Randomized Optimization

Jarrett Alexander

October 16, 2022

1 Introduction

The following experiments are used to discover the behavior and performance of different optimization algorithms. The experiments tune different parameters for 3 different algorithms over 3 different problems. These algorithms include Simulated Annealing, Genetic Algorithms, and MIMIC. Further, Random Hill Climbing is used for comparison with all of these algorithms as its behavior provides a nice comparison with a simple algorithm. The 3 problems that will be evaluated are: One Maximum, N Queens, and Traveling Salesman.

Then, a Neural Network is trained using Gradient Descent as well as Random Hill Climbing, Simulated Annealing, and Genetic Algorithms. The behavior of these algorithms becomes very clear when looking at the loss curves that are generated by this training. Their performance in both accuracy and training time will be completed on the Red Wine Quality Dataset provided by the UCI Repository[DG17].

For all following experiments, the mlrose-hiive library was used[Rol20].

2 Problems

2.1 One Maximum

One Maximum is a simple problem that, as the name describes, has a single maximum. For this problem, the fitness function is simply the number of 1's in a given bit-string. N for this problem is the number of bits in that bit-string. Finally, the single maximum is defined as the bit-string of all 1's. For example, if N=5, the only maximum for the entire space is [1,1,1,1,1].

By using the mlrose library[Rol20], we know that the neighborhood for this problem is defined as all values that shares all but 1 value with the current value. For example, [1,1,1,0,1] is a neighbor with [1,1,1,1,1] but not [1,0,0,0,1].

This is a very simple problem that pretty much any optimization problem should have problems solving. There are no local maxima which means that algorithms like RHC and SA should perform very well. Both "climb" upwards by looking at nearby neighbors. Because there is a guaranteed neighbor with a higher value, they should quickly and easily climb to the optimal point.

2.1.1 Simulated Annealing Tuning

For the following experiments, max attempts were set to 200 and decay was geometric unless stated otherwise.

Max Attempts Max Attempts defines the number of times that the algorithm will either check for another greater value or explore downward due to the temperature. If it does not explore downward or find a better neighbor within the number of max attempts defined, it will stop there. A low max attempts will not allow SA to search enough at a given point in order to find a better point. However, a high number of max attempts will for the algorithm to take much longer.

To experiment with this, a range of max attempts from 1 to 181 was evaluated over a size 20, 200, and 1000 One Maximum problem. Refer to Figure 1.

For the smallest problem, all but the single max attempt had reached the optimal value (20) by the 200th iteration of the algorithm. Some were faster than others but this behavior was mostly stochastic due to the different starting places for the different algorithms. This was run without a random seed because it was very hard to interpret the chart as they all followed the same path.

For the 200 size problem, lower max attempt values did not make it to the optimal value and only the 121 max attempts value made it to the optimal before 1000 iterations. However, this behavior is again fairly stochastic because if the size 181 problem started at the same value as the 121, it would

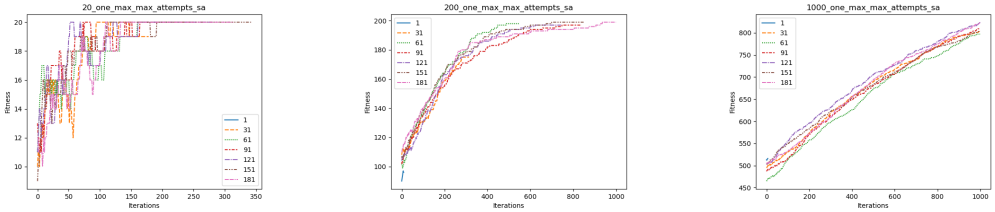

Figure 1: Max Attempts

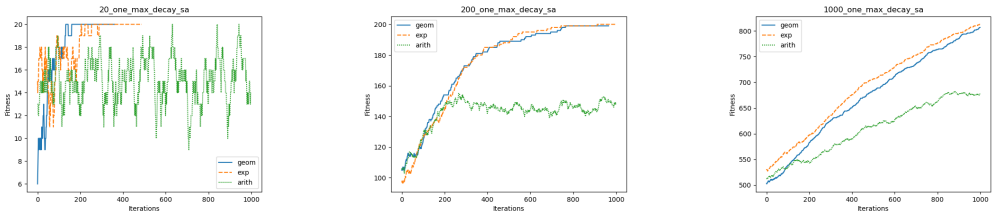

Figure 2: Decay Schedule

have also traversed to the maximum location (given the same random seed). The 200 size problem also reveals the trade off between runtime and optimization. The higher value max attempt runs were able to make it to near-optimal solutions but it took them about 1000 iterations to do so. However, with the lower value max attempts, the algorithm was able to get to a significant percentage of optimality within 500 iterations. Once again, there is a tradeoff that must be played with if this algorithm is going to be used.

The curves also seem to flatten out as they approach maximum fitness. This is because there are simply not as many neighbors with a higher fitness value as you increase your own fitness. For example, the global maximum only has neighbors with a lower value than it. This flattening seems to occur near the 180 fitness level. Each value at the 180 level has only 20 neighbors with a fitness higher than it.

Finally, the 1000 size problem showed that with a large space for exploration, the number of max attempts does not matter. There was a steady rate of improvement from start to finish. This was stopped at the 1000 iteration value to give it a fair comparison with the other experiments. Only the experiment with a single max attempt was not able to improve greatly over the starting value of about 500.

Decay The decay defines the schedule in which the temperature value is decreased. In the mlrose library[Rol20], there are 3 built in that will be experimented with at each of the sizes above. Refer to Figure 2. On all sizes, geometric and exponential decay seemed to work pretty much the same. The minimum temperature that is allowed with all of these decays is .001. This means by the 1000th iteration, the temperature value of geometric and exponential decay are essentially the same. .001 for geometric (due to the limit) and .006 for Exponential decay. In both cases, not much exploration is going to happen unless given many attempts to do so. If you are solving a problem that could have many local optima closely distributed around the global optimum, exponential decay would allow for more exploration in the later stages but only by a small margin.

Arithmetic decay performed the worst on this problem. At 1000 iterations, arithmetic decay is still .9. There is a 90% chance that it will still be exploring at this point. This is very clear to see in every chart. While it does tend to improve over time, the rate at which it does so is very slow with many large movements down. While the One Max problem is not well suited for this decay schedule, it could be a useful schedule for more complicated spaces with many spread out local maxima. This decay schedule will force exploration for much longer. This also means that it would take quite a bit longer to reach an optimal value. As we can see in this experiment, it was unable to reach any optima within 1000 iterations and therefore will take quite a bit longer to run.

2.1.2 Algorithm Performance

RHC, SA, GA, and MIMIC were all evaluated on One Max with sizes 20, 200, and 1000. For each, a search over the possible hyperparameters was conducted to find the best ones. For GA and MIMIC, a similar search will be explained in further detail in the N Queens and Traveling Salesman sections

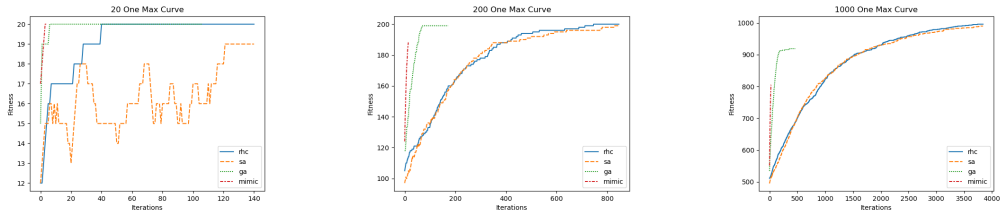

Figure 3: One Max Algorithm Comparison

which follow this one. In the interest of space, these are omitted here.

For RHC, a max attempt value of 100 and a max number of restarts of 10 was chosen. The number of restarts gives the maximum number of times that the algorithm can start over from a different random starting point to find the max.

For SA, the above experiments give a max attempt of 150 and a geometric decay schedule. For GA, experiments were ran and the best performance was achieved with a population size of 100, breed percentage of .5, and a max attempts of 100. For MIMIC, a population size of 100 and a keep percentage of .25 were used. Refer to figure 3.

Size 20				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	20	149	141	0.017
SA	20	392	348	0.006
GA	20	10910	107	0.264
MIMIC	20	609	5	0.379
Size 200				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	200	2315	847	0.09
SA	200	1222	1094	0.016
GA	199	17507	172	0.44
MIMIC	188	1731	16	118.95
Size 1000				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	996	7789	3389	.37
SA	991	4587	4076	.07
GA	919	46863	462	1.33
MIMIC	805	3057	29	5168.64

One Maximum is a great problem for Simulated Annealing and Random Hill Climbing to solve because it is able to move up from every single point except the maximum. It also scales incredibly well with scale because it scales by a factor of N. When solving for N, if the fitness is N/2, the algorithm has a 50% chance of finding a better neighbor. This gets harder as fitness approaches N. While it does get harder, the algorithm is still able to gain fitness very quickly during the beginning of optimization. While algorithms like MIMIC and GA are able to find the optimal value in very few iterations and function evaluations for small N, they must explore the same amount due to population size regardless of the current fitness.

As N gets larger, SA and RHC do not have a hard time keeping scaling with the problem. For N=200 and N=1000, the rate of change in fitness/N appears to be close to the same. For small values of N, we also see a relatively smooth climb to maximum fitness for RHC. The climb for SA is not as smooth because the decay on the temperature has not had enough iterations to actually decrease by much. SA is exploring a lot early on so within 150 iterations there is quite a bit of movement.

For N=1000, GA and MIMIC are not able to converge to a maximum in any amount of time close to SA or RHC. SA and RHC are both able to get very close to the maximum in under 4000 iterations. However, this is an unfair comparison because the runtime of the algorithms over 4000 iterations is much faster. RHC and SA run in .37 and .07 seconds respectively. This is faster than GA and much faster than MIMIC at 1.33 and 5168.64 respectively.

Finally, when it comes to function iterations, SA is comparable to MIMIC. However, it seems that MIMIC spends much of those function evaluations creating a population at each step. This wastes quite a bit of compute for a problem that has a single maximum. SA almost always generates a better or equivalent next step and therefore is able to only use on average 1.11 evaluations per iteration. GA suffers from the same problem as MIMIC where much of the function evaluations are used on the population which doesn't gain very much information for this problem.

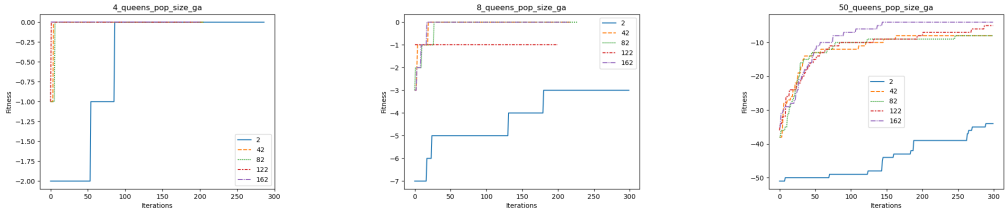

Figure 4: Population Size

2.2 N Queens

The N Queens problem is optimal in the case when all N queens placed on a chessboard are placed in such a way that none of them can kill each other. For most N, there are many global optima to this problem. The fitness function is the negative of the number of queens pairs that can attack each other. Also, there are many local optima to this problem as well[NQ]. It is possible to get all quarters of a chessboard (or really any internal set of squares) to adhere to the "no attack" rule but this can force others to be able to attack.

The problem also cannot be easily shifted to generate other global maxima. For example, a solution to the 8 Queens problem cannot be used to create another optima by simply moving all pieces one to the right (and shifting the one in the right column all the way to the left). This will almost certainly allow for there to be an attack. Once again, the locations of the queens are very sensitive to movement which forces any algorithm to explore or build the correct answer from the underlying structure of the problem.

Finally, for any N, this problem is composed of smaller versions of the same problem. For example, 8 queens must be composed of 4 4x4 boards which must satisfy 2 queens each as well. Therefore, this problems solution must almost built from the ground up. The structure of the problem must be observed and used at each iteration for the performance to improve. To avoid too much shuffling, the structure should be part of the solution from the start or explored at each iteration.

2.2.1 Genetic Algorithm Tuning

For the following experiments, population size was set to 50, breed percentage .75, mutation probability to .1, and max attempts to 200 unless specified otherwise.

Population Size The population size for GA is the number of samples generated at each iteration. A larger population size will increase the space that is explored but will also increase the number of fitness evaluations at each iteration. For the following experiments, 3 different size problems were evaluated with N=4, N=8, and N=50. Populations sizes varying from 2 to 162 are tried for each problem. Refer to Figure 4. For all problem sizes, adjusting the population size had a similar effect. Very small populations performed consistently poorly for all sizes. N=4 was the only time an optimal solution was found with a population size of only 2. However, this was achieved after the other population sizes had gotten there.

To see the effect of population size on fitness, a much larger problem needed evaluated. N=50 shows the minor differences in larger population sizes. All population sizes seem to increase it fitness fairly evenly from the start. This makes sense because at this point, there is not much fidelity needed in exploration. Major gains are made and that is all. After this point, the higher populations tend to keep increasing in fitness at a higher rate. The population size of 162 continues above the others as it is able to explore more of the space at the higher fitness levels. At each iteration, it is pairing more high fitness pairs together and some of these are able to generate better pairs. It is simply more likely for this to happen with a higher population size and therefore it continues to increase it fitness. This occurs as well in the population size of 122. While not as performant as 162, it continues improving.

While the population size of 162 does converge in less iterations, it also converges in less function evaluations. The population size of 122 converges to a similar fitness by the 300th iteration as the 162 at the 150th iteration. But the population size of 162 does not have 2x the fitness evaluations because it only has 40 more parts of the population. This means that with a higher population, the algorithm is able to uncover more about the structure of the problem with each iteration. Each function evaluation is able to be paired with another in order to actually increase the value of each evaluation.

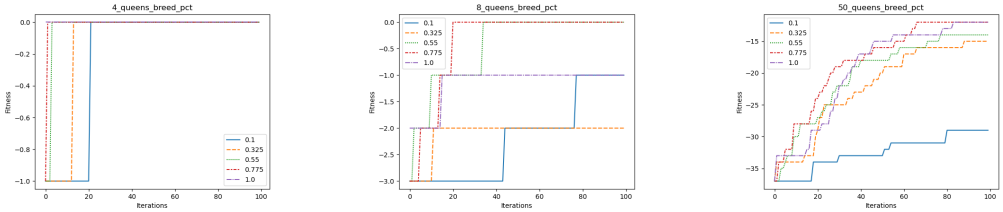

Figure 5: Breeding Percentage

Population Breeding Percentage Breeding percentage defines how much of the population is allowed to breed. With a lower percentage, only the higher fitness are allowed to breed at each point. To test this, 5 different percentages were tested on the same problem sizes as above. Refer to Figure 5. Once again, we see the worst performance from very low values of breeding percentage. In all 3 problem sizes, a breeding percentage of only .1 was the slowest to improve. There is not enough diversity when the population size is that small. Even though we are picking the very best of the best samples at each iteration, we are not able to create any better solutions because there are so few possible outputs based on those few inputs.

As we increase the breeding percentage, the performance tends to increase. For the standard N=8, the best fitness is achieved with a breeding population of .55 and .775. By allowing a large percentage of the top of the population to breed, you allow a lot of exploration. The N queens problem requires a lot of exploration as each queen can be in 1 of N positions. By limiting this the algorithm is unable to explore enough and will get stuck in local maxima.

Interestingly, a breeding percentage of 100% with 50 queens is able to compete with a breeding percentage of .775. With 50 queens, it is likely that a lot of exploration is required to achieve optimality. Therefore, by just allowing all members of the population to breed, the algorithm is more likely to find a better solution.

Mutation Probability This is the probability that at each child generated, there is a mutation. The full analysis is omitted here due to space. However, it is important to note some of the characteristics found during experimentation. This probability behaves very similar to breeding percentage. If mutations are low, exploration does not occur and local optima are found. If mutation rates are high, there is not much actual improvement because the changes are all random. A trade off is required and through experimentation, a value between .1 and .5 are found best.

2.2.2 Algorithm Performance

Size 4				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	0	1176	128	0.08
SA	0	268	235	0.02
GA	0	20502	101	1.90
MIMIC	0	102102	101	6.94
Size 8				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	0	1066	234	0.12
SA	0	425	366	0.05
GA	0	23118	114	3.28
MIMIC	0	107109	106	13.4
Size 50				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	-7	9608	696	29.5
SA	-4	2610	2369	1.61
GA	-1	55901	277	35.5
MIMIC	-17	113122	112	133.2

Similar to the experiments ran on GA here and SA in the One Max section, experiments were ran on RHC, SA, and MIMIC for this problem to determine optimal hyperparameters. RHC was given 100 attempts and 100 restarts. SA was given 100 attempts with a Geometric decay. GA was given a population size of 200, breeding percentage of .75, and a mutation probability of .5. MIMIC was given a population size of 1000 and a keep percentage of .2. With a smaller pop-

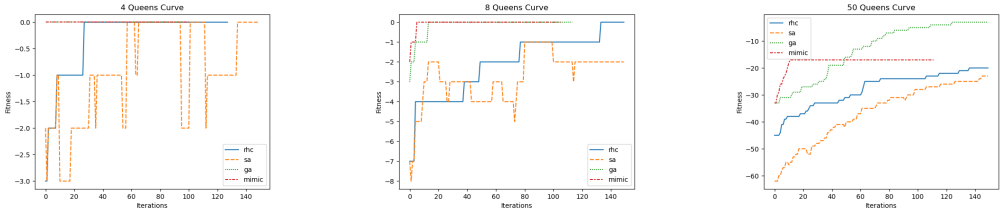

Figure 6: N Queens Algorithm Comparison

ulation size MIMIC was consistently getting stuck in local maxima on this problem. While it is not a direct comparison to GA with population size, it is important to note this difference. Refer to figure 6. As can be seen in the figure, GA was able to achieve the best performance with N=50. It also did so with the least number of iterations. This is slightly misleading as it also used the second most function evaluations. MIMIC is able to gain fitness more quickly from the start but it get stuck in a local optima and does not improve past that point. The probability distribution it had generated likely did not have any samples that would improve performance. It is able to gain fitness quickly as it is building a structure underneath its exploration but without doing a lot of exploration of the entire problem space, it is easy for it to get stuck in a local optima. For a successful run of MIMIC on this problem, a much larger population size and higher keep percentage would be needed to allow for more exploration.

SA and RHC both perform relatively similarly. RHC is much slower because it requires random restarts to be effectived. SA is quicker because it avoids this. SA is able to find a similar solution to RHC but SA takes 1.6 seconds and RHC takes 29.55 seconds. Without those random restarts, it is unlikely that RHC is able to perform similarly.

GA is able to find the best solution of all algorithms for N=50. It leaves a single pair of queens able to attack each other. It does so in the least number of iterations as well. It has second most function evaluations however. It also is close to runtime as RHC at 35.5 and 29.55 seconds respectively. MIMIC ws the slowest at 133.2. I suspect that GA is able to explore the most at each iteration and therefore is able to converge in a low number of iterations. It is not limited by a probability distribution like MIMIC because it can add randomness through breeding percentage and mutation probability. This randomness forces it to look at many solution at each iteration and therefore find the best one quickly.

For the smaller problems, all algorithms were able to converge and did so in similar iterations as well as time. Due to the small problem size, exploration is not as key and algorithms such as SA and RHC are able to find solutions easily.

While N went from 4 to 50, the number of fitness evaluations for GA only increased by roughly 100%. While the time it took to solve the problem greatly increased (likely due to the increase complexity in mating), the actual computation for fitness evaluations did not need to increase that much to find a near optimal solution.

2.3 Traveling Salesman

Traveling Salesman is defined as the shortest possible route that visits a list of cities and returns to the start. It is NP hard and therefore a problem that is good for optimization problems to solve. A brute-force search is pretty much intractable for any sufficient value of N cities.

Random samples from the distribution are very very unlikely to be very good either. For N, there are (N-1)!/2 possible paths[TSP]. The search for a good solution must do lots of exploration if the algorithm builds no structure. However, for algorithms that rely on the structure of the underlying problem, it can generate new results based on previous results that worked well. Unlike N-Queens, there is typically a single optima for this problem.

Also, a sub-structure to this problem (a path inside of the optimal path) may or may not be optimal for the final solution. Optimizing the path for a set of n/4 cities may need changed completely depending on the correct sink and source for that subpath with respect to the larger optimal path. This can lead to many local optima during exploration. Finally, this problem scales with N!. For any algorithm, this is going to force a lot of exploration and compute time as N gets large.

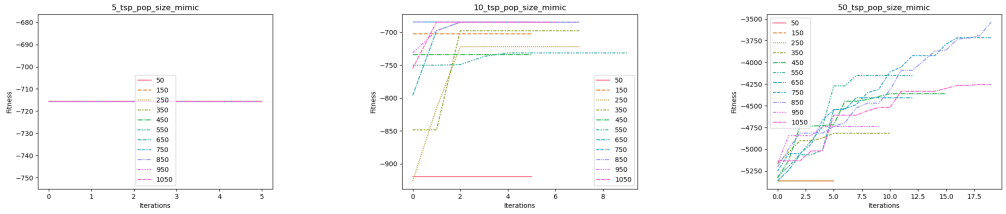

Figure 7: Population Size

2.3.1 MIMIC Tuning

For the following experiments, population size was set to 500, keep percentage .2, and max attempts to 5 unless specified otherwise.

Population Size Population size for MIMIC is very similar to the population size discussed in the GA section. At each iteration, MIMIC generates a new set of samples from the previous iterations probability distribution. This parameter just determines how many will be generated. Similar to GA, this shows to be a good way of inducing exploration in the algorithm. The more samples are added, the more accurately the underlying distribution is represented. For experimentation, a TSP with size 5, 10, and 50 were examined in order to get an idea about scalability as well as performance. All were examined at population sizes ranging from 50 to 1050. Refer to Figure 7. On the smallest TSP size, there was no difference in performance by any population size. This is because with so few paths between cities to explore there are no real gains to be made from a larger population. The first iteration of the algorithm is likely to have a near optimal solution in it because the space is so small. Therefore, a small population will perform to the same ability as the larger one and it will do so in a much shorter amount of time.

At the 10 and 50 size TSP, population size begins to have an effect. In both cases, the population size of 50 performed very poorly. With so few examples, it is unlikely that MIMIC is able to generate a probability distribution that can create new samples with higher value. TSP is full of local optima and with only 50 samples it is unlikely that the algorithm is able to generate a distribution that can escape these points.

While the performance of the population sizes is somewhat stochastic as the population is increased, it typically trends to perform better in less iterations as population size is increased. This is only broken by the size 50 TSP which seems to show strong performance on both 750 and 850 sized populations. This problem was evaluated with a different random seed and this was just a coincidence. It is likely that in the chart shown, the starting place of the 1050 and 950 population size forced them into a local optima that could not be escaped in this case. Changing the random seed allowed them a different starting place and they performed quite well.

The trade off for population size is run time. It will tend to increase performance per iteration but it also increases run time. In the size 50 experiment, the 2 best performers were 750 and 850 population size. They appeared to have very similar performance finding the optimal value in similar iterations. However, the population size of 750 took 37.4 and 850 took 53.2 seconds. A 13.3% increase in population caused a 42.2% increase in run time.

While increasing population should increase performance per iteration and prevent getting stuck in local optima, it should be avoided if your problem size does not require it because of the extra time it requires to run.

Keep Percentage Keep percentage defines the percentage of the population at each iteration that is used to generate the next probability distribution to be sampled from. A low keep percentage will cause the algorithm to only generate a distribution from the best of the best samples. A tradeoff must be made with population size. With a low keep percentage, a high population is almost necessary because MIMIC will simply not have enough samples to generate an accurate probability distribution otherwise. The same experiment sizes were used as above and a range from .01 to 1.0 were tested. Refer to figure 8. Once again, the smallest problem size showed no difference in performance because the starting population for all percentages contained the optimal value. With a TSP this small, there are very few actual combinations possible. With a high enough population size, it is very likely that it will contain the optimal value in the start.

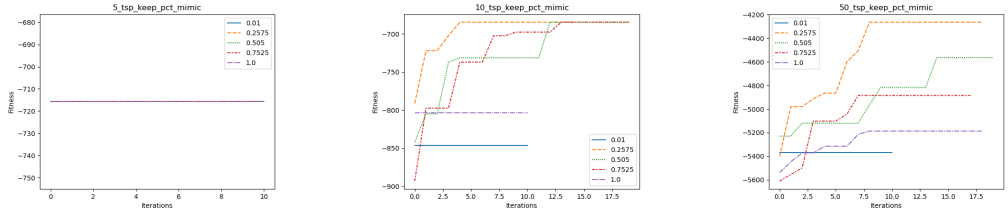

Figure 8: Keep Percentage

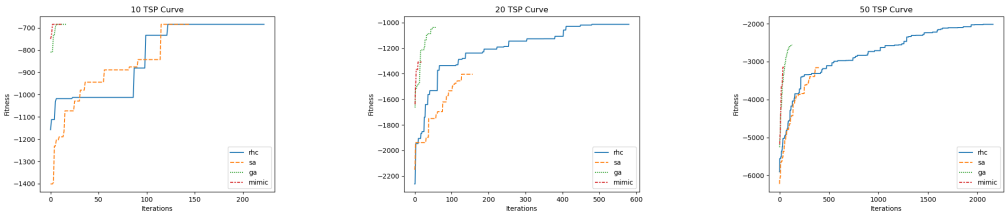

Figure 9: TSP Algorithm Comparison

On sizes 10 and 50, the keep percentages in the middle of the range .25 -¿ .75 performed the best. A keep percentage of .01 is so small that it likely does not contain more than just a few samples. These samples are not likely to create an accurate probability distribution and therefore get stuck in a local optima quickly. At 1.0, we are pretty much just randomly sampling the entire dataset over and over. This will force the algorithm to just reach an "average" case and never improve from there. Because we are not allowing it to go down, we see it stagnate at the size 10 and then have a slight increase in the size 50 problem.

25% seems to be the sweet spot for this problem and this algorithm combined. On both 10 and 50 size TSP, it is able to climb in the least number of iterations. At higher percentages, MIMIC is likely retaining too much information at each iteration. This allows it to have a better understanding of the problem space but does not allow it to climb quickly. 25% still allows for a healthy probability distribution but still only keeps the best which forces it to climb.

While it is able to climb the fastest at 25%, 50% showed the highest score after many more iterations. (The charts are cut off to show the most interesting part clearly). Because the 50% was able to explore some of the worse examples more often and had a more accurate distribution at each step, it did not get stuck in the same local optima as the 25% run. It did however take almost 2x as long to do so.

2.3.2 Algorithm Performance

Size 10				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	-684.5	2005	223	.38
SA	-684.5	159	146	.007
GA	-684.5	19023	18	.83
MIMIC	-684.5	14016	13	3.78
Size 20				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	-1012	12153	581	0.76
SA	-1403	175	158	0.008
GA	-1036	61075	60	3.69
MIMIC	-1307	21025	20	12.13
Size 50				
Algorithm	Fitness	Fitness Evaluations	Iterations	Total Runtime
RHC	-2016	51570	2159	2.76
SA	-3158	455	399	0.025
GA	-2558	130175	129	14.00
MIMIC	-3153	59077	58	108.71

Similar experiments were run on RHC, SA, and GA for this problem but omitted here for space to tune the parameters. RHC was given 100 attempts and 50 restarts. SA was given 30 attempts and a geometric decay. GA was given 1000 population size, .5 breed percentage, 0 mutation probability, and 10 attempts. These were chosen to keep it comparable with MIMIC in this case. MIMIC was given 1000 population and a .25 keep percentage. Refer to figure 9.

The largest problem showed the biggest struggles for MIMIC. Further experiments with much larger populations and percentages were tried but nothing seemed to improve its performance. The problem space is so large that simple algorithms like RHC seem to do well because it is easy to make gains. Finding a single neighbor with a better fitness is easy. However, creating a distribution that represents an entire set of those better fitness values is much harder with a problem size this big. MIMIC becomes intractable on larger problem sizes like this one and a simpler greedy approach must be used.

On smaller problems, MIMIC is able to achieve similar or identical fitness. However, it still requires quite a bit longer to do so. In all sizes, my hypothesis that TSP would be a good problem for MIMIC to solve ended up not true. It struggled with the scale of the problem and was unable to generate valuable probability distributions.

SA and RHC seem to be the real winners on this problem. They are both able to find an optimal value or a near optimal value in very few fitness iterations compared to the other algorithms. They gain an advantage by ignoring any structure in the problem. They simply look for a better neighbor and move there. SA is very performant because it is doing this without random restarts. It is able to explore enough by simply having a higher temperature early on.

TSP seems to be so complex that without extreme computational power, it is a hard problem to solve in a non-greedy manor. This was not expected when this problem was first chosen and is an interesting result.

3 Neural Network Training

To experiment with RHC, SA, and GA further, they are used below to train a NN on the UCI Wine Quality dataset from Assignment 1[DG17]. Performance was judged previously upon accuracy and therefore these use accuracy as well. The network structure is identical from Assignment 1. There are 4 hidden layers each with 10 nodes. These nodes use relu activation. Early stopping is used. This is equivalent to setting max attempts in the above sections for the applicable algorithms. In figure 10, loss is scaled differently for GA. This seems to be because it is taking a loss metric over the entire population and not the best. However, the behavior can still be evaluated accurately.

Random Hill Climbing Many different parameters to the optimization were tested (similar to the analysis in section 1) and the following were selected. Learning rate was .1. It was given 10 max attempts. It was allowed only 2 random restarts as this did not have a major effect on the solution. Loss never increases with RHC just like was seen with previous experiments above. Max attempts had the largest impact on learning. With low max attempts, no generalization occurred because it was not able to explore enough to find a better set of weights. With high max attempts, overfitting occurred because it was able to explore so much that it memorized the data. For this problem, a max attempt of 10 was the best.

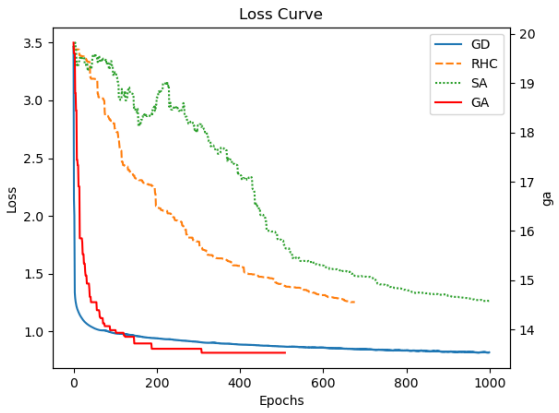

Figure 10: Credit Card Default Correlation Matrix

Simulated Annealing A baseline learning rate of .1, 10 max attempts, and geometric decay were used. Unlike RHC, loss is able to increase with SA. This also means that it is slower to learn than RHC. As is seen in the table, it is able to perform equally with nearly 3x the training time. This is opposite from previous experiments and is likely because RHC doesn't need many restarts to find a decent set of weights in this space. As the number of epochs reaches 5000 (not shown in the figure), loss actually increases again. While the temperature has cooled off almost completely, it is still able to explore upwards to try and find a better optima. RHC cannot do this. This

added exploration near the end seems to actually decrease its ability to overfit. With nearly 10x the epochs, it did not overfit. This added exploration seems to prevent memorization of the data.

Algorithm	Training Accuracy	Test Accuracy	Training Time
RHC	.57	.55	10.3
SA	.58	.56	31.8
GA	.63	.60	62.5
GD	.71	.53	12.4

Genetic Algorithm A baseline population size of 20 and mutation probability of .01 were used. The breeding percentage was .2. This provided the overall best performance for this algorithm. Similar to RHC, GA can only improve in performance per iteration. GA has a very stair step like shape unlike other algorithms. At each iteration, a new combination of weights for the network might have a better performance that the last. These then may not be updated for many generations because the new weights are not as good. GA was the most sensitive to different parameters when training a NN. A large population size over 200 would cause the algorithm to achieve close to 0% accuracy consistently. This is likely because the exploration is so great that the output of the top 20% were relatively random still. Performance could not be gained because of this near random output. GA performed the best with a training accuracy of .63 and test accuracy of .6. It also took the longest at 62.5 seconds compared to 10.3 and 31.8 for RHC and SA respectively.

Gradient Descent As a baseline, GD was used to compare with the other algorithms. GD improves much more quickly than any of the other algorithms. This is because it is actually able to find the best path downward at any given step. While a learning rate of .1 was used on the other algorithms, a learning rate of .0001 was used here due to this ability to actually find "direction". It provided an accuracy of .71 and .53 on training and test respectively. It did so in 12.4 seconds which is almost as fast as RHC. Yes, this is overfitting, but the ability to overfit this quickly shows that this algorithm is very capable of optimization for this problem.

While GA appears to be the most accurate and RHC the quickest of the 4 explored, GD is the best across the board because it is able to learn the most in the shortest amount of time.

4 Conclusion

For small problem spaces or problems where compute power is limited, RHC and SA are the best options. They provide decent results very quickly. For complex problems with varied underlying structure, MIMIC and GA are able to explore that structure and take advantage of it in a way that RHC and SA are not. They are better for those solutions if you have the time and power to use them.

References

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[NQ] Solutions for N Queens. https://www.quora.com/Does-the-N-queens-problem-have-at-least-one-solution-for-every-N-3. Accessed: 2022-10-15.

[Rol20] A. Rollings. mlrose: Machine Learning, Randomized Optimization and SEarch package for Python, hiive extended remix. https://github.com/hiive/mlrose, 2020. Accessed: 2022-10-15.

[TSP] https://math.stackexchange.com/questions/725396/how-many-routes-possible-in-the-traveling-salesman-problem-with-n-cities-and. Accessed: 2022-10-15.