Assignment 2:
Randomized Optimization

Oindril Dutta
odutta3@gatech.edu

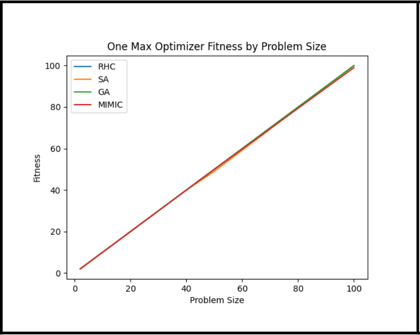
1 COMPARING RANDOM OPTIMIZATION ON DISCRETE FITNESS PROBLEMS

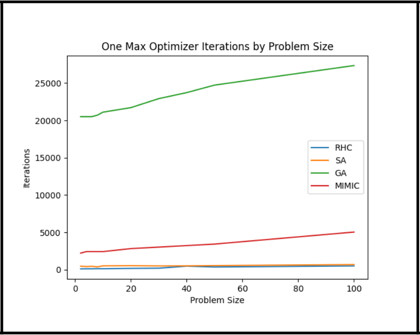
The goal of this section is to implement and compare a few simple discrete fitness optimization problems that can each be solved by random optimizers, namely random hill-climbing (RHC), simulated annealing (SA), genetic algorithms (GA), and mimic (MIMIC) - which I found out stands for Mutual - Information - Maximizing Input Clustering! Very fascinating. I chose MLRose as the primary library of choice for this assignment, and picked a few problems already implemented in the library that I think will do a good job showcasing the strengths and weaknesses of these optimizers - I picked one-max, flip-flop, and knapsack.

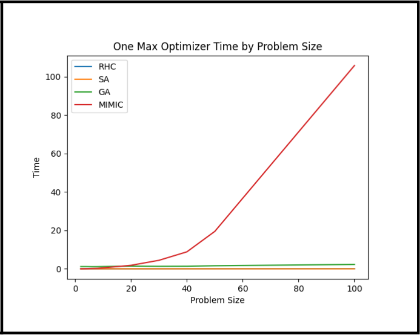
I started by simply implementing all the fitness problems with all the random optimizers and then did the same tuning across all problems to have consistent tuning across all the problems. I made sure to have each of the problems have variable problem sizes that I could use to uniformly compare performance over all the optimizers. Namely, for one-max and flip-flop, I did the length of bits, and for the knapsack, I did the number of items that can be picked to fit in the bag. To compare all the optimizers, I will see the best possible fitness and time/iterations vs sample size, namely the optimizer with the highest fitness and lowest time/iterations will be the "best". I also applied a random seed for all the algorithms based on my ID in the main functions of each file to make analysis easier.

1.1 One Max

The one max is a very simple problem when it comes to writing an algorithm to solve it - simply populate the entire space with 1s. The fitness function is based on how many 1s there are, and then sums it up, irrespective of order. I'm hoping to see RHC & SA's strengths in this algorithm because there are no local optima - every point should have at least one neighbor that's greater than itself, other than the global optimum.



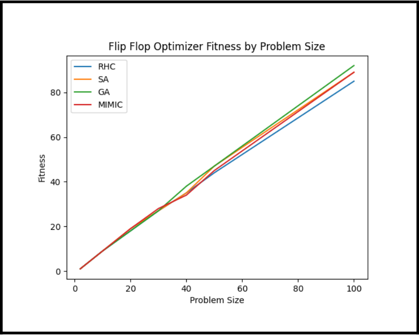


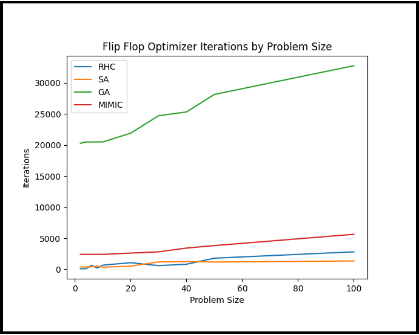


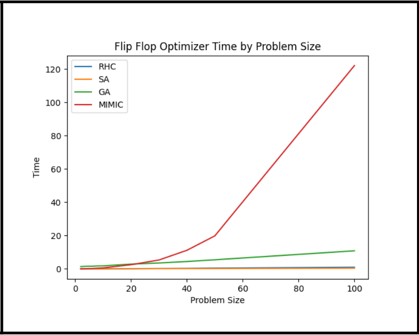
As expected, all the algorithms can easily figure out a good solution to One Max, as indicated by the fitness vs problem size on the left, where they're all effectively linear lines showing they all managed to get to the state where the sum is all the elements of the array. What's interesting here is that RHC and SA are showing their superiority by basically being constant time algorithms in terms of iterations and wall clock time when it comes to finding a solution regardless of the problem size, which GA & MIMIC are struggling with. As can be observed, while GA needs a lot of iterations, it doesn't take that much wall clock time, but MIMIC needs more wall clock time.

1.2 Flip Flop

Flip-Flop is interesting. It's also a very simple problem - simply alternate 1s & 0s the entire space, and it will be optimized since the fitness function sums the number of alternations between 0 and 1. However when it comes to running a random search optimizer algorithm, this isn't as clear when it comes to the neighbor direction to head towards, since a lot of neighbors can increase the fitness, especially if the problem size is large, but it may not be the right direction to go in, leading to lots of possible local optima. For example, going from 01111 to 01011 and 01101 has the same change in fitness (+2) but the latter isn't going to lead to the optima without first lowering the fitness. While RHC and SA can do that through randomness (randomly jumping to a new point, random restarts, high-temperature SA) I feel like GA & MIMIC should shine here. MIMIC obviously because it should be able to create a chain dependency tree structure that leads it towards the right answer every iteration. GA because while it doesn't have the same type of "memory" as MIMIC, given a high enough population and mutation, it should have future generations with points like 01111 allowing it to find its way to the global optimum.



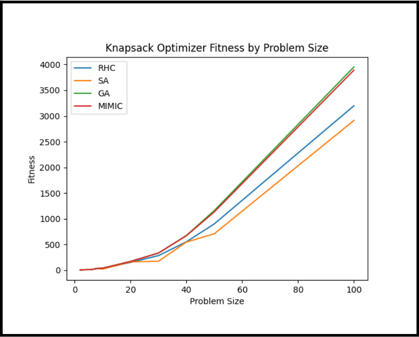


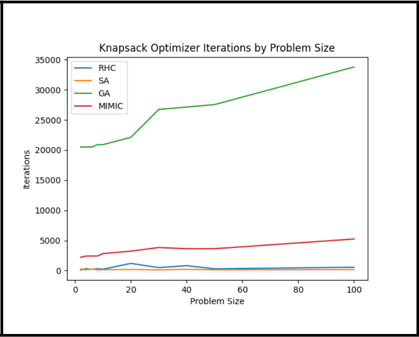


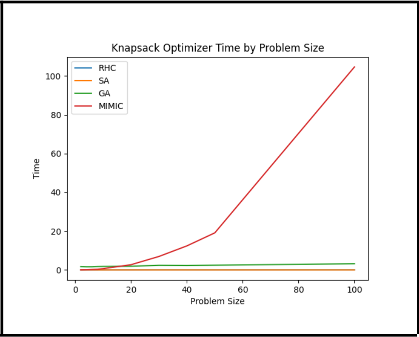
In terms of best fitness for time, GA is the clear winner here. It maximized the best across all the algorithms, and while it had a lot of iterations, it didn't take that much extra wall clock time, especially when compared to MIMIC. MIMIC does come in 2nd place here in terms of fitness because RHC & SA are at the bottom of the fitness graph, but clearly, this problem is too simple for MIMIC to not have runaway wall-clock time.

1.3 Knapsack

The knapsack problem is also really interesting. While it's not a simple problem - given a set of items' weights and values, pick the items that maximize the total value of the knapsack, the best way to solve this is dynamic programming - I feel like creating a structure can successfully solve this problem optimally. For that, I feel like MIMIC should be the best algorithm here, bypassing the others with a chain dependency tree structure that can represent internally an analogous dynamic programming structure being built.







While this was supposed to show MIMICs superiority, GA barely beats the fitness MIMIC approaches, and perhaps even worse it takes so much more wall-clock time than any of the other algorithms. I don't know if any more tuning could help MIMIC in this case, however, since when I tried to do it, it only worsened wall clock time. A larger problem space might help better illustrate these differences, but it's just simply too much wall clock time.

2 NEURAL NETWORK RANDOM OPTIMIZATION: YOGA POSES DATASET

For Assignment 1, I used a dataset of Yoga Poses - I'm going to reuse that dataset. Around 2000 rows of angles & magnitudes of human body joints doing yoga poses, labeled as either Warrior II, Triangle, Tree, or None. The closest analogy of this data set I can draw is the Iris data set, which I used as an inspiration to create this simple dataset. Both datasets' features are the physical measurements of the thing it's trying to classify. For example, the iris dataset measures petal count & lengths, I measure joint angles and magnitudes lengths.

I hand-generated this dataset, by inputting in youtube URLs along with timestamps and a label to a server I made. Each frame within those timestamps was turned into a frame, cropped and scaled down to 100x100 images, run through OpenPose, that outputted a point map skeleton of the joints, which I then converted to angles and magnitudes, used as the features. To see the source/data itself, look at the JSON file in the yoga folder after cloning my repo - look for the URL under the objects in the history key, to see the videos I captured frames from. To see the frames themselves, you can decode the base64 string images in the objects in the frame key.

The first 12 features of the data are the angles at the joints of the human body, such as the angle between the forearm and the upper arm at the elbow. The last 12 features of the data are the magnitude lengths between the center of the body and each joint of the body, which I calculated by getting the center of mass and then measuring the euclidean distances to the 12 joints. The labels are 0 - 3, with 0 as None (no pose, random pose) with 900 rows ~43%, 1 as Warrior II pose with 360 rows ~17%, 2 as the triangle pose with 343 rows ~16%, and 3 as tree pose with 459 rows ~24%. If you google these poses, you'll see I picked them because all of the joints of the human body in the pose are visible in 2 dimensions from a webcam.

It's a pretty interesting dataset since I know when I used it in the past I could achieve 99% accuracy with a fully connected neural network, but I've had some difficulty repeating that in Assignment 1. For this assignment, for use in the NN, I have an 80-20 train test split of the ~2000 rows after shuffling all the rows across the training and testing set to ensure an equal split. For cross-validation, I don't see any options for that in MLRose, otherwise, I would've done another 20% of the training set. MLRose also required one-hot encoding of the labels.

2.1 Neural Network Implementation & Tuning

I'm using the Neural Network implementation in MLRose. My structure is very simple, two hidden layers of 48 nodes each - 2 * number of features. For the methodology, I'll be tuning each algorithm as best as possible to see what I can get out of them. MLRose has made it so that I can't share parameters such as learning rate across algorithms due to how it's written.

One note on tuning - while I was trying out different activation functions, I saw that some did significantly better - much higher accuracy and less loss and fewer iterations - for different algorithms. What I found was that TanH fit RHC and SA well, while ReLU fit GA well, and of course Gradient Descent stuck to sigmoid. I believe this is because RHC and SA are both very similar randomized search optimizations, lending themselves to the same activation function, but GA is not like RHC/SA but also not like Gradient Descent, doing well on a different one.

In general, the main three knobs I have for tuning for all algorithms are learning rate, max # of iterations, and max # of attempts. Of these three, the main one to determine is learning rate, since increasing or decreasing it can be good or bad for performance. So to tune, I will keep increasing attempts/iterations if they keep improving performance, and then once that plateaus, determine the best learning rate. After I do that, if there are other knobs, such as for RHC / SA / GA, I will play with them also on high attempts/iterations until I don't think the results can be better, and then lower the attempts/iterations maintaining the same results, to get the best performance for time possible. I tested "smaller" and "bigger" numbers in orders of magnitude.

2.1.1 Gradient Descent - Backpropagation Baseline

Following the structure I outlined before, I increased the iterations and attempts a lot, but after 10/10000 attempts/iterations, I only got increases of ~1% after already getting to ~90%+ accuracy. It took around ~20 seconds on average to train. Learning Rate tuning landed me at 0.001, going any smaller or higher significantly decreased performance. This is the best tuning to get the training/testing accuracies above 90% the quickest.

2.1.2 Random Hill Climbing Results

I started RHC with 1 restart, thinking if performance was iffy, I could just increase the number of restarts to see if it did better. I started the same process as above with a learning rate of 0.1, increasing the iterations and then the attempts until performance plateaued. I settled on 100/10000 attempts/iterations, which for my seed gave me ~90%+ accuracy already. I then tuned the learning rate from 0.1 up and down.

I observed that smaller learning rates lead to worse accuracies, for example, 0.01 or 0.001 or 0.001 resulted in worse accuracies (< 50%), which makes sense because the baseline NN training final weights were large-ish numbers - I should've normalized the data - which leads me to question why the NN Baseline did so well with tiny 0.001 learning rates, which lowered performance below 50% at learning rates any higher. I read on ED that this is a quirk of MLRose over something inherent about these algorithms, so I'll move on.

Increasing the learning rate to 1, 10, 100, 1000, and more leads to higher accuracies, plateauing at 100 and then worsening performance at 1000. It seems like the sweet spot for learning rate was 100 for accuracy and loss, but the number of iterations and total time did keep going down as I increased the learning rate, minimally. The best time I saw was 77 seconds at 98% accuracy and 0.0456 Loss for a learning rate of 100000. I didn’t bother tuning the restarts since I already had gotten such good results, and I couldn't speed it up more.

2.1.3 Simulated Annealing Results

SA was very interesting. I could continuously keep increasing the attempts/iterations and get better accuracies, without any tuning on the learning rate or the schedule. Following suit from RHC, I decided to start the learning rate at 100. On one run, using LR 100, 100000/100000 attempts/iterations, I got ~98%+ training and testing accuracy, but it took 737 seconds. However it seemed like the learning rate mattered less for SA compared to RHC, because I also had a run, using LR 0.01, 100000/100000 attempts/iterations, and I also got ~90%+ training and testing accuracy there, but it took 1091.54 seconds, much longer, regardless, both results were unreasonable for tuning, so I lowered the numbers.

I decided to focus on tuning the learning rate with 10000/10000 attempts/iterations, and I got ~90%+ training and testing accuracy with any learning rate greater than 0.01, below which I got poor accuracies < 50%. However, as I noted before, SA is interesting because even with a tiny learning rate, with high enough attempts/iterations. I could still get good performance, which was not true for RHC. I kept trying higher learning rates until I found a plateau at the LR of 100, the same as RHC. At this point, this ideal LR for RHC & SA might have something to do with my non-normalized dataset more than anything else, but let’s see how GA does! It might also be because they're very related.

After I reached 100 LC, I decided to cut down on the 10000/10000 attempts/iterations. I couldn’t cut down at all on the iterations, as it would dramatically drop accuracy, so I settled on 100/10000 attempts/iterations with 100 LR. I got some pretty good results with this. The next thing to try was tuning the exponential constant of decay, and it gave me some interesting results! It made the algorithm faster, halving the time when I made the decay faster: 0.001 -> 0.01 -> 0.1, however it sacrificed the results a little, very marginally.

In general, the more time SA was given in iterations, and the higher the learning rate, the faster it could converge on a really good solution. It’s still slower than Gradient Descent but on par with RHC. However, SA’s real strength is control of the temperature decay, which can be tuned to balance speed (faster decay) and performance (slower decay) - which allows it to beat RHC.

2.1.4 Genetic Algorithms Results

GA was probably the most annoying algorithm for training. Every training session took a lot of time per run, and I wasn't able to achieve ~90%+ accuracy at all. Regardless of the size of attempts/iterations or LR that I threw at the network, it always returned the same values, as if none of the parameters mattered. I figured there must be two reasons for this. #1 - I set a random seed, so it always returns the same values. #2 because the algorithm was already doing the best it could, at least based on its current config. Population Size and Mutation Probability would need to be tuned - probably why I wasn't getting better results. To tune them, I decided to do a grid search on 100/10 attempts/iterations with an LR of 1 to find the best combination of population size and mutation rate. I got these results:

200 |0.1: 70.95% train, 69.73% test accuracy.
200 |0.4: 65.13% train, 60.53% test accuracy
200 |0.7: 71.19% train, 69.73% test accuracy
200 |1.0: 64.70% train, 72.39% test accuracy
466 |0.1: 69.07% train, 70.46% test accuracy
466 |0.4: 67.49% train, 64.89% test accuracy
466 |0.7: 69.19% train, 67.55% test accuracy
466 |1.0: 68.89% train, 70.21% test accuracy
733 |0.1: 76.04% train, 77.96% test accuracy
733 |0.4: 72.65% train, 69.97% test accuracy
733 |0.7: 71.49% train, 71.42% test accuracy
733 |1.0: 70.70% train, 73.60% test accuracy
1000|0.1: 72.40% train, 70.70% test accuracy
1000|0.4: 73.07% train, 70.46% test accuracy
1000|0.7: 71.74% train, 72.15% test accuracy
1000|1.0: 75.31% train, 78.20% test accuracy

As you can see from the data above, unfortunately, this search wasn't very fruitful, because nothing really broke 80%. I picked the best one from here, 733 pop and 10% mutation chance, and ran it with 100/1000 attempts/iterations and LR 0.1.

I don't know why GA would fail so spectacularly, especially when compared to the other ones, like RHC and SA. It can't be the data, since all the other algorithms are able to do a pretty good job on the data. It could be the seed, but I tried other seeds and removed the seed restriction and didn't get any better numbers. It could be tuning still, but I feel like I've exhausted all the options I have on that front since I've grid searched the attempts/iterations and LR, and the GA-specific hyperparameters. It could be MLRose, it didn't seem to do the best job with the baseline backpropagation gradient descent. Lastly, it could be that GA is a terrible fit for NNs. The biggest contributing factor to that would probably be how pairs are merged to create new ones. Unlike SA and GA that can just check if the next step's fitness is better, GA is at the mercy of the pairing algorithm to properly mix features to move towards better fitness. While the algorithm should keep picking the fittest points of a population, the mutation probability or the fact that a child generation can be worse than the parents may be actively limiting how well GA performs in this context. SA and RHC might also be closer to Gradient Descent conceptually, while GA must be further. Any of these reasons above could have contributed to GA's overall poor performance.

2.2 Overall Comparison & Conclusion

	Time	Train Accuracy	Test Accuracy	Loss	Iterations
RHC	57.68s	99.09%	98.54%	0.0398	10663
SA	45.64s	98.60%	97.82%	0.0566	13305
GA	918.88s	76.77%	77.96%	8.0220	147218
Baseline	17.88s	94.23%	93.94%	0.1930	N/A

My understanding of all these different algorithms is that it's completely a matter of time. Depending on the context/problem and size of the network, any of these algorithms can find the most optimal weights, but - especially as Model sizes grow; mine is only a meager 24*48+48*48+48*4 = 3648 weights now - I think Gradient Descent backpropagation becomes the default baseline option purely due to speed. That said, given small enough networks, RHC/SA can match or beat this baseline, at least for accuracy and loss. But even in these scenarios, fewer iterations and time is needed for Gradient Descent to do "good enough", such as 90%+ accuracy.

Having said that, however, alongside the number of weights in my network, I also think the fact that I didn't normalize my Yoga Dataset features probably contributed to how close the algorithms were to each other in terms of accuracy and loss, skewing the learning rate toward abnormally large numbers like 100. I also believe the MLRose implementation of NNs might be slightly inferior and buggy, because I know Gradient Descent can be more accurate and faster on this same dataset for the same size network. It doesn't have any concept of iterations the same way RHC/SA/GA does, allowing it to be faster. To be very terse, Backpropagation Gradient Descent is the best way to train neural networks, specifically as the data and structure get bigger, with normalized data with better libraries.

3 CONTINUATION

I would love to re-do GA NN tuning and figure out where it truly sits in comparison to SA and RHC. I believe the library is to blame since I can't tune anything differently.