Assignment 2
Dylan Lawrence
CS 7641

After testing a wide variety of problems, I decided to use a weighted knapsack problem, a six peaks problem, and an 8-Queens problem. I chose these because they do a good job at highlighting the strengths, weaknesses, similarities, and differences of each of the three algorithms we needed a problem for.

Knapsack was chosen because it does a good job at highlighting the strengths of genetic algorithms. The way it works is you have a "knapsack" that can fit a certain weight of items. Each of these items is assigned a value, and the algorithm tries to maximize the value of items it fits into the knapsack. I find this to be an interesting problem as I am currently working on a binning algorithm at work that will use an optimizer. This problem is found often in computer science in the context of scheduling jobs.

The 2nd problem I chose was the six peaks problem. I originally was testing with four peaks, but felt that it did not highlight the advantages of one particular algorithm enough, particularly simulated annealing, the algorithm I chose this problem for. I am not aware of any real life use cases that this problem applies to, but because changing one bit in the state can cause a drastic change in fitness, it does a good job highlighting the strengths and weaknesses of different algorithms.

The final algorithm 8 queens is used to highlight the advantages of MIMIC. While genetic also performs well here, it can fail to get an answer that is as fit as MIMIC's within the same amount of time, showcasing the stability of the MIMIC algorithm. This is another algorithm that is mostly used as a toy problem, but has some relevance to path finding in video games or game engines.

Knapsack

The knapsack algorithm was chosen because it highlights the strengths of the genetic algorithm. These types of problems are very common in computer science, so they have many real worls application. You could think of the knapsack capacity as the number of computations a processor could run, the weights as the number of computations a process

needs to be completed, and the value as a priorty level for each process. This would be almost identical to how many processor and load balancing algorithms work. Below is the setup for this particular knapsack problem:

Weight	10	5	3	20	15	16	3	1	11	12
Value	2	3	3	5	10	14	4	1	2	6

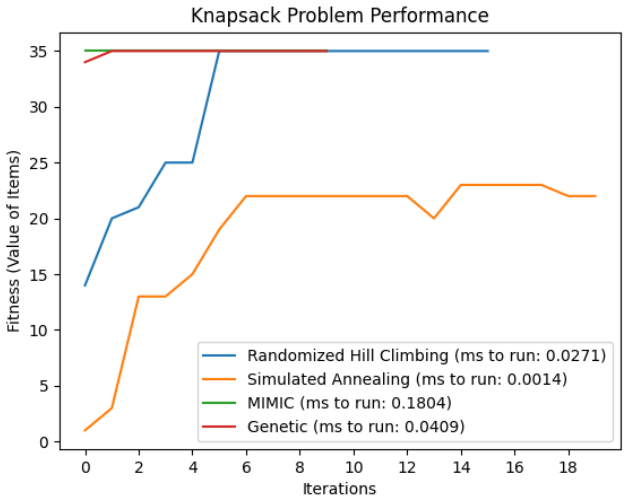
The total weight of these items is 96, and the bag has a maximum bag weight of 48. The goal is to fit the maximum value in the bag without exceeding the maximum weight.The state of current proposed solution is represented by a bit string of length 10 such as: [1, 0, 1, 0, 0, 1, 0, 0, 1, 0]. A 1 represents that the item (counted from left to right) is included, and a 0 represents that it is excluded.

All of the algorithms, with the exception of simulated annealing were able to find an optimal solution consistently. However, the genetic algorithm was faster by a significant margin.

Algorithm	**Best State**	**Fitness**
Random Hill Climbing	0 1 1 0 1 1 1 1 0 0	35
Simulated Annealing	0 1 1 0 0 1 0 1 1 1	23
MIMIC	0 1 1 0 1 1 1 1 0 0	35
Genetic Algorithm	0 1 1 0 1 1 1 1 0 0	35

As you can see, random hill climbing, MIMIC, and the genetic algorithm all ended up in the same best state. The fitness for this state was 35, while simulated annealing had a fitness of 29.

The chart below shows the fitness curve for each of the algorithms. To generate the chart I gave MIMIC and the genetic algorithm a higher max_iters value than was needed to make it more readable. As you can see MIMIC only needs one iteration to get the ideal solution, but takes much longer per iteration than the genetic algorithm. The difference in time is relatively much larger with only one or two iterations, likely due to increased overhead with MIMIC. As you can see from the graph, randomized hill climbing got the optimal solution with more iterations, but in less time than the genetic algorithm. However, when tested with a more complex knapsack setup, it did not always find the optimal solution.



The best hyperparameter configuration I found with the genetic algorithm was:

Hyperparameter	Value
max_attempts	5
max_iters	10
pop_size	100
pop_breed_percent	0.5

With these hyperparameters, the algorithm finds the optimal solution in .01 ms, far faster than even randomized hill climbing. These hyperparameters were tuned to this specific knapsack problem with the goal of maximizing performance. pop_size had a large impact on performance, and works in tandem with max_attempts and max_iters to determine the overall run time. If I lowered pop_size below 100, the results became less reliable since the algorithm would occaisonally get the wrong solution. The hyperparameters would need to be changed, and pop_size likely increased for a more complex knapsack problem.

For knapsack, the genetic algorithm is the clear winner. It consistently gets reliable results in less time than the other problems. However, MIMIC is a close competitor. This is likely due to the the different bits in the knapsack state being relatively uncorrelated to the fitness. Since a single change can push bring the fitness to 0 by overfilling, or result in a negligble change due to value and weight not being related to each other.

Six Peaks

Six peaks is a toy optimization problem often used to test optimization algorithms. While the six peaks problem doesn't appear in the real world per se, there are many examples of problems where there are multiple near-ideal solutions, and only a handful of ideal solutions. The algorithm heavily weights the number of 1s at the beginning of the state and 0s at the end of the state, depending on which has more. This

creates a correlation between bits next to each other, which should allow algorithms like simulated annealing to excel, since it will search for local optima. The formula below is modified from the docs for mlrose:

$$fitness(x) = max(tail(0,x), head(1,x)) + R(x,4)$$

$$R(x) = n, if\ (tail(0,x) < 4 \wedge head(1,x) > 4) \vee (tail(1,x) > 4 \wedge head(0,x) > 4); Otherwise\ R(x) = 0$$

My t_pct value was .2, with a state size of 20. This means that an optimal solution is going to have at least four 1s at the beginning and 4 0s at the end, or vice versa.

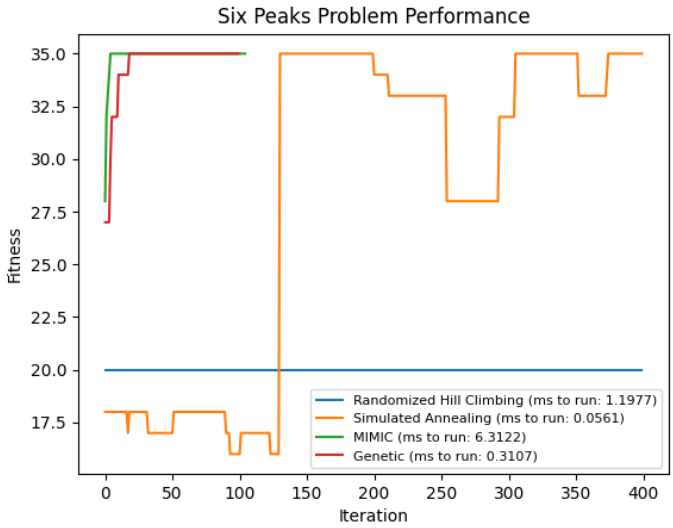
Both simulated annealing and the genetic algorithm performed well on this problem, however simulated annealing was far faster than genetic.

Algorithm	**State**	**Fitness**
Random Hill Climbing	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	20
Simulated Annealing	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	35
MIMIC	0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1	35
Genetic Algorithm	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1	35

For the six peaks problem the state with best fitness varied from column to column. Random Hill Climbing was unable to get a better state than the initial state of all 0s passed into it. This is likely due to the changing of a specific bit not making a big difference until there is a sequence of all 0s on one end, and all 1s on the other. I tried giving randomized hill climbing 10000 attempts over 100000 iterations, and unless the t_pct was set to a much lower value than .2, it was unable to find a better solution.

Genetic performing well also makes sense here since it would save population members with a sequence of 0s or 1s on each end as elite, using them to build the next generation until both sequences were achieved. Mimic was unable to find an optimal solution in most cases without giving it many iterations, increasing the runtime drastically.

Simulated annealing was able to quickly find a state with the required sequences due to it making many major modifications to the state at the beginning of its run, and then making smaller modifications until it found the ideal state of just two sequences, with both having a length greater than or equal to four. However it required many iterations to reliably get the optimal state, even though this still resulted in simulated annealing running much faster than the genetic algorithm, the performance may be worse than genetic comparatively on a problem with a larger state size, or a computer with limited memory.



Note: Simulated Annealing starts at Iteration 800

In the graph above we can see that genetic and MIMIC reached the optimal solution quickly, they took longer to run than simulated annealing. As is mentioned in the caption, simulated annealing took over 800 iterations to even begin to approach the optimal solution. However each iteration of simulated annealing is much faster than its competitors with this problem.

MIMIC would often find the optimal solution, however it did not do so reliably enough for me to consider it a legitimate contender for this problem, unless there was a specific use case where runtime wasn't a concern, and an ideal solution not required.

The following is my hyperparameter configuring for simulnd weaknesses of each approach. Think hard about this. To be interesting the problems should be non-trated annealing:

Hyperparameter	Value
max_attempts	1000
max_iters	10000
schedule	ArithDecay()
init_state	[0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]

For this problem I passed in an initial state of alternating 1s and 0s so that the algorithm was not already halfway to an ideal solution. If I passed in a state of all 0s or 1s, it found the ideal solution in fewer iterations, and failed to find the ideal solution in fewer cases. However, even with alternating 1s and 0s, simulated annealing is able to almost always find the ideal solution, and the success rate goes up with more iterations.

I initially started with a max_attempts value of 2^20 so that in theory the algorithm could check every possibility in every run, however that was unnecesary and I was able to work that value down to 1000 while maintaining an extremely high success rate. max_iters is set to 10000 because I found that to be a good balance between showcasing the performance advantage of simulated annealing, while maintaining an acceptable level of accuracy. The higher this hyperparameter is, the more likely you will be to get the optimal solution, but the longer the algorithm will take to run.

I chose to use ArithDecay() from mlrose as the decay schedule. This is a form of linear decay. I tested the algorithm with geometric decay and exponential decay. While geometric decay ran almost 6x faster, it had noticeably decreased accuracy. Exponential decay had a similar runtime to geometric decay, but noticeably increased accuracy. This makes sense since we do not want to force the algorithm into making small changes before it can work the state into having at least one correct sequence.

Modified N-Queens

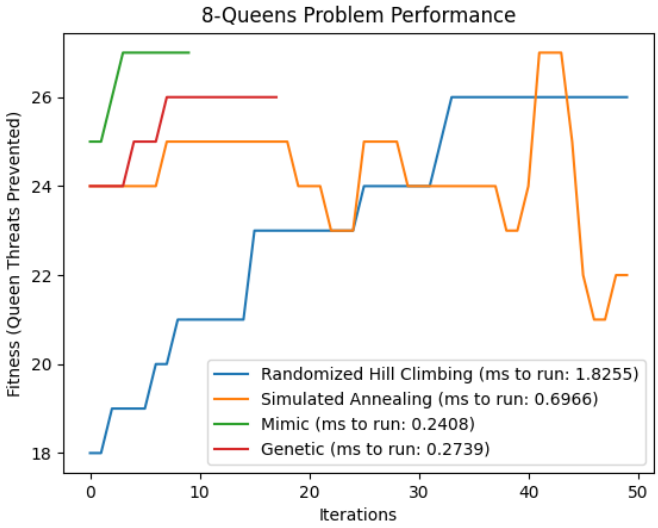
N-Queens is another toy problem that does a good job at isolating the characteristics of a problem to show the differences in algorithm performance. The goal of the problem is to place N (n = 8) queens on a NxN chessboard, without having any conflicts. However this problem is traditionally a minimization problem where you try to minimize the number of threats from each queen to other queens. To fix this problem and make it eligible for this assignment, I made a custom fitness function that subtracts 27 the result of the queens fitness function from mlrose [mlrose]. 27 is the maximum number of threats, so the values can be thought of as the number of potential threats that have been eliminated. In this problem the effect a value in the state has on fitness is highly correlated with every other value in the state bit array. This means that algorithms capable of having a high degree of randomness during the optimization process work well.

Because of the characteristics mToday at 8:11 PM entioned above, simulated annealing performs very well alongside MIMIC. However I will break down why MIMIC might be a better solution to optimizing a problem similar to this in a real world scenario. Below are the states and their fitness for each algorithm.

Algorithm	State	Fitness
Random Hill Climbing	4 6 0 3 1 7 5 2	27
Simulated Annealing	0 4 7 5 2 6 1 3	27
MIMIC	5 3 6 0 7 1 4 2	27
Genetric Algorithm	7 5 3 0 6 4 2 1	26

The state bit arrays for this problem represent the row position of the queen, starting from zero at the bottom-most row, in each column, starting from zero in the left-most column. This means state[0] = 0 would be the bottom left hand corner, and state[7] = 7 the top right hand corner. As you can see, all algorithms but the genetic algorithm achieved an optimal solution in this particulaToday at 8:11 PMr run.

An interesting note about this problem is that simulated annealing almost always gets the correct answer in a short amount of time, having a higher success rate than MIMIC. However, when the hyperparameters for MIMIC are properly tuned, it gets a near optimal, or optimal solution in much less time.



Note: Simulated Annealing starts at iteration 1650. Randomized Hill Climbing starts at iteration 25.

This graph shows that three of the algorithms slowly move closer to the solution, while simulated annealing jumps around before finding the optimal solution. We can see that while simulated annealing was able to find the correct solution, it took almost 3x as long as MIMIC. MIMIC did not find a correct solution nearly as often as simulated annealing, but I was unable to tune simulated annealing's hyperparameters so that it ran faster than MIMIC with a higher success rate. An interesting note is that simulated annealing often either got an ideal solution, or was extremely far off, while MIMIC was consistently getting at least a 26 in much less time. This would make MIMIC an ideal option for a problem like this if runtime was a concern, and not getting an optimal solution every time was acceptable.

Hyperparameter	**Value**
max_attempts	5
max_iters	5
pop_size	350
keep_pct	.2

These are the hyperparameter values I ended up settling on for MIMIC. max_attempts and max_iters had relatively little effect on the overall outcome of MIMIC, but did have the biggest impact on runtime. pop_size and keep_pct work together to determine accuracy, with pop_size still having a notable impact on runtime. This configuration aims to maximizing performance while still maintaining near full accuracy. I was also able to cut runtime in half as long as a few 25s were acceptable by lowering the pop_size down to 100 and raising keep_pct up to .5.

Neural Network Optimization

I am using the same neural network problem that I used in assignment one. The goal of this problem is to classify whether or not someone has heart disease based on a set of given data. (I will not go into metadata here due to limited space and it already being included in assignment one.) I used the sigmoid function in this assignment as well, even though I am now no longer sure it was the correct choice.

Hyperparameter	Random Hill Climbing	Simulated Annealing	Genetic Algorithm
max_iters	50000	10000	5000
max_attempts	50000	10000	1000
learning_rate	0.001000	0.001000	0.01
early_stopping	True	True	True
clip_max	3	3	3
bias	True	True	False
schedule	-	ArithDecay()	-
mutation_prob	-	-	0.2

These are the hyperparameters used for each neural network optimization problem. Unfortunately due to computing resource constrains I was not able to test the genetic algorithm with the range of hyperparameters I wanted to.

Algorithm	Random Hill Climbing	Simulated Annealing	Genetic Algorithm
Train Score	0.72	0.54	0.75
Test Score	0.73	0.58	0.74
Loss	0.62	.68	0.54
Training Time (seconds)	28	8	452

From the data we can see that randomized hill climbing and the genetic algorithm far outperformed simulated annealing. This is expected since they do make the wild swings at the beginning like simulated annealing. Interestingly, randomized hill climb performed relatively similarly to the genetic algorithm, but optimized in far less time. This is also expected as randomized hill climbing has far computational overhead (particularly in regards to memeory usage) than a genetic

algorithm. We can also see that random hill climbing sat inbetween the genetic algorithm and simulated annealing in loss.

I think this could be further improved with better hardware, which would allow me to run larger models optimized by genetic algorithms, but unfortunately that is not an option.

Overall the notable advantage in training and testing scores, as well as lower less levels found in randomized hill climbing and the genetic algorithm is likely explain by their behavior when it comes to modifying the bit array. Genetic algorithm and randomized hill climbing are good at finding a local optima, but can struggle to find the global optima in certain cases unless they are given enough time.

References

1. [mlrose] MLROSE documentation. (n.d.). Retrieved from https://mlrose.readthedocs.io/en/stable/

2. Isbell, C. L., Jr., & Viola, P. A. (1997). A Natural Actor-Critic Architecture (Technical Report No. GIT-GVU-97-11). Georgia Institute of Technology. Retrieved from https://faculty.cc.gatech.edu/~isbell/papers/isbell-mimic-nips-1997.pdf

3. Hiive. (n.d.). mlrose: Machine Learning, Randomized Optimization and Search Library. GitHub. Retrieved from https://github.com/hiive/mlrose