CS7641 A2

Randomized Optimization

Wei En Chen

***Abstract* – This paper discusses 4 different randomized optimization algorithms (randomized hill climb, simulated annealing, genetic algorithm, MIMIC) on 3 different optimization problems (fourpeaks, k-color, flipflop). Each algorithm's strength and weakness is presented. The first 3 algorithms are then applied on the neural network from Assignment 1. The algorithms with finding neighbours are more suitable for NN, while genetic algorithm was unable to find a good set of weights.**

I. INTRODUCTION

In this assignment, four random search algorithms, randomized hill climbing, simulated annealing, genetic algorithm, and MIMIC are applied on three optimization problems. I chose fourpeaks, flipflop, and k-colors. All four algorithms will attempt to optimize (maximize) the three fitness functions, and compare/contrast their performance.

In addition, the first three algorithms (excluding MIMIC) are used to replace gradient descent (back propagation) in a neural network, which has the same structure as the one in my first assignment. The performance of all three algorithms will be compared and contrasted. Since the majority of the code package is mlrose [1], which is different from sklearn [2] in assignment 1, there are fewer hyperparameters to tune in neural network models with mlrose and the model will not be 100% identical. To have a fair comparison between gradient descent and the other three algorithm, gradient descent from mlrose will be also included in the analysis.

II. FITNESS FUNCTIONS

The three fitness functions are introduced and discussed here to gain further understanding of each function and form hypotheses on how they may affect the optimization algorithms. All three fitness functions take bit strings as inputs and evaluate them with some specific equations.

FOURPEAKS

The fitness function is written as [3]:

$$Fit(x,T) = max(tail(0,x), \text{head}(1,x)) + \text{R}(x,T) \qquad (1)$$

The fitness function basically counts leading 1's and trailing 0's in x (bit string), takes the max and add the R term which is equal to length of the bit string if the leading 1's and trailing 0's are both greater than T, threshold, and 0 otherwise. From the function we can come to the conclusion that for a bit string which will maximize the value, it will need to have either continuous leading 1's or trailing 0's and the shorter of the two should be bigger than the threshold to get extra value. This means there are two global maxima where longer leading 1's plus the R term, and the other being longer trailing 0's plus the R term. There are also two local maxima where the bit string is either all 1's or all 0's. This is an interesting function because while it is quite intuitive for humans to see where the global maxima are, algorithms which can get stuck in local maxima may not do well here such as randomized hill climbing or simulated annealing. The inherent structure or dependency of the bits is either leading 1's or trailing 0's need to be as long as possible while shorter of the two is above threshold.

K-COLOR

The bit string (state vector) in this case represents colors (binary colors) as 0's and 1's in each node. There is another input called edge which is a list of pairs of connected nodes by their indices in the state vector. If the two nodes connected by an edge are different colors, it counts as score of 1. If the nodes are the same color, the edge scores 0 [3].

The global maxima is theoretically the number of pairs of connected edges if they do not form a closed loop with odd number of edges. There can be multiple global maxima, if some nodes are not connected, or nodes are connected in a way that changing a node's color gains score on one edge but loses score on another edge, which cancels out. This also means there can be plenty of plateaus along the way. These can be local maxima but they are less pronounced than FourPeaks. This characteristic is interesting to see how the algorithms behave. It is also interesting that the relationships between two nodes are not from their positions in the state vector, but from the edge list. One can visualize k-color as a tree easily, meaning there are parents/children dependencies. Algorithms that do attempt to convey structures, such as genetic algorithm or MIMIC should do better than others.

FLIPFLOP

FlipFlop can be treated as a special case of k-color problem. The fitness is evaluated as the number of consecutive pair of nodes in the state vector when they are different [3]. This is essentially saying the edges are formed between adjacent nodes in the state vector. There is one global maximum, which is the length of the state vector minus 1. There are can local maxima (or plateaus, if you will), which occurs when alternating 0's and 1's propagate from two or more locations in the state vector. Below is an example:

$$[0,1,0,1,1,0,1,0,1,0]$$

The state vector consists of 10 bits, and most of the adjacent pairs are alternating except 4th and 5th bits. When this happens, some algorithms can struggle because updating 4th bit does not increase the fitness. It takes all 1st to 4th bits to flip to get to global maximum. If the state vector consists of 100 bits, and the non-alternating pair of bits is in the middle, it will take about half the state vector length in iterations (for some algorithms) to get to global maximum and the fitness does not change during these iterations. They could even return that they have found an optimal state vector before reaching global maximum. None of the three algorithms knows the inherent structure of the fitness functions, but some algorithms do attempt to model dependencies may be able to take advantage and excel at this optimization problem.

III. ALGORITHMS

RANDOMIZED HILL CLIMBING (RHC)

This algorithm is similar to regular hill climbing, but at each iteration, the algorithm will find a random neighbour state by flipping one of the bits in the state vector. If the new fitness is higher, it will go to the neighbour state, if the neighbour state is worse in fitness, algorithm will iterate until max attempt is reached. Once it's reached, it will terminate and return the best state vector [4, 5].

This algorithm should work great on problems with fewer local maxima which will have lower chance of getting stuck at sub-optimal solution. It assumes neighbours are defined as states with any one bit difference. It can work well in continuous state space and some discrete state space, but it may not be suitable for discrete state vectors don't align well with the neighbour state definition. For example, FourPeaks rely on either leading 1's or trailing 0's so if a bit is flipped to cut the 1's or 0's short, the fitness will dramatically reduce and it's not considered a "neighbour".

SIMULATED ANNEALING (SA)

This algorithm is similar to RHC such that in each iteration, it will search for a neighbour by randomly flipping one bit from the state vector. If the new state has a better fit, it will jump to the new state. However, it also has another condition that at certain probability the state will jump to the new state regardless of the fit of the new state. The probability usually decays from high to low throughout the fitting process [4]. The term annealing is taken from a heat treating process to increase a material's ductility by carefully controlling the cooling rate. The cooling rate here is analogous to the probability in the algorithm.

This algorithm should typically perform better than RHC since it is easier for RHC to get stuck in local maxima because it has no mechanism to escape local maxima. On the other hand, SA will have some probability to jump to new states even if the new state has a lower fit. This provides the algorithm some mechanism to get out of local maxima. However, the mechanism is dependent on some probability and also the definition of "neighbour", it is not guaranteed to always find the global maxima. Instead it would have a higher chance of find one compared to RHC.

GENETRIC ALGORITHM (GA)

GA randomly selects two states as parents from a specified population such that if a state has higher fitness, it has higher chance of being selected, then it randomly selects leading bits from one parent, and the rest bits from the other, put them together to form a new child state. Next the child state is mutated by a specific probability on every bit individually. This process is repeated until a specified population of children states are generated. If the best fit from the children population is better than the current fit, the state vector will move to the new state.

We can immediately see the power of GA that states with higher fitness will be likely selected, and their states are passed down to new proposed states, and new states can be mutated at multiple bit locations. This will increase the likelihood of escaping from local maxima compared to RHC or SA. This works well with optimization problems that have discrete state spaces and implicit neighbour bit structures (i.e. leading bits and trailing bits are passed down from parent states). It may struggle on states with continuous state space because the size of the step to change each element in the state vector can be different. Each new child state could simply have no effect or miss the global maxima. Another potential weakness of GA is the run time. The success of GA can depend on the size of the population. More offspring increase the chance of finding better new state. It also leads to more computation time.

MIMIC

This algorithm is similar to GA in the way that it samples some specified quantity from a population using each member's currently defined probability distribution, and then keep the best fit samples with some specified percentile threshold. Next, establish dependencies or structure among the bits from the kept subset of the population by maximizing the sum of mutual information of each parent-child pair bits. The result is a minimum spanning tree and probability distribution can be updated for sampling again. This iterates until an optimal fitness is found [4, 5].

This version of MIMIC utilizes a dependency tree where each bit (or node) has most one parent[4, 5]. This will be very useful for optimization when the problem has some inherent structure or dependency. For example in k-color problem, each node should have different color from connected nodes to maximize fitness. In four peaks problem, however, there are more complicated dependencies than simple parent-child connections. Since the MIMIC used here does not support more complicated dependencies, it may have some trouble finding the optimal fitness, or take longer iterations. Another drawback of MIMIC is the wall clock time. In each iteration it needs to go over all samples it keeps to update the probability distribution. For longer problem size, it likely will need more samples to estimate a better probability distribution. This will be discussed further in the analysis.

IV. ANALYSES

In this section each optimization problem (fitness function) is inserted into each algorithm to understand how well the algorithms perform. Each optimization from start to finish is repeated by a number of times for a few reasons. 1. There are randomness in both algorithms and problems so having some average values over a number of runs is a better way to illustrate. 2. The problem is in discrete state space so output fitness curves will look like stairs. With multiple curves in a plot, they can cross over each other which is harder to compare and contrast differences in a wholistic way. Having multiple runs will smoothen the curves.

For number of attempts, it is set to some high number such that equilibrium is reached, whether global maxima is found or stuck at local maxima.

FOURPEAKS

Each algorithm's hyperparameters are discussed to understand algorithms' strength and weakness. As mentioned earlier, this problem has more complicated dependencies among bits, as well as multiple global and local maxima. This will be more difficult for algorithms to find the global maxima.

Threshold

There are two parameters to adjust the problem's difficulty. One is the problem size, which is common in all three optimization problems, the other is the threshold, unique in four peaks. Having threshold smaller than half but also close to half, it is generally more difficult for optimization algorithm to jump out of local maxima to reach global maxima. For the purpose of this paper, threshold of 0.1 and data length (problem size) of 50 are used, unless the analysis is on problem size, which will be varied.

GA - Mutation Prob

Usually lower mutation probability in GA means there is not much difference between parents and children, meaning it can take longer to find optimal fitness. This is the same behaviour we observe in the result from the runs, shown in left of Figure 1. At mutation = 0.1, it takes close to 300 iterations to converge to global maxima of 94. In fact it does not reach 94. Instead it went to 93, same as mutation = 0.4. If number of iterations is increased, it is likely they eventually go to 94. Higher mutation probability means more different (as in more bits are flipped) samples are generated to escape from local maxima.

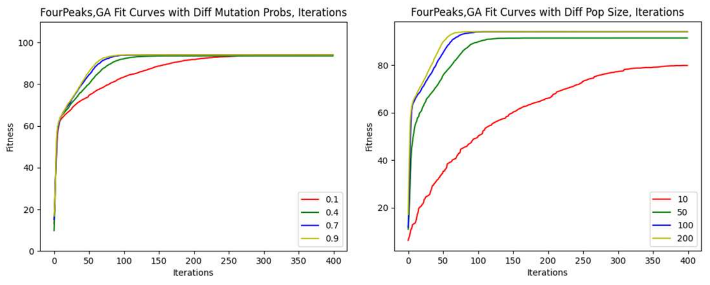

Figure 1 – Left: FourPeaks, GA fit curves with various mutation prob vs iterations; Right: with various population size vs iterations

GA - Population Size

This is a similar concept as mutation probability that more different (in this case quantity) samples will have higher chance of finding global maxima. Population size and mutation do affect each other. If mutation probability is high but population size is low, it can still struggle trying to find global maxima. The exact behaviour is shown in right of Figure 1. All curves have mutation probability of 0.9. When population is below 100, GA has difficulty finding global maxima with the given iterations. It is still impressive that GA is able to find the global maxima given the difficulty of the problem. The downside here is the computation time. From Figure 2, one can see that the wall clock time is proportional to the population size.

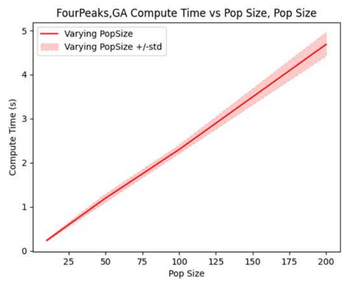

Figure 2 - FourPeaks, GA compute time vs population size

RHC - Restarts

There is only one hyperparameter to tune in RHC, which is number of restarts. Intuitively, more restarts will allow the algorithm to start over from a random state, and potentially can find a better solution if not achieved previously. We can definitely observe this trend in Figure 3. The curves range from 0 to 30 restarts. The solid lines are the mean values over 100 runs. The shaded areas are +/- one standard deviation. On average the fitness does improve with higher number of restarts, but RHC does not reliably reach global maxima. This is due to the multiple local maxima in four peaks problem, and RHC is easier to get stuck in local maxima due to the way it defines a neighbour state. More restarts also leads to more computation time. It is not plotted but it is very intuitive that more restarts means the size of the loop is bigger for the code to complete.

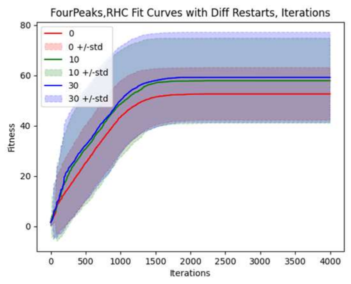

Figure 3 - FourPeaks, RHC fit curves w/ various restarts vs iterations

SA - Decay Schedule Types

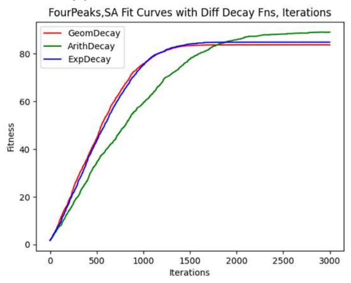

Figure 4 - FourPeaks, SA fit curves w/ various decay fns vs iterations

SA requires a decay schedule that lowers the probability of jumping to a random state over time. There are three types provided, GeomDecay, ExpDecay, and ArithDecay. The first two are essentially exponential decays that GeomDecay decays by: $T_o \times r^t$, and ExpDecay decays by: $e^{-r \times t}$. ArithDecay is more linear: $T_o - r \times t$, where r is a fixed ratio, and t is time (or iteration) [6]. From Figure 4, we see ExpDecay and GeomDecay are very close to each other because they are both in exponential form. ArithDecay has a shallower slope but it does reach higher fitness at the end.

This is due to slower decay of ArithDecay (linear) compared to exponential types. Note that the r ratio are default values, which are not tuned yet. This plot shows that a slower decay is needed for SA to jump to a better state to climb.

SA - ExpDecay Decay Speed

Only ExpDecay is discussed in details as they don't differ too much from each other and the important thing is the decay speed. Left of Figure 5 shows four different decay speeds. 400 runs are averaged to get smoother curves to analyze them properly. The general trend is that in early iterations, the faster decay means SA will lock onto a hill early on and improvement in fitness is faster, but they may lead to local maxima at the end so the curves converge to a lower mean, as shown in the figure. If the decay is slower, SA will have higher probability to jump to lower fitness state at the beginning, and hence the shallower slope of fitness (red). More random jumps also lead to higher average fitness at the end. Notice that none of the curves reliably reach global maxima of 94. This is due to the same reason as RHC that four peaks problem has multiple local maxima, and SA gets stuck in local maxima due to how it defines a neighbour state.

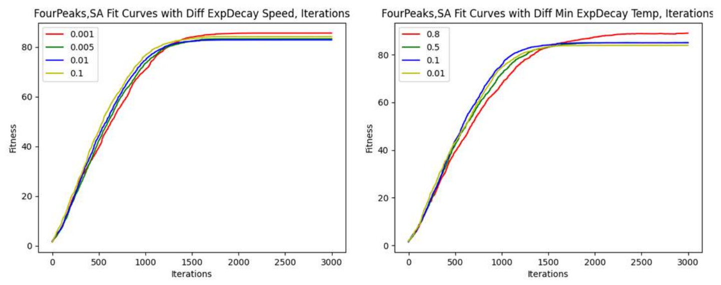

Figure 5 – Left: FourPeaks, SA fit curves w/ various ExpDecay speeds vs iterations; Right: w/ various ExpDecay min temperature

SA - ExpDecay Min Temp

The minimum temperature determines the minimum probability that a decay schedule can go down to. In right of Figure 5, we see a similar story that higher probability as minimum temperature is preferred because it is hard for SA to find global maxima in this problem.

MIMIC - Population Size

Similar to GA, MIMIC requires a population size to conduct probability distribution estimation. Higher population size means better estimation, which should result in higher fitness. In left of Figure 6, we can see the fitness score increases as population size increases. However, none of them was able to reach global maxima of 94 with the given 20 iterations. It is possible that more iterations is needed for the curves to reach 94. There will be a final comparison of all algorithms on four peaks using reasonably tuned hyperparameters for analysis. As for wall clock time, it is similar to GA that computation time is proportional to population size, shown in right of Figure 6. The fitness benefit of higher population is definitely diminishing from 300 to 600 while the computation time proportionally increases.

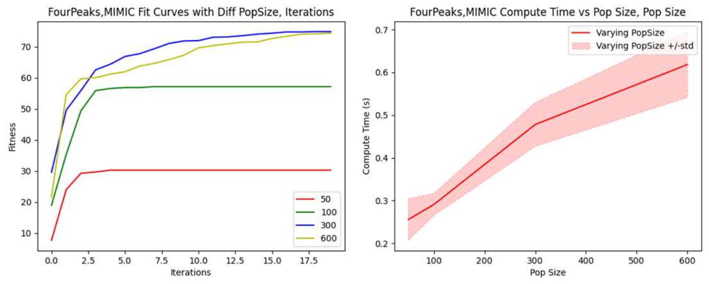

Figure 6 - Left: FourPeaks, MIMIC fit curves w/ various population size; Right: compute time vs population size

MIMIC Keep Percentage

This hyperparameter determines both quality and quantity of the samples used to update bits dependency. If percentage is small, higher fitness samples are kept, but quantity of them are low. On the other hand, if percentage is high, there will be a lot of samples which reduces sampling error, but lower fitness samples are in the mix which does not help with the dependency estimation. From Figure 7, we can observe the same behaviour of trade-off between quality and quantity. None of the runs was able to reach global maximum of 94. As mentioned earlier, there is a final comparison done at the end.

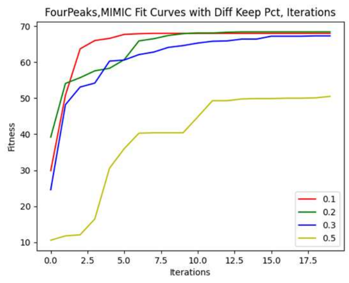

Figure 7 - FourPeaks, MIMIC fit curve w/ various keep pct vs iterations

Comparison of All Algorithms

All four algorithms are compared here against increasing state vector length (problem size) with reasonably tuned hyperparameters. Maximum iterations and attempts are all constant for all algorithms. In general, as shown in left of Figure 8, all algorithms suffer when the state vector length is bigger, given a fixed iteration. This is true for all three chosen optimization problems here that more iterations are required for bigger problem size, but the struggles in four peaks problem are a lot more pronounced. 1000 iterations were applied here while in other problems only a several hundred iterations were applied and there was no plateau or decrease in fitness in other optimization problems, which will be discussed later.

When the state vector length is small, GA is clearly the winner. It consistently reached global maxima and the wall clock time, shown in right of Figure 8, is almost constant. The GA algorithm itself has a constant compute time if the population size is fixed. The evaluation of fitness is where the compute time increases with problem size.

MIMIC comes in close second in terms of fitness, but the significant downside is the wall clock time. At smaller state vector lengths below 20, the difference in compute is not bad, but also not great compared to others. From 20 to 60, MIMIC's compute time is magnitudes more than other algorithms. MIMIC could be improved in terms of the fitness at higher problem size by having more complicated tree with dependencies more closely matching the problem. In other words, domain knowledge is needed to represent the structure better, similar to kernels in SVM.

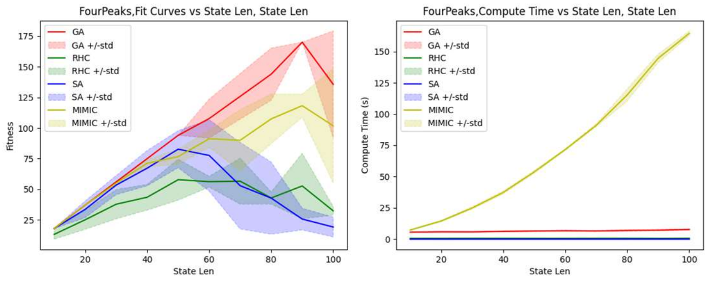

Figure 8 - Left: FourPeaks, fit curves among algorithms vs state length; Right: Compute time vs state length

For RHC and SA, they are similar in terms of how they find neighbours. SA in addition searches in wider state space by willing to jump to lower fitness, which makes it better than RHC in this case. The wall clock times for these two are very quick since there is no population needed unlike GA.

K-COLOR

The structure/dependency in k-color should make GA and MIMIC to excel in this problem. There is no very pronounced local maximum like four peaks, so other algorithms should not suffer as much. In the mlrose code implementation, it evaluates based on edges of the same color [7], so algorithms will instead look for global minima. The fitness plots in k-color will be lower fitness the better, and the y-axis labels will be "unfitness" instead of "fitness".

Edges

The edges define if any two bits are connected in a state vector, which essentially turns into a graph. A complete graph will have some edges with the same color nodes so the unfitness cannot be zero. A sparse graph will have higher chance of zero unfitness. Figure 9 looks at how algorithms respond to different percentage of connected nodes, where number of nodes is 20, and complete edges of 190 (= (20 x 19)/2). 20%, 40%, 60%, 80%, 100% edges were plotted but it's only necessary to show 80% and 100% in the plot.

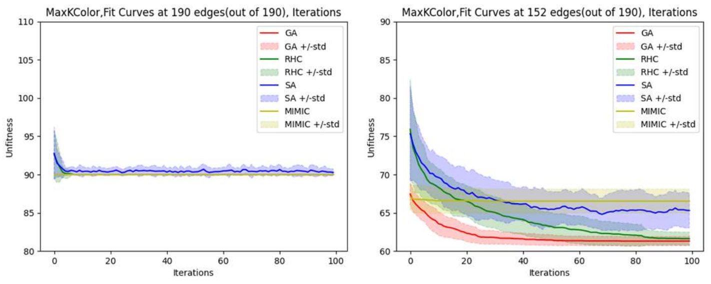

Figure 9 - Left: MaxColor, 100% edges unfit curve vs iterations; Right: lower % (80%) edges unfit curves vs iterations

When edges are 80% or lower complete, it is easier to distinguish which algorithm is better or worse in terms of fitness. When edges 100% complete, it is hard to see which algorithm is better at fitness since they score pretty much the same. The reason is more edges lead to over constraints and it leads to fewer choices to maximize the fitness (or minimize unfitness).

For the purpose of the analysis later in k-color, 20% edges is used to better distinguish which algorithm is better.

GA - Mutation Prob

From the same intuition from before, higher mutation probability should provide higher chance to reaching global maxima, or in this case, global minima. Figure 10 shows this is indeed the case. Lower chance of mutation leads to less diverse children to explore the solution space.

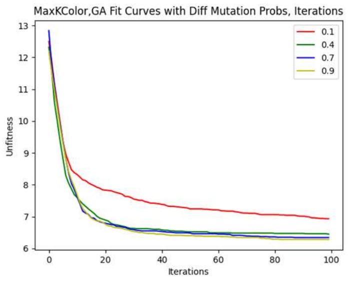

Figure 10 – MaxColor, GA unfit curves w/ various mutation probability vs iterations

GA - Population Size

Same intuition that higher population will lead to higher chance to explore the solution space and hence higher chance to finding global minima. Left of Figure 11 shows exactly this behaviour where higher population size finds lower unfitness faster w.r.t. iterations. Downside is shown on the right, as population increases, computation time also increases.

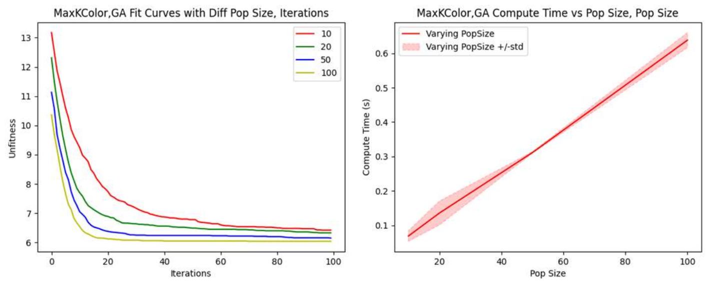

Figure 11 – Left: MaxColor, GA unfit curves w/ various population size vs iterations; Right: Compute time vs population size

RHC – Restarts

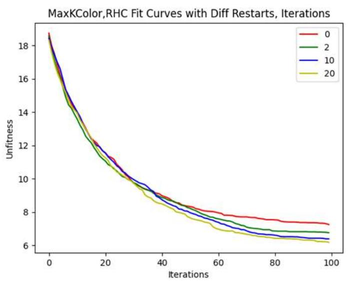

Figure 12 - MaxColor, RHC unfit curves w/ various restarts vs iterations

Same intuition as before, more restarts generates higher probability to "reset" and find a better hill to climb, that is, a hill that leads to global maxima. In this case, it is like gradient descent where we want to find global minima. Figure 12 describes the behaviour where 0 restarts lead to highest unfitness while 20 restarts leads to the lowest unfitness.

SA - Decay Schedule Types

Different decay types tell a different story here. Left of Figure 13 shows exponential types lead to lower unfitness compared to the linear type. Upon further analysis, ArithDecay at the end of 200 iterations with the default parameters, the T is higher than the other two types. Higher probability to jump to a neighbour state even if it has a higher unfitness will affect the curve shape as we can see in the plot.

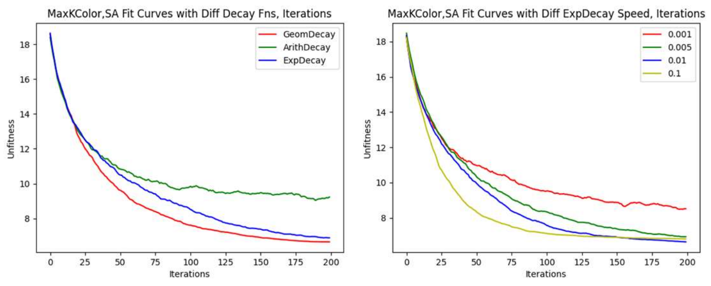

Figure 13 – Left: MaxColor, SA unfit curves w/ various Decay functions vs iterations; Right: w/ various ExpDecay speeds vs iterations

SA - ExpDecay Decay Speed

The slower decay the longer it takes to converge, and we see this same behaviour in right of Figure 13. At the highest decay speed of 0.1, the descent is the steepest at the beginning, but since it reaches final minimum temperature faster, it has less ability to jump out of local minima, and hence at the end of iteration 200, the average unfitness is higher than decay speed of 0.01.

SA - ExpDecay Min Temp

Unlike four peaks, lower minimum temperature is preferred, as displayed in Figure 14. This is because the difficulty of k-color is considered easier compared to four peaks, meaning less pronounced local minima which are easier to get out of.

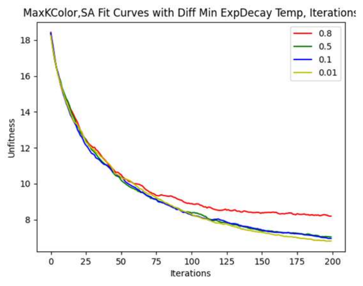

Figure 14 - MaxColor, SA unfit curves w/ various ExpDecay min temperature vs iterations

MIMIC - Population Size

As mentioned before, more population leads to better chance to finding the global minima, and this is the same behaviour here in left of Figure 15. Also the same as before, the importance of the amount of sampling is quite critical to MIMIC to have a good probability distribution estimation. In k-color problem, the unfitness difference between population 50 and 300 is slightly less than four peaks, which is due to k-colors is less difficult than four peaks. The compute time relationship is the same as before, proportional to the population size, shown in the right.

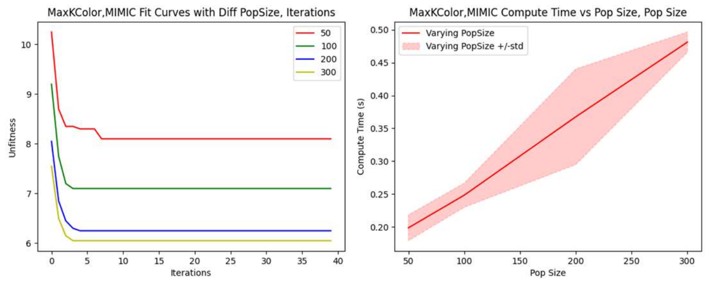

Figure 15 - Left: MaxColor, MIMIC unfit curves w/ various population size vs iterations; Right: Compute time vs population size

MIMIC Keep Percentage

Same story as before, there is likely a sweet spot or small region where the unfitness is the lowest. In Figure 16, we can see that the optimal value range is likely between 0.1 and 0.5, close to the range in four peaks problem.

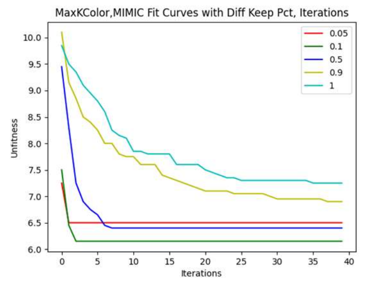

Figure 16 - MaxColor, MIMIC unfit curves w/ various keep pct vs iterations

Comparison of All Algorithms

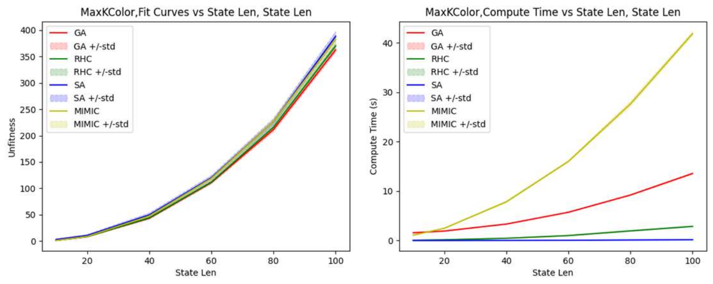

Figure 17 - Left: MaxColor, unfit curves among algorithms vs state length; Right: compute time vs state length

We can infer from left of Figure 17 that k-color problem is easier than four peaks. All four algorithms score close to global minimum. GA still wins with the lowest unfitness. MIMIC can likely score lower if more population is used. However, the wall clock time on the right tells us that MIMIC by far is the slowest algorithm, by a big margin. MIMIC does not reach the same unfitness as GA likely due to the assumed "dependency tree" structure. It is possible for k-color to have multiple parents which is not modeled in the MIMIC version used for the analysis. In right of Figure 17, all wall clock time curves seem to have exponential behaviour which is due to number of edges increases exponentially with number of nodes. Therefore it takes exponentially more time to evaluate the fitness. If global minima is required, GA is likely the winner, and if "close enough" solution is fine, SA can solve this in a fraction amount of time.

FLIPFLOP

The structure of flip flop can be treated as a single chain, which is a simple structure ideal for GA and MIMIC. Neighbour states also results in similar fitness so RHC and SA can benefit from it. Long plateaus can be a problem for all algorithms here to go from almost global maximum to the global maximum. The problem size is set to 20 (global maximum is 19).

GA - Mutation Prob

As mentioned earlier, flip flop plateau can be long before the algorithm can find the global maximum. In Figure 18, when mutation probability is low, it can only occasionally find global maxima, given that the red curve is the average and it's slightly above 18. For higher mutation probabilities, they are edging higher towards 19.

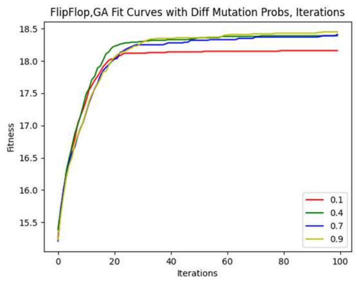

Figure 18 - FlipFlop, GA fit curves w/ various mutation probability vs iterations

GA - Population Size

More population always helps with fitness, and it is always a trade-off with compute time, as shown in Figure 19. Interesting behaviour here is when population is increased a lot, GA still cannot reliably find the global maxima. Flip flop pattern is extremely intuitive for humans but since GA does not understand the underlying structure of the problem, it is simply searching the wide solution space with lots of samples.

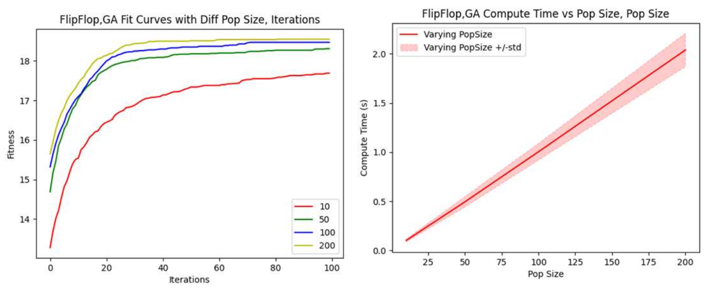

Figure 19 - Left: FlipFlop, GA fit curves w/ various population size vs iterations; Right: Compute time vs population size

RHC – Restarts

In Figure 20, restarts helps only a little in terms of fitness. This is due to the nature of RHC plus flip flop problem. Hill climb relies on finding a better random neighbour. When the fitness is 1 or 2 scores away from global maximum in flip flop problem, changing any bit will result in the same score (big plateau) and RHC will likely get stuck.

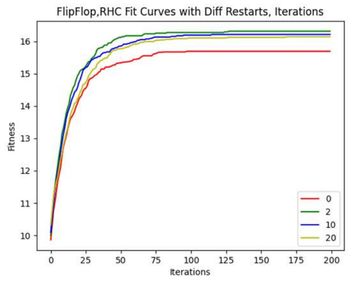

Figure 20 - FlipFlop, RHC fit curves w/ various restarts vs iterations

SA - Decay Schedule Types

A unique behaviour we see in Figure 21 is that ArithDecay suffers badly with the default setting compared to the other two algorithms. From the default settings, ArithDecay temperature at iteration 200 is still very high (0.98) while the other two temperatures are lower (< 0.8). From intuition, when the temperature is still high, it is easy for SA to jump to lower fitness state, and the fitness will not improve unless the temperature cools down more, like the other two decay functions.

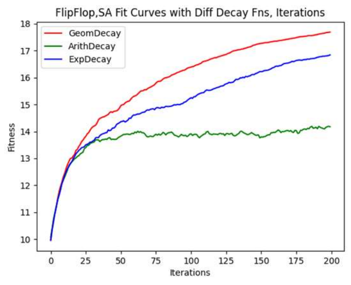

Figure 21 - FlipFlop, SA fit curves w/ various Decay fns vs iterations

SA - ExpDecay Decay Speed

The decay speed here perfectly describes the same behaviour. In left of Figure 22, when the decay speed is too low, like the red curve, the average fitness is stuck at around 14, just like ArithDecay function. Once the speed is increased, they converge faster. However, they do not show if they go to global maximum of 19. A final comparison of all algorithms will look into this.

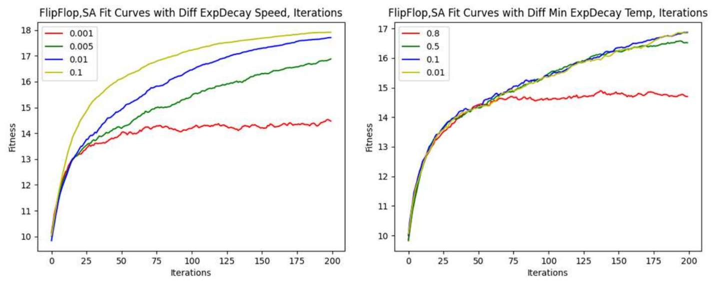

Figure 22 – Left: FlipFlop, SA fit curves w/ various ExpDecay speeds vs iterations; Right: w/ various ExpDecay min temperature vs iterations

SA - ExpDecay Min Temp

We can see the same condition in right of Figure 22. That is, higher temperature will lead to lower scores. This behaviour is the same in k-color, but the opposite in four peaks. This is likely due to the common characteristics of big plateaus in k-color and flip flop, while four peaks have multiple local maxima which require the algorithm to randomly jump out of to reach to a better state.

MIMIC - Population Size

For MIMIC, flip flop is likely the most suitable problem because each bit can be treated as having one parent except for the first bit (or the last bit), essentially a dependency tree. We can see in the left of Figure 23, population helps with fitness, which is expected. At 300 population, the fitness is almost at 19, which means most of the time the algorithm can find the global maxima. The benefit of large population also shrinks from 200 to 300, and the compute time is always a downside from right of Figure 23.

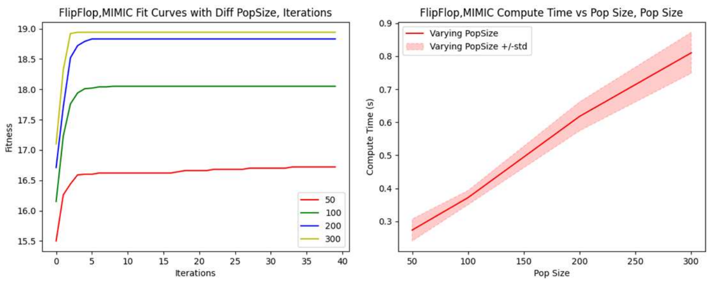

Figure 23 – Left: FlipFlop, MIMIC fit curves w/ various population size vs iterations; Right: Compute time vs population size

MIMIC Keep Percentage

Just like other observations made with keep percentage, Figure 24 here shows at both low and high percentage (0.1 and 0.8), the fitness does not perform very well compared to middle range like 0.3 to 0.6.

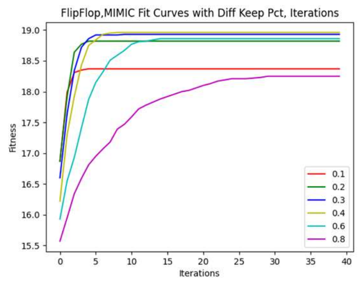

Figure 24 - FlipFlop, MIMIC fit curves w/ various keep pct vs iterations

Comparison of All Algorithms

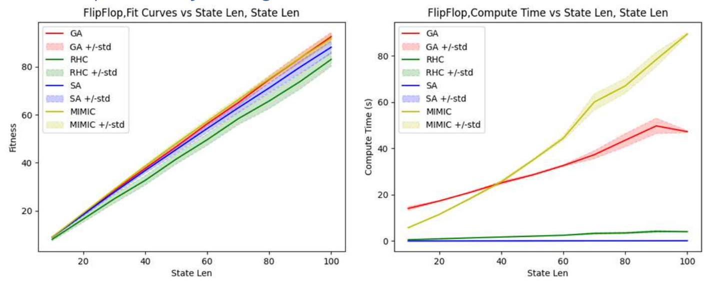

Figure 25 - Left: FlipFlop fit curves among algorithms vs state length; Right: Compute time vs state length

As expected, MIMIC is the first in terms of fitness shown in left of Figure 25. GA is just slightly below MIMIC for most of the problem size. SA and RHC are quite similar as discussed before and SA provides mechanism to jump to other lower fitness states but in the long run finds better solutions. As a result, SA scores slightly better than RHC. The wall clock time for MIMIC in right of Figure 25 is always expected to be slow as the problem size grows. At small problem size GA is worse since it has a higher overhead from mixing parent bits and random mutation. For SA and RHC they are always faster since there is no generation of population involved.

V. NN WITH RANDOMIZED OPTIMIZATION

The final section in this paper is to use RHC, SA, and GA to optimize the same neural network from Assignment 1 and use one of the datasets from Assignment 1. Due to different software packages (sklearn vs mlrose) being used, mlrose does not have some hyperparameters such as mini batch size and L2 regularization. Comparing results with the neural network from Assignment 1 is not really a fair comparison. The workaround is to use the gradient descent from mlrose, and go through the hyperparameter search to find the best hyperparameter(s).

All hyperparameter searches go through 5-fold methodology, and the model with the best average validation score is used. The hidden layers are 60 and 20, and the dataset is “credit prediction”. Optimization in this section is to minimize loss, instead of maximizing fitness. A simple accuracy is used, which is the same in Assignment 1.

NN WITH RHC

The only two tunable hyperparameters using RHC instead of gradient descent are the learning rate and restarts. From preliminary screening, we can see in left of Figure 26 that the validation score peaks when learning rate is around 0.1. For restarts, it echoes what we have learned in the paper, more restarts will help with loss minimization. Right of Figure 26 shows both training and validation scores increase with restarts. Downside is of course the increased in wall clock time.

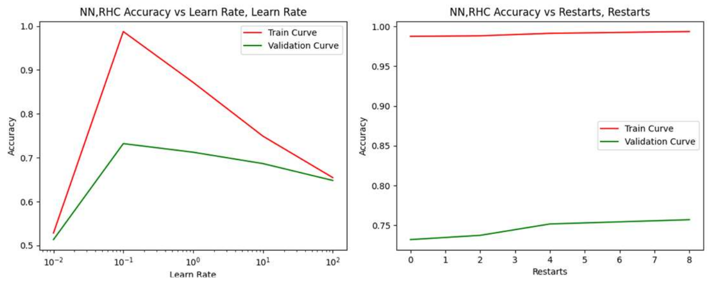

Figure 26 - Left: NN using RHC, accuracy vs learning rate; Right: accuracy vs restarts

Table 1 - RHC final hyperparameter tuning and results

Learning rate	0.15
Restarts	10
Wall clock time	548.10
Training accuracy	1.0
Validation accuracy	0.768
Test accuracy	0.779

With the preliminary results, a more refined hyperparameter search is conducted to find the best hyperparameters. Table 1 has the results. Figure 27 shows the learning curve for RHC. There is only training curve included since mlrose [1] does not have validation built into the training process. The unfitness (or error) goes to zero, which matches the training accuracy in Table 1. With the validation accuracy, we know there is overfitting, which occurred in Assignment 1 as well and L2 regularization was used to reduce some of the overfitting.

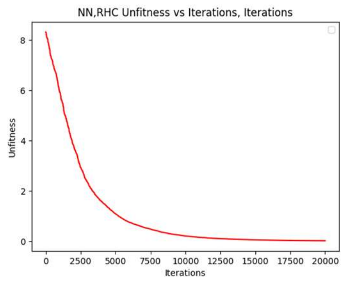

Figure 27 - NN using RHC with tuned hyperparams, learning curve

NN WITH SA

There are some tunable hyperparameters with SA, namely, learning rate, types of decay functions, and decay speeds. From the preliminary screening, learning rate of ~1.0 produces the highest validation accuracy, according to left of Figure 28.

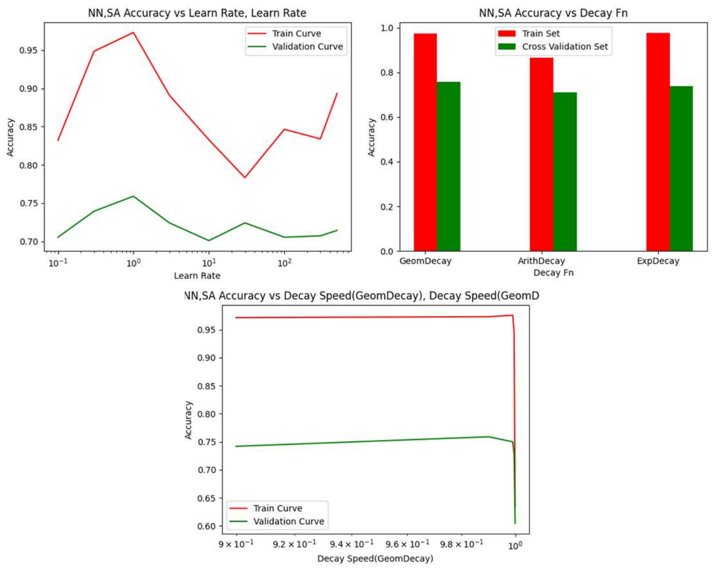

Figure 28 - Left: NN using SA, accuracy vs learning rate; Right: Accuracy vs decay functions; Bottom: Accuracy vs GeomDecay decay speed

For decay function types, ArithDecay scores the lowest among the three with the default settings, shown in right of Figure 28. This behaviour is similar to k-color and flip flop problems. This suggests the weight space (state vector space) prefers the random jumping probability to die down quicker because there are no pronounced and undesirable local minima. This supports the more recent studies on the presence of local minima in neural networks [8, 9]. In addition, jumping to a random state in continuous state space is a very different story from discrete state space. In bit strings, it is as simple as flipping a bit. In continuous space, changing a single weight using a specified learning rate can be problematic. If the learning rate is tuned for the most sensitive weight among the weight space, then other less sensitive weights might not show enough difference with the random state jump. This can induce unnecessary plateaus for the algorithm. GeomDeday is chosen to analyze the decay speed. In bottom of Figure 28, we see that indeed when the decay speed is very low (0.9999), both training and test accuracies do not do well, as also observed previously. It is hard to see what x data points were used. They are [0.9, 0.99, 0.999, 0.9995, 0.9999].

Table 2 shows refined final hyperparameter search results.

Table 2 - SA final hyperparameter tuning and results

Learning rate	0.5
GeomDecay speed	0.99
Wall clock time	73.96
Training accuracy	0.965
Validation accuracy	0.779
Test accuracy	0.761

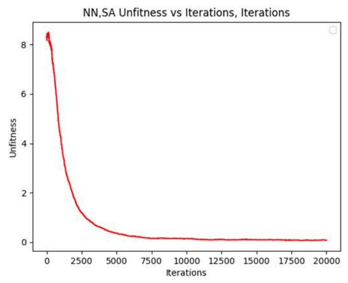

Figure 29 - NN using SA with tuned hyperparams, learning curve

Figure 29 shows the learning curve for SA. The unfitness (or error) goes to very close to zero, which matches the training accuracy in Table 2. With the validation accuracy, we know there is overfitting, which occurred in Assignment 1 as well and L2 regularization was used to reduce some of the overfitting.

NN WITH GA

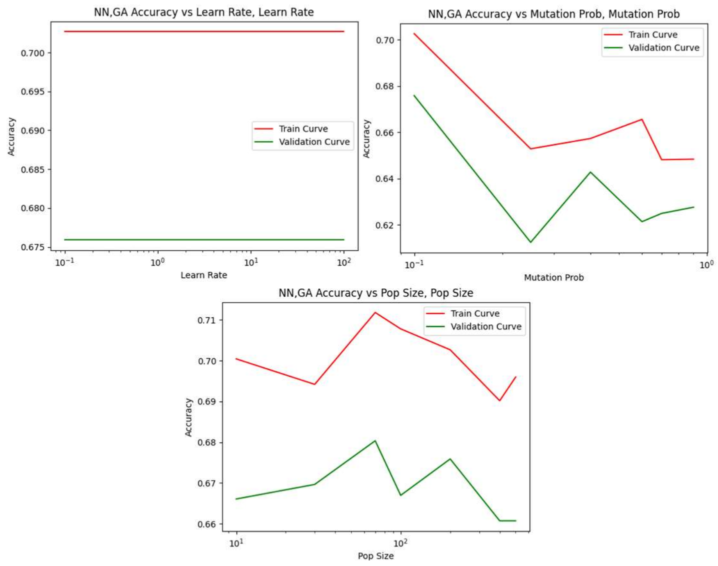

Figure 30 – Left: NN using GA, accuracy vs learning rate; Right: Accuracy vs mutation probability; Bottom: Accuracy vs population size

GA is quite different from the previous two algorithms such that it relies on a certain population generated with some mutated weights. This can be a problem for continuous state space because it is no longer just flipping bits between zero and one. Randomly changing the multiple weights by small amounts can prevent the algorithm from finding the global minima (or decent local minima).

We can use this reasoning and it explains that in right if Figure 30, low mutation probability is preferred.

For left of Figure 30, learning rate has no effect on the accuracy. It is possible the variable is not used in the version of mlrose [1] and a hardcoded value is used for changes in weight values. This means the tunable hyperparameters are limited to two, with the last one called population size. When population increases, it should increase the chance for algorithm to find a better solution. In bottom of Figure 30, we can see the accuracy is low when population is low. The accuracy peaks when population is in double digits. The noise in the zig zag curves come from low number of runs executed and the nature of GA which uses mutations to find global minima which does not result in great scores.

Table 3 shows refined final hyperparameter search results.

Table 3 - GA final hyperparameter tuning and results

Mutation probability	0.04
Population size	200
Wall clock time	129.67
Training accuracy	0.767
Validation accuracy	0.733
Test accuracy	0.700

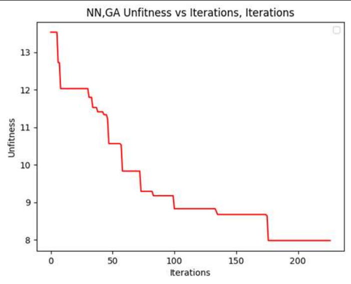

Figure 31 - NN using GA with tuned hyperparams, learning curve

Figure 31 shows the learning curve for GA. A few differences here compared to other learning curves. The curve is not smooth, due to the nature of GA which does not look for neighbours to improve the fitness. The curve is short even when maximum iteration is 20k, same as others. This means GA struggled to find better states early on.

NN WITH GD (Gradient Descent)

There is only one hyperparameter to tune, learning rate. Multiple runs are executed to find the best learning rate. In Assignment 1, the back propagation was done with Adam optimizer. Since there is no such optimizer in mlrose [1], GD is used.

Table 4 – GD final hyperparameter tuning and results

Learning rate	0.0003
Wall clock time	173.80
Training accuracy	0.989
Validation accuracy	0.780
Test accuracy	0.782

Figure 32 shows the learning curve for GD. The way GD was written in mlrose [1] requires maximization. The training went through the whole 20k iterations but the solution converges early on so only 50 iterations are plotted.

Table 4 shows the tuned hyperparameters using GD.

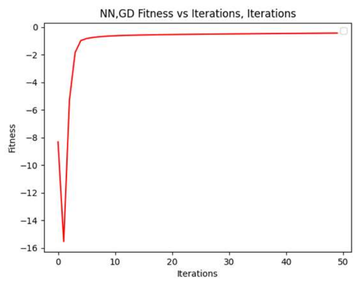

Figure 32 - NN using GD with tuned hyperparams, learning curve

OVERALL COMPARISON

From Table 5 we can see the algorithms which find neighbours do better than GA because GA modifies multiple weights but not necessary updating them in the direction towards minimizing loss, like GD. GD is still the best among four algorithms. RHC is very close to SA in validation/test accuracies but RHC requires a lot more time to train. GD in conclusion is quite a robust optimization algorithm for NN type problems. Last column shows the results from Assignment 1. Validation score is the best due to more tuneable hyperparameters, and the slightly lower test scores is due to low number of samples in the dataset.

Table 5 – Comparison of all algorithms on dataset "credit prediction"

	RHC	SA	GA	GD	Adam
Wall clock time	548.10	73.96	129.67	173.80	N/A
Training accuracy	1.0	0.965	0.767	0.989	0.957
Validation accuracy	0.768	0.779	0.733	0.780	0.829
Test accuracy	0.779	0.761	0.700	0.782	0.779

VI. REFERENCES

1. Mlrose-hiive general usage, URL: https://pypi.org/project/mlrose-hiive/
2. Scikit learn general usage, URL: https://scikit-learn.org/stable/
3. Mlrose fitness functions usage, URL: https://mlrose.readthedocs.io/en/stable/source/fitness.html
4. Mlrose algorithms usage, URL: https://mlrose.readthedocs.io/en/stable/source/algorithms.html
5. Mlrose discrete optimization source code, URL: https://mlrose.readthedocs.io/en/stable/_modules/mlrose/opt_probs.html#DiscreteOpt
6. Mlrose decay schedule usage, URL: https://mlrose.readthedocs.io/en/stable/source/decay.html
7. Mlrose max k-color problem source code, URL: https://mlrose.readthedocs.io/en/stable/_modules/mlrose/fitness.html#MaxKColor
8. P. Bhadani. "Is the Local Minima a real issue in deep neural learning?" medium.com. URL: https://medium.com/@pranabbhadani/is-the-local-minima-a-real-issue-in-deep-neural-learning-6d812b28d684 (accessed Feb. 28, 2023)
9. S. karki. "Saddle point are major bottleneck in learning curve in Neural Networks, not Local Minima." medium.com. URL: https://shivaraj-karki.medium.com/loss-function-in-cnn-doesnt-get-stuck-in-local-minima-saddle-point-is-the-culprit-316f456c23f3 (accessed Feb. 28, 2023)