maximum number of attempts to find a better neighbor at each step, while keeping cooling and temperature constant for each respective decay schedule (initTemp = 1, expCooling = .005, geomCooling = .99, arithCooling = .001). Figure 3 shows cross-validation percent accuracy w.r.t random configurations of SA. The following hyperparameter sets define these random configurations: $decaySchedule = [exponential, arithmetic, geometric]$) and $maxAttempts = \{rand(1...500)\}$. The best hyperparameters were maxAttempts = 31 and decay schedule = exponential, with a μ 5-fold CV accuracy of .73.

Note that as SA gets closer and closer to RHC (faster cooling decay functions; more exploitation, less exploration) we see faster convergence of SA. Because the speed of both algorithms are close to the same, the less "risks" SA takes (i.e the closer it is to hill climbing), the quicker it can reach this convergence. At worst, SA could cool down too quickly with a high starting temperature, resulting in local minima convergence. Max attempts has little effect on the cross validation accuracy for the same reasons as RHC. We will almost always find a better neighbour due the simplicity of this problem; if not we may be allowed to explore the search space.

However, because SA spends more time exploring rather than exploiting, it wastes precious iterations as compared to RHC. Figure 4 highlights this.

1.5 Genetic Algorithm

Because of the conclusions drawn when training SA and RHC, I hypothesized that GA would struggle to reach the same maxima as RHC. Convergence would also take an order of magnitude longer because it would

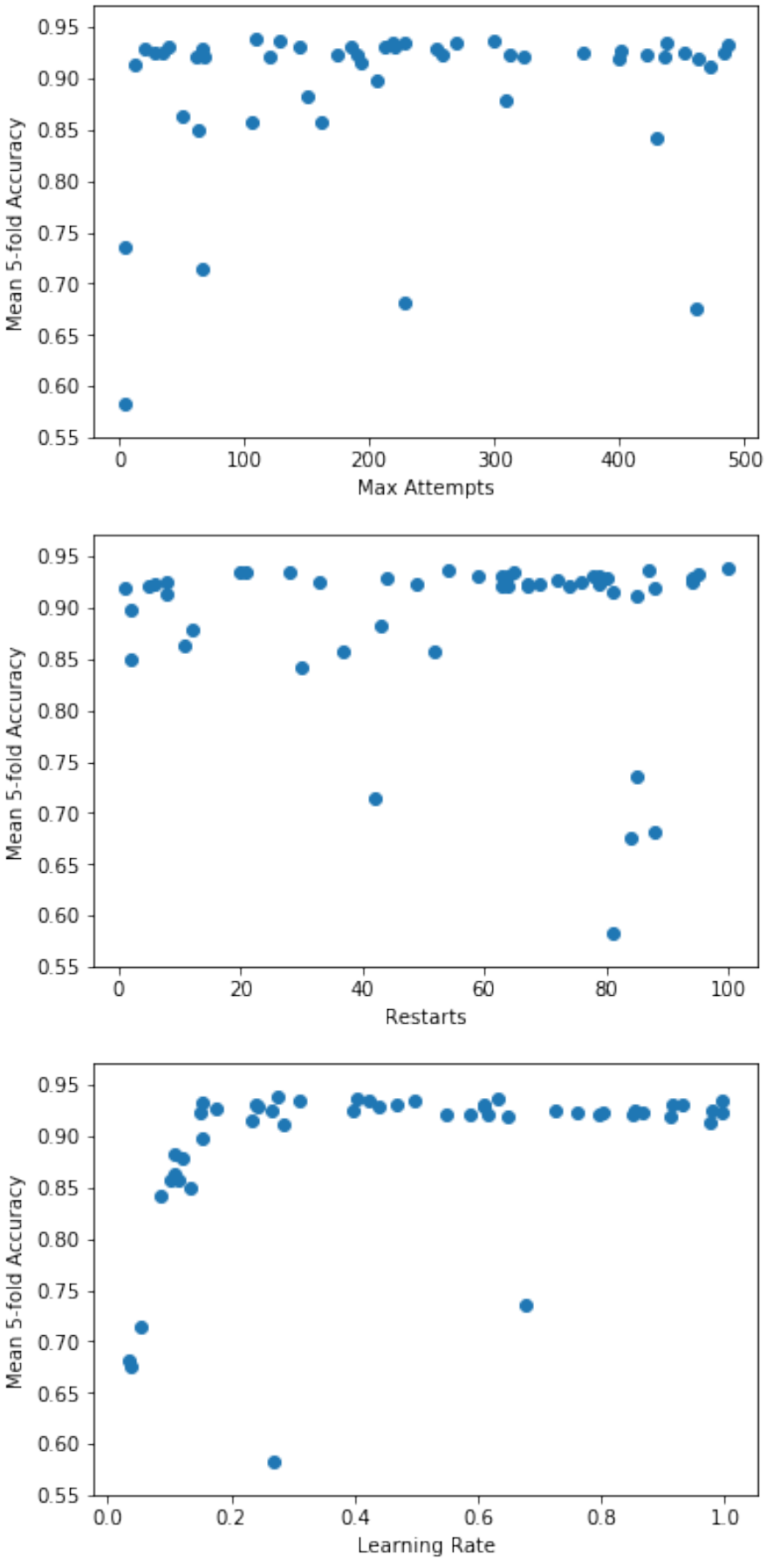

Figure 2: Random Hyperparameter Search for RHC

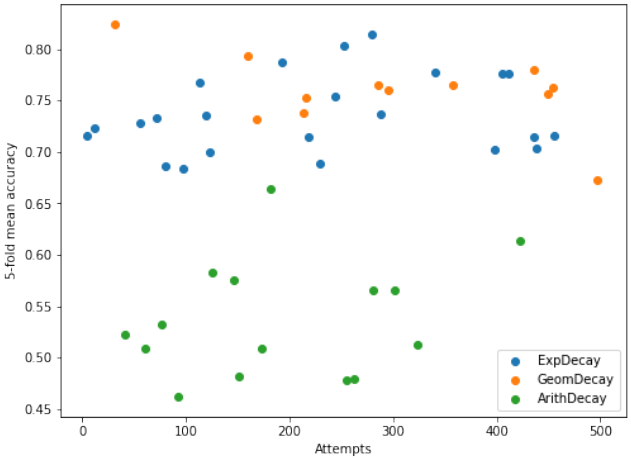

Figure 3: Random Hyperparameter Search for SA

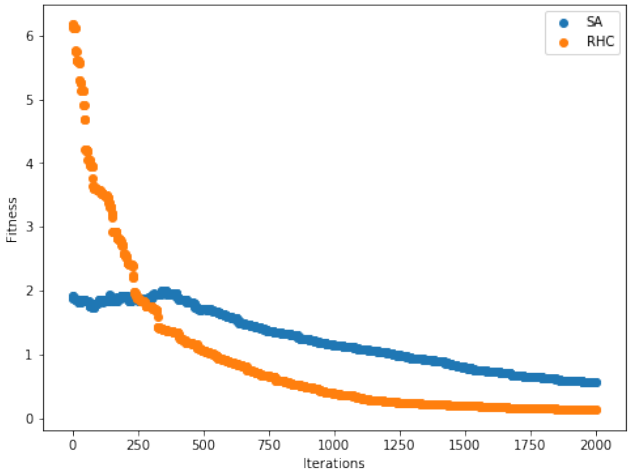

Figure 4: Loss functions for SA and RHC w.r.t iterations

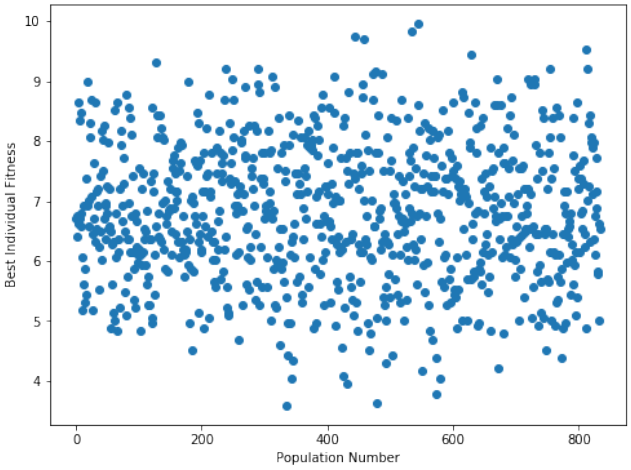

Figure 5: Loss for best individual w.r.t to generation/poulation # for GA

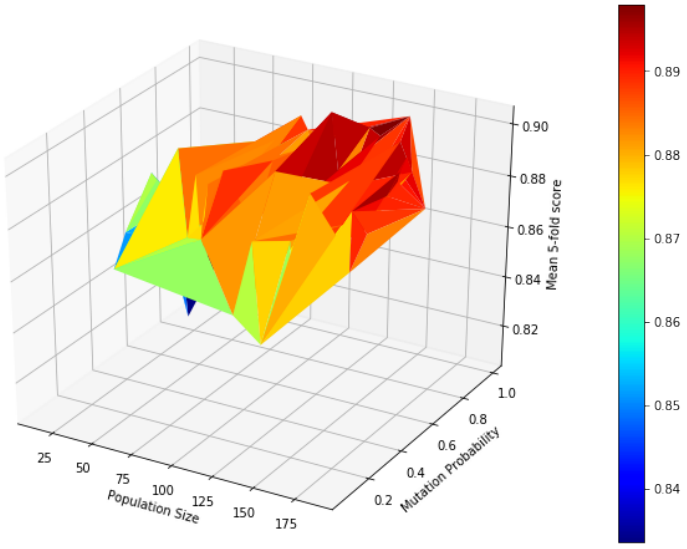

Figure 6: μ CV score w.r.t Population Size and Mutation Probability for GA

generate an entire population of random points. Furthermore, the increased computation time would be wasted, as GA may direct the weights of the NN towards local optima through this misinformed crossover. Again, I used random search and cross validation on the following hyperparameter search spaces: $pop\_size = \{rand(0...200)\}$, $maxAttempts = \{rand(1...300)\}$, and $mutation\_prob = \{rand(0...1)\}$. Note that the implementation of the algorithm I used utilized one point crossover and uniform random mutation. The best parameters were mutation_prob=0.788, pop_size=114, and max_attempts=500 with a μ 5-fold accuracy of .893.

Figure 5 shows GAs inability to improve fitness of the best individuals as the population continues to reproduce. Surprisingly, cross validation scores still appear relatively high w.r.t hyperparameters pop_size and mutation_prob (Figure 6). However, as population size and mutation probability increases, μ CV scores increase too. I suspect that increasing these values forces GA to consider the obvious global maxima moreso than when having smaller respective hyperparameters. However, increasing these values feels naive, in that I'm "brute forcing" the search space. I also didn't plot max attempts, as it appeared to be independent of μ accuracy. Finally, I suspect that because GA doesn't fit the dataset too closely, it generalizes better.

1.6 Comparative Analysis

The following table summarizes our hyperparameters, μ 5 fold CV test time, μ CV accuracy, and test accuracy.

Hyperparameter & CV Accuracy/Time Tables

Algorithm	Best Hyperparameters	μ CV sec/fold	μ CV Accuracy	Test Accuracy
RHC	restarts = 78, learningRate = .27	208.11[2]	.914	.928
SA	maxAttempts = 31, decay = geom	5.70	.73	.807
GA	max_attempts=500 pop_size=144 mutation_prob=0.788	74.88	.879	.888
Gradient Descent	Same as Assignment 1. See 1.1.	N/A	.949	.934

In my problem, RHC is the best, because it easily stumbles upon the optimal basin of attraction and turns into gradient descent (from Assignment 1) but with a "dumb static derivative." This dumb derivative (the learning rate of RHC) was good enough that we converged on what I think were very close to the weights of gradient descent. The same assumptions generalize well to SA – the closer the decay was to RHC, the better it performed. In other words, our temperature with exponential or geometric decay decreases rapidly, encouraging exploitation earlier. This makes SA "closer" to RHC. Finally, GA's loss function was the worst (Figure 5), because its implicit bias is that mutation/crossover leads to a better optima. However, there's only one optima (or it appears so), so every crossover would just harm minimizing loss and be a wasteful computation. Note how this is reflected in Figure 5, where the loss fluctuates wildly after each generation #.

RHC does not fit the fastest because of the excessive size of the restart search space used by random search cross-validation. Note that an additional model I tested with the same learning

2. This is overblown due to excessive restarts. Note the faster RHC discussed in the comparative analysis subsection.

rate and only one restart had a fit/fold time of 2.55 sec, and a CV score of .91, which is close enough to be attributed to chance. Our other observations line up closely with what I expected – SA follows close behind the faster RHC, and GA wastes time computing unnecessary info.

To conclude, our best cross validation accuracy (from RHC) was 3.5 percent off (91.4% vs 94.9%) and our best testing accuracy was .6 percent off (93.4% vs 92.8%), when compared with Assignment 1. I suspect that this difference may be due to randomness in weight initialization. On top of this (while accounting for unnecessary restarts) RHC was the fastest by a significant margin. This is wild because, realistically, RHC should do the worst, confirming my suspicion that this dataset is "easy" to learn.

Further work to improve the aforementioned results could include playing with the temperature and cooling of simulated annealing, though I suspect simply decreasing temperature and/or increasing cooling would lead to better results (again, because this makes SA more like RHC). The same could be said for increasing mutation rates for GA.

2 Miscellaneous Optimization Problems

2.1 Hyperparameters and General Configuration

For each of the following problem subsections, here are the common procedural precautions and notes.

- Used a personally edited version of mlrose,[3] a machine learning and random optimization package for Python.
- Ran 10 or 3 trails per algorithm for the problem size and iteration graphs respectively. This helps reduce random variance. I used fewer trails on iterations graphs due to time and computation limitations.
- Set max attempts to 200 and infinity for each algorithm on problem size graphs and iteration graphs respectively. Did not limit max attempts for iteration graphs because I wanted to run through ALL iterations and not prematurely stop.
- Set the problem size to 50 when testing iterations.
- Used 10 restarts for RHC.

Note that the same distinction between iterations and attempts as detailed in Section 1.2 applies. The following table summarizes the rest of our hyperparameter choices.

Hyperparameter Choices Table

Algorithm	Hyperparameters
RHC	restarts = 10
SA	maxAttempts = 31, decay = geometric, init_temp = 1, cooling=.99
GA	pop_size=200, mutation_prob=0.1
MIMIC	pop_size=200, proportion_pop_keep=0.2

2.2 The Parity Sum Problem

This problem aims to highlight the strengths and weaknesses of our algorithms on a search space with many local optima with configurable depths. The problem is as follows: maximize the sum of the bits; if the sum is even, you receive a parity bonus. For example, if we set our bonus to 1.1 and our sequence size to 4, then following sequence [0, 1, 1, 0] receives fitness 3.1, while this sequence [0, 1, 0, 0] has fitness 1 (no bonus). The global optima of this problem, where size = 4, is [1, 1, 1, 1] with fitness 5.5. By increasing the parity bonus, we increase the depth of the local optimas, posing an interesting challenge to our algorithms.

One of the nice elements of this problem is its ability to highlight how RHC fails when there are multiple local optimas, and how SA struggles when the depth of the local optimas increases. I tested these algorithms on two different setting of depth/parity bonus; depth = 1.1 (Shallow) and depth = 5 (Deep).

2.2.1 Testing Iterations

As hypothesized, SA improves when the local optima are shallow, but gets stuck when the optima are deeper (Figure 7). I tried adjusting temperature and cooling values, but the effect of the local optima depth could not be accounted for by these hyperparameters. MIMIC instantly converges on the optimal solutions (51.1 and 55), while RHC continues to struggle in both scenarios.

3. Hayes, *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python.*

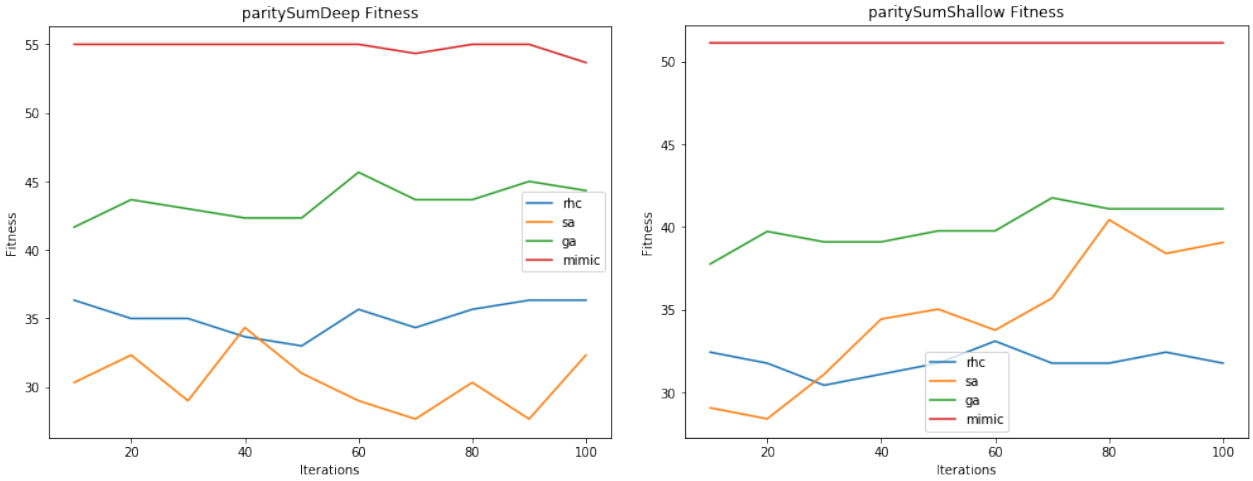

Figure 7: Fitness for Deep (left) and Shallow (right) Parity Sum w.r.t Iterations.

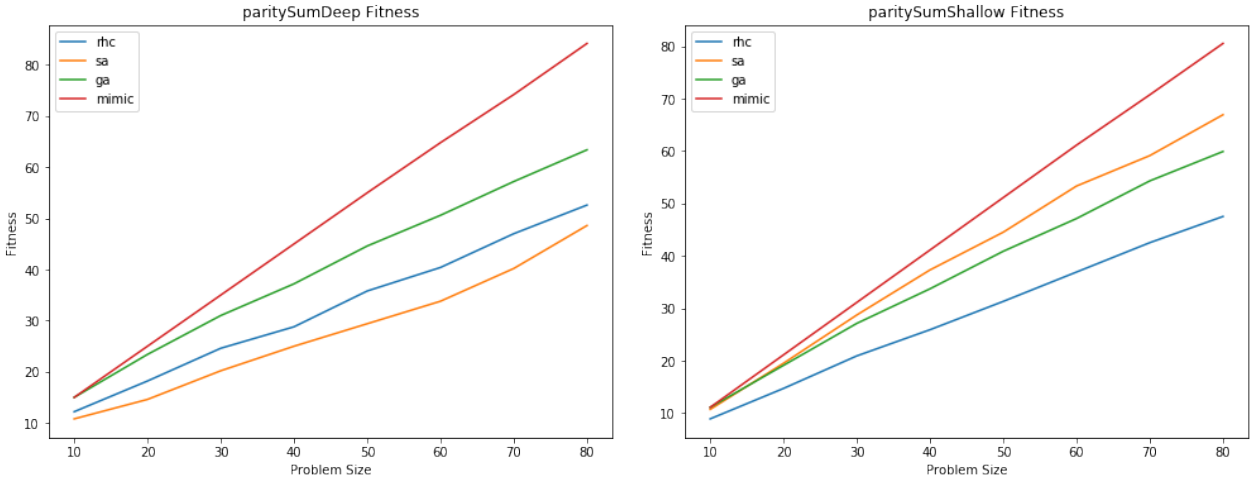

Figure 8: Fitness for Deep (left) and Shallow (right) Parity Sum w.r.t Problem Size.

What's somewhat surprising is RHC's capacity to do better in the deep situation than SA, though I suspect this may be due to the limited number of iterations. GA also improves steadily, but not significantly – this may also be due to our iteration limit. We'll see the true performance w.r.t problem size in the next subsection, where the iteration count is unbounded (but max attempts are).

2.2.2 Testing Problem Size

Here we see SA shine in a situation with many shallow optima. It performs better than GA, and just under MIMIC (Figure 8). Conversely, we also see it struggle when the optimas are deep, for the same reasons discussed in the subsection above. In fact, it does worse than RHC, which is somewhat surprising. In retrospect, the number of restarts (10) may have been enough for RHC to perform better than SA. Note that although MIMIC converges at the optimal solution more often for both variations, it takes far longer to evaluate it (Figure 11). GA, taking less than half the time on average, performs nearly as well.

2.3 Consecutive One & Product Sum Problem

My main goal for this problem was highlighting a situation that plays well to GA (spoiler: GA does the worst and RHC does the best, hah). The problem is as follows – we take the sum of the consecutive ones, and multiply them together to get the fitness. For example, the following sequence [0, 1, 1, 0, 1, 1, 1] has fitness 2 * 3 = 6. For this problem, GA's crossover should help find points to "break" the sequence in order to maximize the fitness. Also, because the probability of a "good" solution depends on other sub-solutions, MIMIC may have trouble modeling the distribution for this problem.

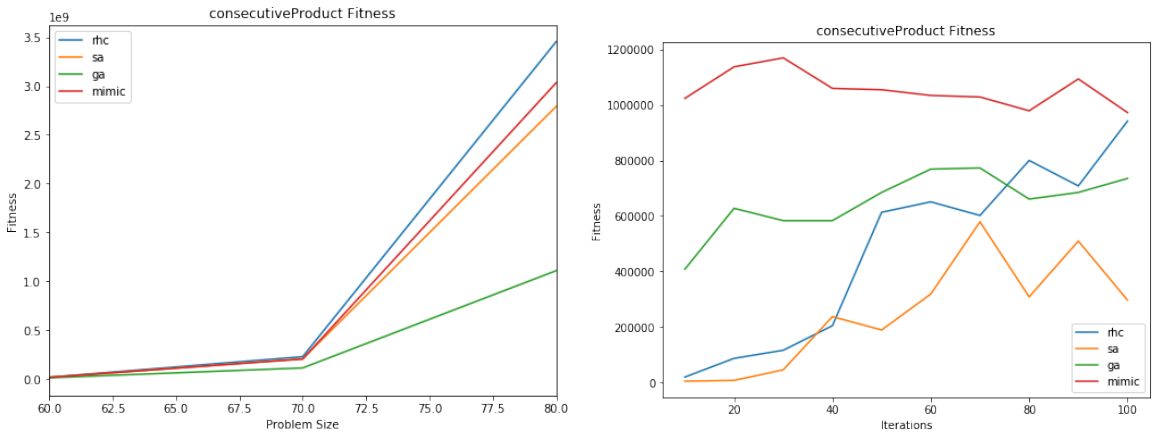

Figure 9: Fitness for Consecutive One Problem w.r.t Problem Size (left), Iterations (right).

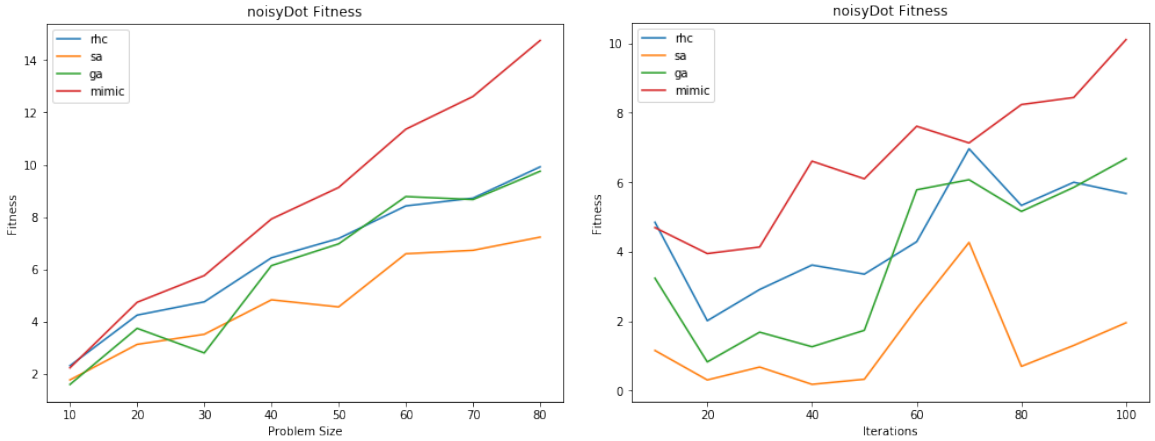

Figure 10: Fitness for Funky Dot Product w.r.t Problem Size (left), Iterations (right).

2.3.1 Testing Iterations

I was annoyingly surprised that MIMIC converged upon a good solution so quickly, and that RHC was converging upon the same one too (Figure 9)! In retrospect, this problem is good for RHC because it has few local optima. I expected GA to perform better, but I failed to see it converge on a better solution. SA performs the worst w.r.t small iteration counts, probably wasting time exploring useless optimas.

2.3.2 Testing Problem Size

Note that Figure 11 starts at problem size = 60. Without this cap, the disparity isn't obvious. If the reader is interested in smaller values, refer to the Jupyter Notebook and remove the xlim argument.

I'm a little disappointed because I tried to break MIMIC, and show a problem where GA runs the best. However, RHC continues performing well, surpassing MIMIC (Figure 9)! The lack of local optima in this problem make it a good example for the benefits of RHC (just like the neural network optimization section above). I suspect that GA is suffering because of a bad mutation rate or a low population count, but I'm not entirely sure why. Future work could involve expirment with the decay schedule and the aforementioned hyperparameters.

2.4 Funky Dot Product

The following problem aims to highlight MIMIC's strengths relative to our other optimization algorithms. The problem is as follows: first, we initialize a uniform random vector with values between -.5 and .5. Then, on each iteration of the fitness function, we add noise to this uniform random vector, by adding another normal random vector with $\mu = 0$ and $\sigma = .25$. Our "misleading" fitness is finally calculated by dotting the noisy random vector with our state bitvector.

Because MIMIC can model the underlying distribution from the noise, it should be able to perform far better than the other algorithms, ultimately discerning the true fitness (just the dot product of the state bitvector and

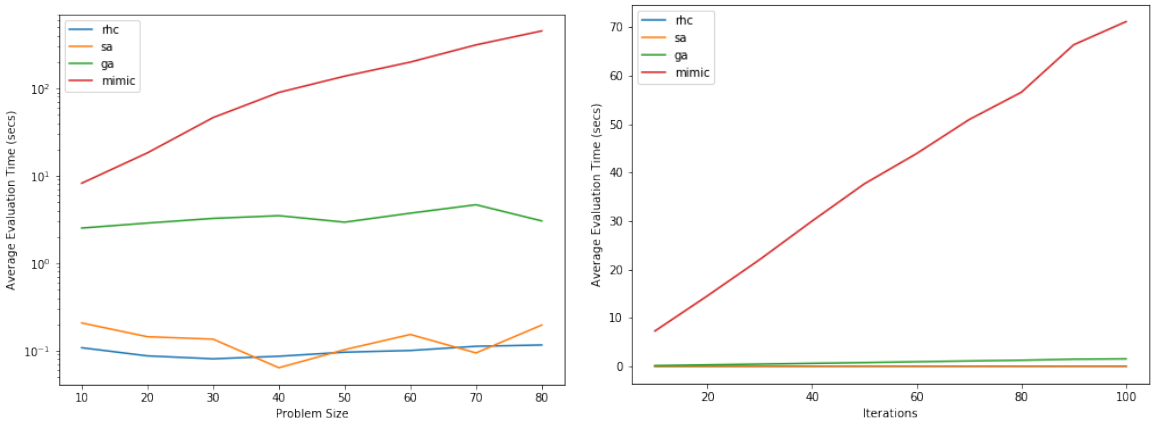

Figure 11: μ Time (secs) w.r.t Problem Size (left), Iterations (Right)

the original uniform random vector) from the noise. Note that for all algorithms, the uniform random vector was held constant.

2.4.1 Testing Iterations

In this situation, we see MIMIC shine (Figure 10). It maintains a lead over the rest of the algorithms w.r.t iterations. RHC gets close as the iterations increase, but because MIMIC learns the noise distribution, it should perform better – in the next subsection, we get to see MIMIC pull away from the competition.

2.4.2 Testing Problem Size

Unsurprisingly, MIMIC leads again (Figure 10), and widens the gap as the problem size increases. RHC also performs well, because of how "random" this problem is. Due to the lying fitness function, RHC is the best you can do after MIMIC. SA breaks down completely, because the explore-exploit strategy is no longer reliable. Depending on the fitness function's noise, SA may be exploring when it thinks it's exploiting, and vice versa. Finally, GA's reproduction method hinges on what individuals have good fitness; again, the fitness is unreliable, so the best individuals may never breed.

2.5 More Comparative Analysis

Before harping MIMIC as a master of all and the general "go-to" algorithm, it's important that we notice how MIMIC scales as iterations increases. The news is not good. While SA, GA, and RHC have a small slope, MIMIC's evaluation time increases ***at a far quicker rate*** with respect to both the iterations and the problem sizes I chose. Figure 11 shows the average time over each problem w.r.t iterations/problem size. Note that the problem size graph has a logarithmic y-axis.

To illustrate convergence performance, here are a set of tables that comparatively highlight global discovery optima rates over all problem sizes. The lowest convergence will be 1X, and an algorithm that converges twice as often will be 2X, etc. If an algorithm has 0X convergence, it never found the global optima. Relative rates will be calculated using the least performant, global optima finding, algorithm.

Parity Sum (Deep)

Algorithm	Convergence Rate
RHC	0X
SA	1X
GA	1.45X
MIMIC	1.79X

Parity Sum (Shallow)

Algorithm	Convergence Rate
RHC	0X
SA	1.09X
GA	1X
MIMIC	1.25X

Consecutive Product Sum

Algorithm	Convergence Rate
RHC	1.23X
SA	1X
GA	0X
MIMIC	1.08X

Funky Dot Product

Algorithm	Convergence Rate
RHC	1.07X
SA	0X
GA	1X
MIMIC	1.41X

A final note: due to my limited number of iterations in the fitness w.r.t iterations graphs, we can't see some algorithms converge to more optimal solution. We do see all algorithms converge at least once in the fitness w.r.t problem size graphs, but future work (and more compute power) could highlight fitness-iteration relationships better, either by increasing the total number of iterations tested or experimenting with a larger hyperparameter space.

3 Conclusion

3.1 The Best Algorithm?

There isn't one, according to the no-free-lunch theorem, but I'll try to pick winners for categories.

For the neural network subsection, not only did RHC converge, but it did so remarkably well. This is due to a lack of local optima. The same applies for section 2.3; the consecutive sum problem has few local optima, so RHC does the best again. Things get interesting when there are multiple shallow optima (section 2.2, shallow version): SA follows closely behind MIMIC, converging less often than MIMIC, yet its clock times are FAR quicker. RHC struggles to converge when there are several shallow optimas – in fact, RHC struggles to find a global optima when the search space is even moderately complex.

Comparatively, GA will fail to find a global optima when it cannot crossover and generate better potential maximas – this is seen in the consecutive product sum problem (section 2.3). SA fails to find a global optima when the fitness function is unreliable – as seen in section 2.4 with the Funky Dot Product problem – since it cannot rely on its exploring/exploiting procedure.

For multiple deep optimas (section 2.2, deep version), SA fails to converge, while GA and MIMIC continue doing well. However, MIMIC converges on the optimal solutions more often than GA at the expense of computation time. Finally, with a noisy fitness function (section 2.4), we see the benefits of creating an underlying probability distribution with MIMIC. When our "heuristic" – the fitness function – is unreliable, MIMIC essentially sees through its lies, which is pretty fascinating!

Annoyingly, I couldn't break MIMIC. It was the best performing algorithm throughout, except when the global optima is glaringly obvious (section 2.3, consecutive product-sum problem). However, it comes with a serious cost – speed.

To conclude, the problem space and time limitations dictate the best algorithm.

References

Hayes, G. *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python.* `https://github.com/gkhayes/mlrose`. Accessed: 22 February 2019, 2019.