Randomized Optimization

CS 7641: Machine Learning Assignment 2

By Divya Paduvalli (GT ID: dpaduvalli3/902913716)

Abstract

In this analysis report, four local random search algorithms have been analyzed namely: Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA), and Mutual Information Maximizing Input Clustering (MIMIC). The report has been divided into two sections. In section one, the first three optimization algorithms were applied to find good weights for a neural network on Adult Census Income data set. In section two, three optimization problem domains namely: OneMax, FlipFlop, and MaxKColor are explored and applied to the four optimization algorithms. The analysis results presented in this report was accomplished using the mlrose library in Python 3.7.

Section 1: Randomized Optimization with Neural Network

Data Pre-Processing

I have used the Adult Census Income data set for this problem. This data set was an extraction from the 1994 census database. There are 14 attributes, 48,861 instances, and 2 classes. The task was to predict whether the income of an adult US citizen is either less than or greater than or equal to $50,000 per year based on attributes such as age, race, education, gender etc. The dataset contained rows with missing values which were removed prior to splitting the data. For mlrose library to work properly, the dataset was loaded in python as a NumPy array. I changed the categorical discrete values to nominal values to make it easier for the algorithms to analyze and process the data. The dataset was then split into 70% training and 30% testing. Cross validation was not performed since this feature is not available in the mlrose library.

Algorithm Parameters Tuned

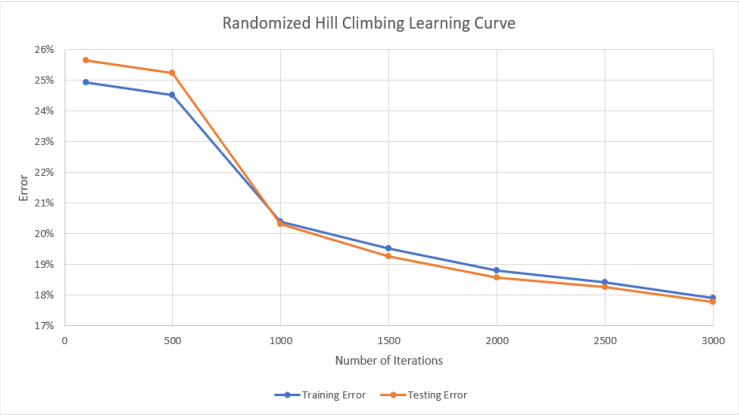
For the first three optimization algorithms, I set the following parameters in mlrose:

- **Hidden Nodes** (hidden_nodes): This is the number of nodes in each hidden layer. I experimented with different hidden nodes and found that anything beyond 4 nodes did not significantly improve the testing accuracy. Hence, I had set my hidden nodes to 4.
- **Activation** (activation): I have selected ReLu activation function because it is the most commonly used activation function in neural network. In addition, it does not activate all the neurons at the same time making the network sparse and hence makes the computation efficient.
- **Max Iterations** (max_iters): Number of iterations used to fit the weights. I experimented with the following iterations: 100, 500, 1500, 2000, 2500, 3000. Going beyond 3000 did not significantly improve the performance.
- **Bias** (bias): I have included a bias term. A bias term helps in shifting the activation function (left or right) which is crucial for successful training.

- **Learning Rate** (learning_rate): It is the step size for optimization algorithms. I have set the learning rate to 0.1 since decreasing beyond this negatively impacts the performance of my algorithms.
- **Max Attempts** (max_attemps): Maximum number of attempts to find a better state. I set this to 100. This number was selected because choosing anything below or above this value did not significantly impact the performance.
- **Early Stoping** (early_stopping): I have set this to true. This stops my algorithm after maximum attempts have been utilized to find a better state.
- **Weight Range** (clip_max): I have set this to 5. This limits the weights to be between the range of -5 to 5.

Randomized Hill Climbing (RHC)

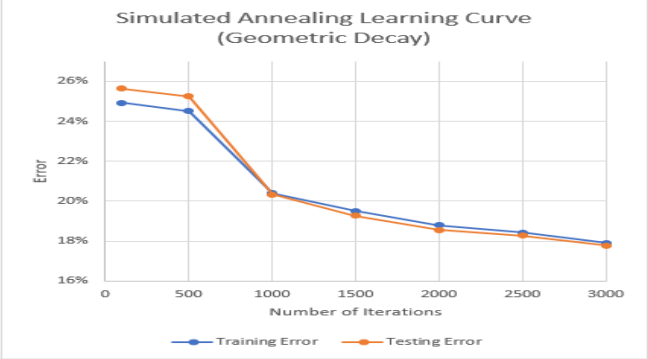
Adult Data Set				
Iterations	Model Build Time (Seconds)	Model Test Time (Seconds)	Training Accuracy	Testing Accuracy
100	1.9343224	0.0044165	75.07697%	74.36181%
500	8.4969161	0.0021548	75.47956%	74.75964%
1000	17.1470346	0.0022802	79.60025%	79.67731%
1500	26.9016783	0.0021703	80.48596%	80.72715%
2000	40.2030418	0.0167251	81.20116%	81.43441%
2500	51.2214139	0.0156457	81.57533%	81.72174%
3000	50.4578691	0.0041475	82.09160%	82.21903%



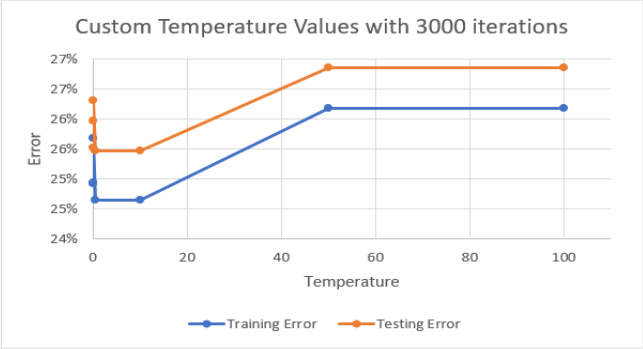
Stochastic hill climbing can get stuck at the local optimum if the problem has too many peaks and sometimes where it gets stuck might not be the best place to be. However, in randomized hill climbing (RHC), once a local optimum is reached, the algorithm will restart the hill climbing at a randomly chosen x. Similar to stochastic hill climbing, it finds the neighbor that has the largest function value and moves in the direction of its neighbors where the function value increases until it has reached the peak. Since this process is randomly repeated multiple times, it increases the chance for the algorithm to reach the global optimum. From the above results, you can clearly see that the accuracy percentage for both training and testing data linearly increased as number of iterations increased. The highest testing accuracy score turned out to be around 82.22%. I was expecting the model's accuracy score to be even greater because the nature of RHC allows it to get multiple tries to find a good starting point to restart the hill climbing process. However, it is also true that since the selection process of weights is random, there is no guarantee that the algorithm could somehow luck into the attraction basin of the global optimum. In a way, the algorithm depends a lot on the size of the attraction basin of the global optimum because if that size is really huge, there is a high probability that the algorithm could luck into that area and get a global optimum. Almost all of the datapoints (the entire space!!) will need to be systematically covered to make a deduction that what we have is indeed the optimal solution. The model's training time increased as the number of iterations increased. However, the model's testing time was very fast because of the size of the dataset used for testing.

Simulated Annealing (SA)

Adult Data Set (Geometric Decay)				
Iterations	Model Build Time (Seconds)	Model Test Time	Training Accuracy	Testing Accuracy
100	2.556444	0.004667	59.00156%	58.12797%
500	13.079957	0.000000	75.35642%	74.52757%
1000	29.015586	0.745276	75.35642%	74.52757%
1500	44.730824	0.770140	77.70094%	77.01403%
2000	52.724122	0.008857	78.29773%	78.13018%
2500	57.112480	0.002468	79.32080%	79.29053%
3000	84.131188	0.003208	80.46701%	80.62769%



Custom Temperature Values with 3000 iterations (Cooling Rate = .05)		
Temperature (T)	Training Accuracy	Testing Accuracy
0.1	75.08170%	74.47232%
0.15	75.05328%	74.03028%
0.2	74.3144%	73.6877%
0.5	75.35168%	74.52757%
10	75.35642%	74.52757%
50	73.81708%	73.14620%
100	73.81708%	73.14620%

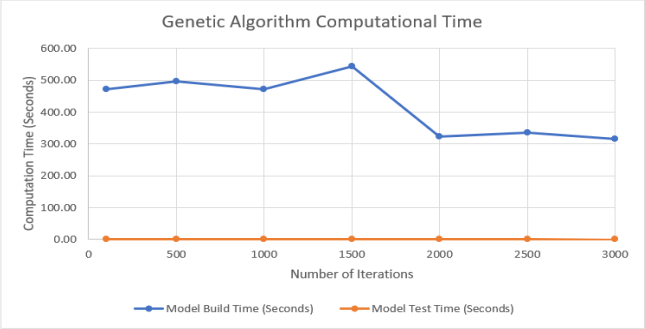


Simulated annealing (SA) uses a combination of exploration and exploitation. In hill climbing, we are always trying to exploit by finding points that gets us to the highest optimum point/peak. However, in exploration, we visit more of the space with the hope of finding the global optimum and we don't just settle for a local optimum. The algorithm checks to see if the new neighboring point is better or worst than the current point. If the neighboring point is better, then we move in that direction. If the current point is better, then the algorithm would calculate the probability to move to the new point based on an acceptance probability function which includes a temperature parameter (T). There is a temperature parameter in mlrose called schedule and the default value for that was set to geometric decay. The geometric decay function takes three parameters namely: initial temperature (default: 1.0), decay (default: .99), and minimum temperature (default: 0.001). The function multiplies the initial temperature with the decay factor (which can be considered as a cooling rate constant) and takes the maximum between that value and the minimum temperature. It decreases the temperature slowly by the decay constant giving the algorithm a chance to explore before cooling it out. The reason why I chose to go with the default values is because when I tried inputting different initial temperature values in geometric decay, the accuracy did not seem to change much. Moreover, the schedule parameter doesn't really show the different temperature values that it uses for different time steps. Based on the above table for geometric decay, the performance of the model increased as the number of iterations increased. Notice that SA takes a lot more iterations when compared to RA to reach to reach its highest accuracy. Due to the cooling factor in the geometric decay, SA did not aggressively search for global optimum and hence was able to avoid some overfitting issues by not prematurely concluding that it reached the global optimum. The model's training time increased as the number of iterations increased. The model's testing time was incredibly fast.

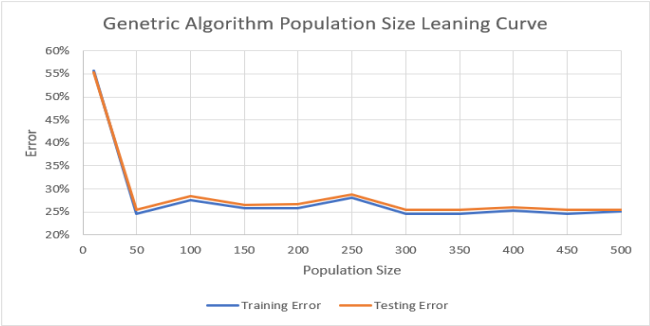
I conducted a separate experiment taking 3000 iterations and manually inputting different temperature and cooling values to see how my model behaves mainly on the property of exploration and exploitation. Based on the custom temperature table and the learning curve, when the temperature is too high it does not notice big valleys and just keeps exploring to the point that it becomes a random walk. This is indicated by the high error values corresponding to the high temperature values. As the temperature starts getting cooler, the boundaries of the valleys become more noticeable and the algorithm can create different basins of attraction for different valleys. The algorithm was given a chance to wander to where the high value points are before cooling it so much that it can't bridge the gulf between different local optimums. The optimal values were between temperatures .5 and 10. It is interesting to notice that the testing accuracy score between .5 and 10 remained the same. I was expecting the testing accuracy to rapidly go down since there is a huge jump from .5 to 10. A possible reason for this behavior could be because of the cooling rate of .05 which was very low. The cooling rate ensures the temperature is gradually increased and hence the difference in accuracy is not too much. This is also indicative of the fact that the algorithm's performance depends a lot on the cooling rate as well.

Genetic Algorithm (GA)

Adult Data Set (Defualt: Population =200 and Mutation Probablity = 0.1)				
Iterations	Model Build Time (Seconds)	Model Test Time (Seconds)	Training Accuracy	Testing Accuracy
100	471.591992	0.015642	74.23388%	73.24566%
500	496.669071	0.015577	74.23388%	73.24566%
1000	471.853964	0.015589	74.23388%	73.24566%
1500	544.849996	0.016764	74.23388%	73.24566%
2000	321.922822	0.004776	74.23388%	73.24566%
2500	335.925047	0.005811	74.23388%	73.24566%
3000	314.81	0.00	74.23388%	73.24566%



Adult Data Set (Iterations = 500, Mutation Probablity = 0.1)				
Population Size	Model Build Time (Seconds)	Model Test Time	Training Accuracy	Testing Accuracy
10	6.76273131	0.00199533	44%	45%
50	29.42222548	0.00000000	75%	75%
100	173.88147187	0.00000000	72%	71%
150	205.88400102	0.00000000	74%	73%
200	114.92065859	0.00199413	74%	73%
250	376.95683670	0.71256492	72%	71%
300	187.35044909	0.00000000	75%	75%
350	224.26620793	0.00199509	75%	75%
400	383.30542350	0.01562047	75%	74%
450	297.12688684	0.00000000	75%	75%
500	292.43448138	0.00199318	75%	75%



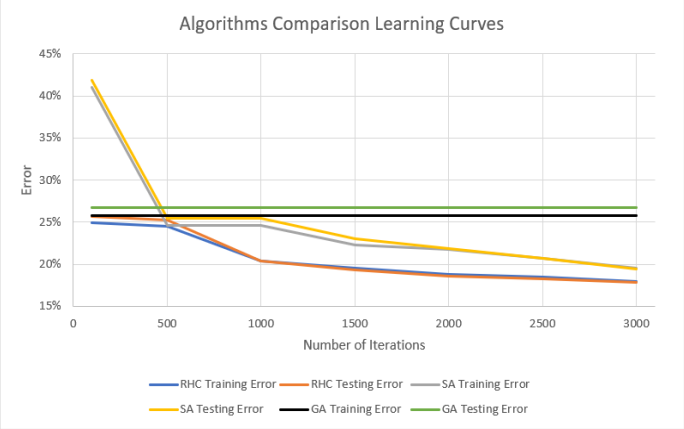
The genetic algorithm (GA) process is derived mainly from biological evolution. The algorithm starts with initial population of individual solutions with size k. The fitness score of all the individuals in the population is computed and the most fit individuals will be selected at random. The most fit individuals will be paired up to be parents to produce offspring using crossover or mutation rules. The crossover rule pair up two parents to create offspring for the next generation. The mutation rule applies random changes to individual parents to create offspring. The new offspring will replace one of the least fit individuals from the population. In mlrose, I went with the default vales for population (200) and mutation probability (0.1). Here mutation probability indicates the probability that each element in the state vector will mutate during reproduction and

is a value between 0 and 1. The reason why I went with the default values is because the values are small, and I wanted to start with a low complexity since I will be running the algorithms on different number of iterations. In general, the computation time was very slow. With just the default values and with only 100 iterations, it took me around 8 minutes to run the algorithm. It is interesting to note that, the algorithm gave the same accuracy (~73.25%) results for different iterations. This indicates that the algorithm converges very quickly in terms of iterations. The model training time did fluctuate a lot between 5 to 8 minutes. As the number of iterations increased the training time increased and then decreased. The reason for this behavior could be because the model has learnt that no matter what the number of iterations is the optimum is still going to be the same and hence when the algorithm had more than 2000 iterations, it required less computation time to yield the same result. Had I increased the iterations to go beyond 3000, the computation time would have still gone down.

I experimented with different population sizes to see how it influences the algorithm's performance. As per the population size table above, increasing the population size causes the number of generations that converge to increase thus increasing the testing accuracy. The population size should typically not be very large because it significantly adds to the computational time of the algorithm. The test accuracy score of ~ 44.62% for a population size of 10 indicates that a smaller population size is also not enough for a good mating pool. The optimal population size in this experiment turned out to be around 300.

Section 1 Conclusion: Algorithms Comparison

Best Performing Algorithms Comparison					
Algorithm	Iterations	Model Build Time (Seconds)	Model Test Time (Seconds)	Training Accuracy	Testing Accuracy
RHC	3000	50.4578691	0.0041475	82.09160%	82.21903%
SA (Geometric Decay)	3000	84.131188	0.003208	80.46701%	80.62769%
GA (Default: Population =200 and Mutation Probability = 0.1)	3000	314.81	0.00	74.23388%	73.24566%



The table above summarizes the results of the best performing algorithms. Note that RHC and SA yielded high accuracy scores as the number of iterations increased (keeping everything else constant). For SA, the accuracy decreased as the temperature parameter was increased. For GA, the number of iterations did not impact the accuracy scores since the scores remained the same, but the computation time was significantly reduced as the iterations increased. The optimal population size for GA turned out to be around 300. Although, GA has the lowest accuracy score, it converges faster in terms of iterations. GA has the highest computation time because of greater complexity involved in the algorithm. Also note that since weights in neural network are continuous and real-valued, each iteration in any of these algorithms was incremented between the values in the range of -5 to 5 (since this is what I had set in the parameters). This was done mainly to decrease the search landscape for the algorithms to find the optimal set of weights.

Section 2: Exploration of Optimization Problem Domains

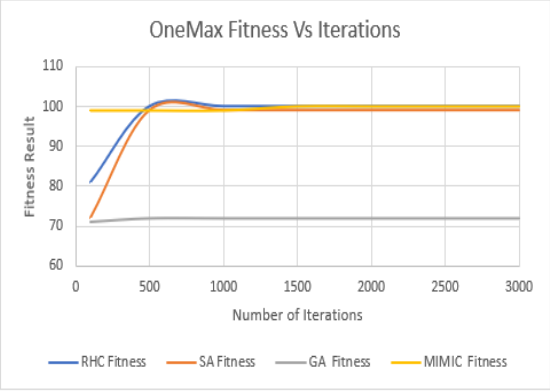
Algorithm Parameters Tuned

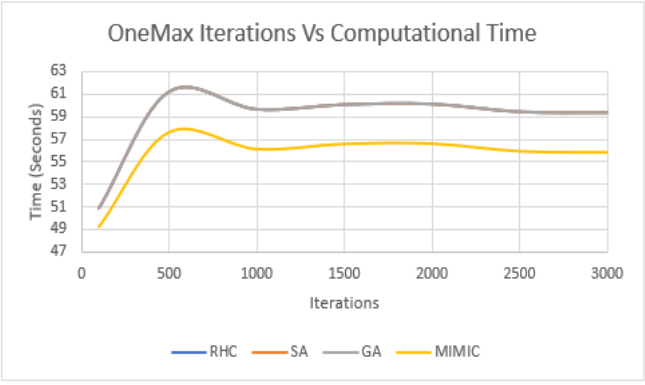
In this section, I have used three optimization problem domains namely: OneMax, FlipFlop, and MaxKColor. I have conducted four experiments to analyze the behavior of the optimization search algorithms:

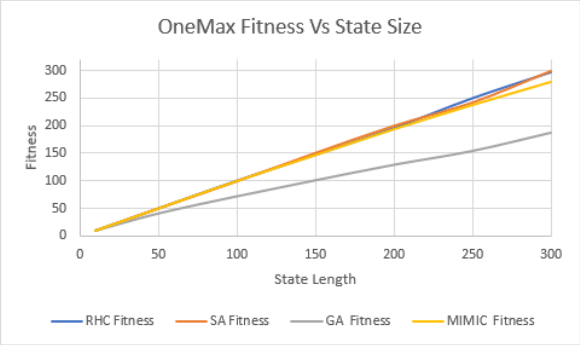
1. **Experiment 1 (Fitness Vs Iterations)**: I kept the length of the state vector to be 100. For the four search algorithms, the parameters were mainly set to default except max attempts was set to 200 and max iterations was set to (100, 500, 1000, 1500, 1500, 2000, 2500, 3000). I stopped at 3000 because going beyond that did not have a significant impact on the fitness values.
2. **Experiment 2 (Iterations Vs Computational Time)**: I kept the length of the state vector to be 100. For the four search algorithms, the parameters were mainly set to default except max attempts was set to 200 and max iterations was set to (100, 500, 1000, 1500, 1500, 2000, 2500, 3000). Python's time library was used to record the building time for every algorithm.
3. **Experiment 3 (Fitness Vs State Length)**: The length of the state vector was set to (10, 50, 100, 150, 200, 250, 300). I stopped at 300 because going beyond that takes a lot of computational time to run the algorithms (around 7 to 8 minutes). For the four search algorithms, the parameters were mainly set to default except max attempts was set to 200 and max iterations was set to 2000.
4. **Experiment 4 (Varying Population Size)**: I tuned the population size for both GA and MIMIC to (10, 50, 100, 200, 250, 300). The iteration was kept to 2000. Other parameters for the two algorithms were kept to default. For MaxKColor, iterations were set to 500 because the algorithms did not take more iterations to converge and hence 500 was sufficient.

***Note: Default settings that were never changed: geometric decay for SA, mutation_prob= 0.1 for GA, and keep_pct = 0.2 for MIMIC.

One Max Results Analysis







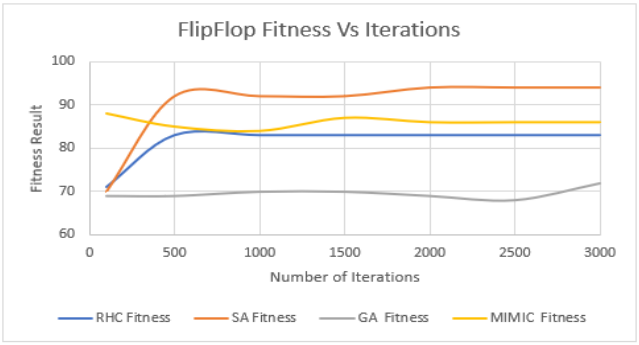
Exp: 4 OneMax (Iterations= 2000 & Length of State = 100)		
Population Size	GA Fitness	MIMIC Fitness
10	66	62
50	69	83
100	72	92
150	71	99
200	74	100
250	72	99
300	72	100

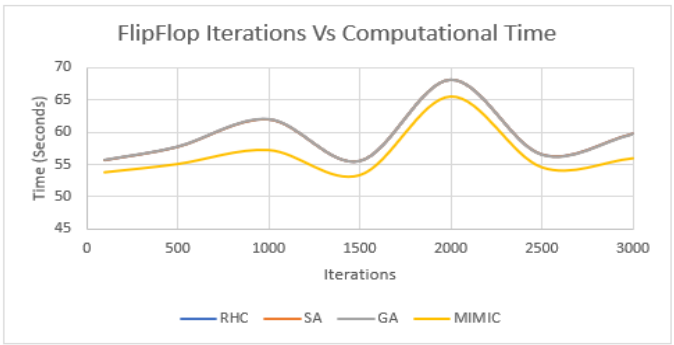
I found OneMax fitness function interesting mainly for its simplicity. Given the obviousness of its easy solution, I wanted to see how each of the search algorithms perform to converge to the optimal solution. In addition, OneMax could also be utilized in a variety of practical applications. For example, you have a small sample of water from a local river and you are specifically trying to examine the amount of contamination of a heavy metal like lead. You take that sample of water to a lab, you feed it to a computer, and you categorize other contaminants as zero and lead as one. Here you are programmatically using a very simple function like OneMax along with other complex algorithms in trying to figure out the total amount of contamination of lead.

The way OneMax works is, given a state vector with n dimensions of bit strings, OneMax evaluates the fitness function of the state vector by searching for all ones and summing up their values in that vector. For example, if x = [0,1,1,1,0], then f(x) = 3.

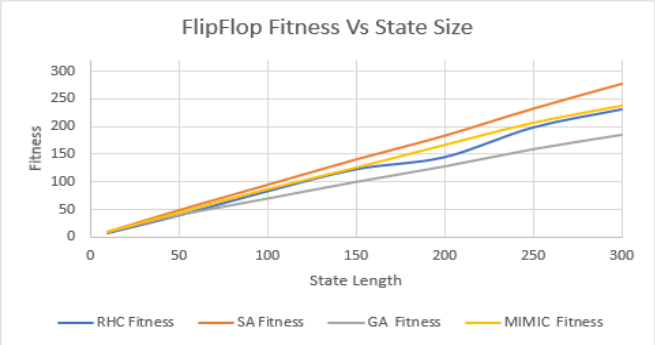
In general, RHC and MIMIC performed the best overall in terms of both number of iterations and computational time utilized. RHC was the first to converge to a global optimum at only 500 iterations and the reason for this is because it gets multiple tries to find a good starting place to find the optimal solution and is the least expensive in terms of computational time used. MIMIC also performed well, although it took around 1500 iterations for it to converge to global optimum. The reason for this is because the algorithm analyzes the global structure of the optimization landscape before prematurely yielding a solution and hence it took quite a few iterations for it make a conclusion about the optimum. SA performed the third best because it utilized exploration and exploitation. I am surprised as to why GA performed the worst because it never converged to an optimal solution. The reason for this behavior could be because GA did not completely recognize or select all the ones in the state vector. Changing the population size also did not seem to influence the performance of GA.

Flip Flop Results Analysis





Exp: 4 FlipFlop (Iterations= 2000 & Length of State = 100)		
Population Size	GA Fitness	MIMIC Fitness
10	68	59
50	67	73
100	72	80
150	70	84
200	69	86
250	69	90
300	71	91

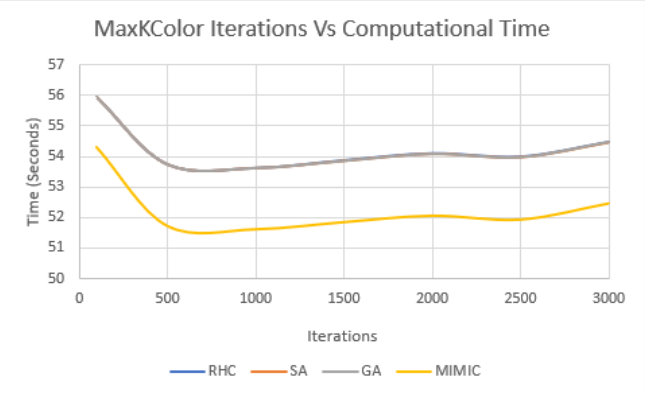


I found the FlipFlop fitness function interesting mainly for its simplicity and for its ability to recognize mismatch patterns. A practical application would be when this function is incorporated with other complex algorithms to recognize patterns in an image. For example, you have a picture of a mountain which was taken at night and you want your algorithm to recognize that. You can assign each pixel in that picture as zero if the pixel has certain shades of black (match) or one if it has other colors (mismatch). If a portion of that picture has a lot of consecutive pixels with mismatched values, then that should indicate that portion has mountains in it and not the dark night sky. Based on the pattern of the Boolean values assigned to each pixel, an algorithm can use FlipFlop's pattern matching functionality to differentiate whether or not that picture was taken at night.

The way FlipFlop fitness function works is, given a state vector with n dimensions of bit strings, FlipFlop evaluates the fitness function of the state vector by recognizing the total number of pairs of consecutive elements that are a mismatch. For example, if x = [0,1,1,1,0], then f(x) = 2.

In general, SA and MIMIC performed the best. SA found the highest fitness to be 94 after 2000 iterations and MIMIC found it to be 88 in just 100 iterations. The reason why SA took 2000 iterations is because of the nature of its algorithm to explore the space first and then slowly cool using the geometric decay factor to exploit and decrease the likelihood of accepting worst solutions. Even when I used different state lengths, SA still outperformed other algorithms. In terms of computational time, MIMIC was the fastest because it only took around 53.835 seconds to converge to a fitness of 88. In experiment four, when I changed different population sizes for GA and MIMIC (keeping the length and iterations constant), the fitness linearly increased. This behavior makes complete sense here because as the population size increase you are capturing everything that is in the state vector and hence performance also increase.

Max-K Color Results Analysis



Exp: 4 MaxKColor (Iterations= 500 & Length of State = 100)		
Population Size	GA Fitness	MIMIC Fitness
10	38	34
50	37	40
100	38	47
150	39	50
200	39	49
250	41	50
300	39	50

I found MaxKColor fitness function interesting for its practical application mainly in the domain of social networking. A practical application would be in detecting friends who are connected on social media who share the same interests.

The way the fitness function works is it assigns a color to every node in a network. The state vector contains the list of colors assigned to every node. If a node of one edge shares the same color as the node of another edge, then the fitness function counts the pair of connected nodes with similar colors as one. For example, if you have the following edge = [(0, 1), (0, 2), (0, 4), (1, 3), (2, 0), (2, 3), (3, 4)] and the following state vector (colors assigned to the five nodes) = [0, 1, 0, 1, 1], then you have 3 pairs of edges which share similar color. These edges are [(1, 3), (0, 2), (4, 3)].

In general, MIMIC, RHC, and SA performed the best. MIMIC was able to find the highest fitness with just 100 iterations. SA and RHC took 500 iterations. I set the number of nodes in my experiment to be 100 which is also the state of the input vector that specifies colors for each node. Since the task at hand is to find edges that has nodes with the same color, RHC just had to randomly pick several edges and verify its colors and hence computationally it took RHC just 53.739 seconds to come up with an optimal solution. It was interesting to see that MIMIC in its first 100 iterations predicted a fitness of 50 but in subsequent iterations it kept predicting a fitness of 49. The reason for this could be, after analyzing the optimization landscape and after several iterations and attempts, MIMIC realized that it might have prematurely picked a fitness of 50 (due to overfitting) but in actuality the global optimum could just be 49. GA yet again performed poorly with having just a highest fitness score of 39. The reason for this could be because of the nature of the fitness function. Typically, in GA you find values with the highest fitness score and produce offspring. Here GA had a trouble finding nodes that shared a similar color and hence performed poorly. Using different population sizes did not seem to have much of an effect on GA, although for MIMIC the fitness increased as population increased.

Section 2 Conclusion: Fitness Functions Comparison

Fitness Function and Algorithms Comparison					
		RHC	SA	GA	MIMIC
OneMax (State size =100)	Highest Fitness Achieved	100	99	72	100
	Computational Time Consumed (Sec)	61.201	61.191	61.178	56.582
FlipFlop (State size =100)	Highest Fitness Achieved	83	94	72	88
	Computational Time Consumed (Sec)	57.776	68.156	59.654	53.836
Max-K Color (State size =100)	Highest Fitness Achieved	50	50	39	50
	Computational Time Consumed (Sec)	53.739	53.739	55.940	54.313

The above table summarizes the best performing algorithms for each fitness function. Overall, GA performed poorly using all the three fitness functions. RHC and MIMIC performed best using OneMax and MaxKColor. SA performed best using FlipFlop and MaxKColor. In terms of complexity, I would characterize MaxKColor as having more complexity when compared to the rest of the fitness functions. If you notice in the table above, as the complexity of the fitness function increases, performance of the three search algorithms decreases significantly. This is especially noticeable for GA. In terms of computational time, MIMIC was the fastest when compared to the rest.

Sources

1. http://archive.ics.uci.edu/ml/index.php
2. https://mlrose.readthedocs.io/en/stable/source/intro.html
3. https://pypi.org/project/mlrose/
4. https://www.cc.gatech.edu/~isbell/papers/isbell-mimic-nips-1997.pdf