RANDOMIZED OPTIMIZATION

Aayush Kumar

T-SQUARE ID: akumar372

Introduction

Procedure

I used ABAGAIL for this project, a JAVA library of machine learning and artificial intelligence algorithms. For each of the Grid Searches, I fixed the iterations to 2000 for Simulated Annealing and 1000 for Genetic Algorithms and MIMIC, simply due to the computational intensive nature of each iteration. After selecting optimal parameters for each of my optimization algorithms, I compared them by running each of them repeatedly for each problem at varying number of iterations. In the case of the Neural Network Optimization Problem, I evaluated {1, 10, 50, 100, 250, 500, 750, 1000, 2000} iterations while for the rest of the optimization problems I evaluated { 1, 5, 10, 25, 500, 1000, 50000, 200000 } for Random Hill Climbing and Simulated Annealing and { 1, 5, 10, 25, 500, 1000} iterations for Genetic Algorithms and MIMIC. I have a redundancy variable in AKNeuralNetTest.java that decides how many times an individual test will be repeated/averaged.

Neural Network Optimization

Problem Overview

Dataset

As I did in our Supervised Learning Case Study, I chose to examine a Kaggle Gym crowdedness Dataset to better understand each of the Optimization Problems. In short, the dataset documents the occupancy of a campus gym over the course of an academic year, a real world situation that we students have domain knowledge of and therefore makes the problem and interesting. I wanted to see how well and quickly each optimization algorithm led the Neural Network to the correct weights that identify the same qualities that I myself do when assessing how busy the Campus Recreational Center will be. The statistics of the Gym Dataset are as follows:

GYM DATASET

Total Instances:	62184
Total Features:	10
Used Instances	20000 (15000 Train + 5000 Test)
Used Features	8

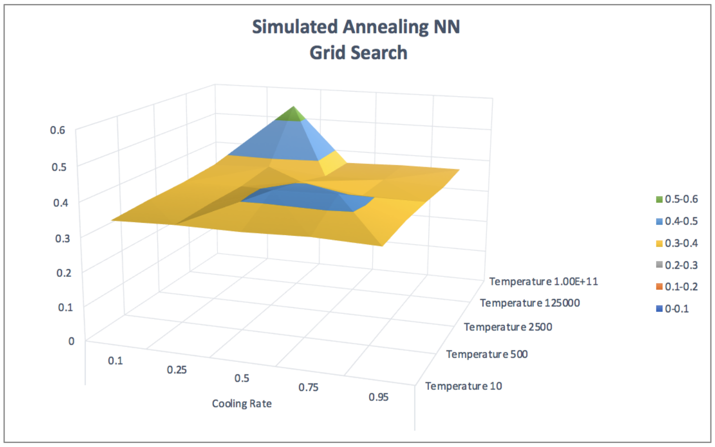
Normalization

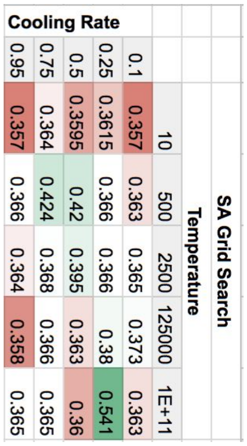
In the last project, I noticed how normalizing my data with zero mean and unit variance using Scikit Learn's StandardScaler[1] module improved performance dramatically with the MultiLayer Perceptron Classifier[2] I was using. I found that normalizing my data in fact made each algorithm perform worse consistently, something which frankly confused me. Therefore, I left the data as is and decided to run backpropagation once again, but with ABAGAIL instead of scikit-learn to account for potential differences in implementation.

Optimization Problem

The objective of this classification problem was to use attributes like the day of the week, whether or not it's a holiday, what hour of the day it is, etc. to gauge how many people are in the gym would be. Now naturally, this becomes a regression problem by its very nature where there, and therefore I decided to bin the number of people into levels of crowdedness. The data labels were originally in units of people from 0 to 160, so I replaced these labels with corresponding 20-person buckets and thereby producing labels of "0-19 people", "20-39 people", "40-59 people", etc. This way, I could make my problem a much more approachable classification task for the neural network optimization problem, where we aim to minimize the training error cost function. The Neural Net that I found to be most optimal from Project 1 had 3 hidden layers of 50 perceptrons each, an architecture I have held constant throughout this project.

Simulated Annealing: *Grid Search on Initial Temperature and Cooling Rate*

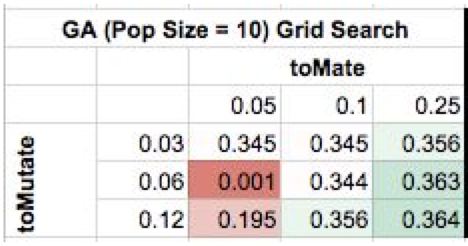




SA Grid Search

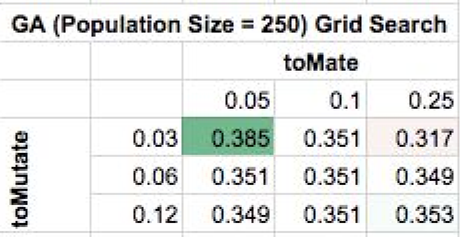
Cooling Rate	Temperature				
	10	500	2500	125000	1E+11
0.95	0.357	0.366	0.364	0.358	0.365
0.75	0.364	0.424	0.368	0.366	0.365
0.5	0.3595	0.42	0.395	0.363	0.36
0.25	0.3615	0.366	0.366	0.38	0.541
0.1	0.357	0.363	0.365	0.373	0.363

Genetic Algorithms: *Grid Search on Initial Temperature and Cooling Rate?*



GA (Pop Size = 10) Grid Search

		toMate		
		0.05	0.1	0.25
toMutate	0.03	0.345	0.345	0.356
	0.06	0.001	0.344	0.363
	0.12	0.195	0.356	0.364



GA (Population Size = 250) Grid Search

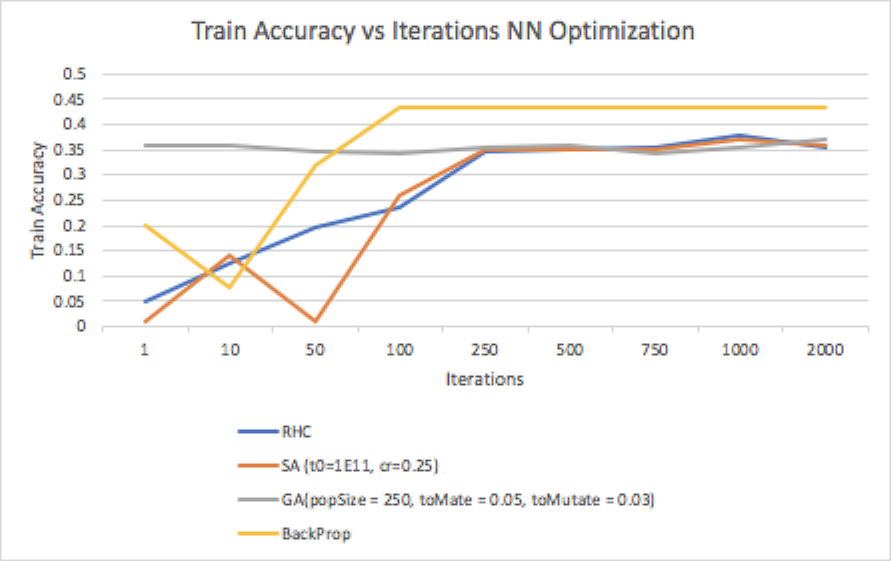
		toMate		
		0.05	0.1	0.25
toMutate	0.03	0.385	0.351	0.317
	0.06	0.351	0.351	0.349
	0.12	0.349	0.351	0.353

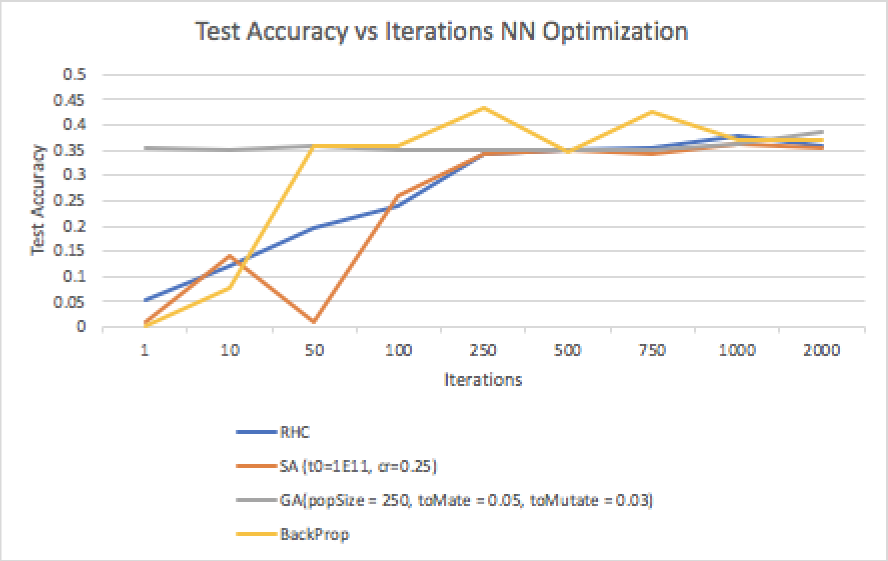
[1] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[2] http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Comparison of Random Optimization Algorithms

Accuracy vs Varying Iterations





To no surprise, backprop performed the best at training accuracy simply because the nature of the algorithm is to use gradient descent on the error function evaluation of the training data. There is much less randomness in this approach of tuning our Neural Network's weights to the best solution for explaining our training when using backpropagation, and in fact this can perhaps show that we are beginning to overfit to our training data, especially when we also consider the results of our Test Accuracy.

We see that Genetic Algorithms in this case are able to both start off the strongest and end with the best solution across all our optimization algorithms and backpropagation on the test set. This is likely due to the notion that attributes that matter in deciding whether or not the gym is crowded can have well tuned weights which become a common characteristic across population's members. These weights can then propagate through crossbreeding and mutation simply due to how influential and important they are- such an idea also explains why the Genetic Algorithm approach does not seem to improve that dramatically as a function of number of iterations run. From project 1, I found that there are not too many attributes that help recognize how crowded the gym will be, so therefore it is fair to assume that there are not too many weights that need tampering after those attributes have been weighted appropriately. From there on out, we would only get marginal improvements here and there, something which we see around the 1000 iteration mark.

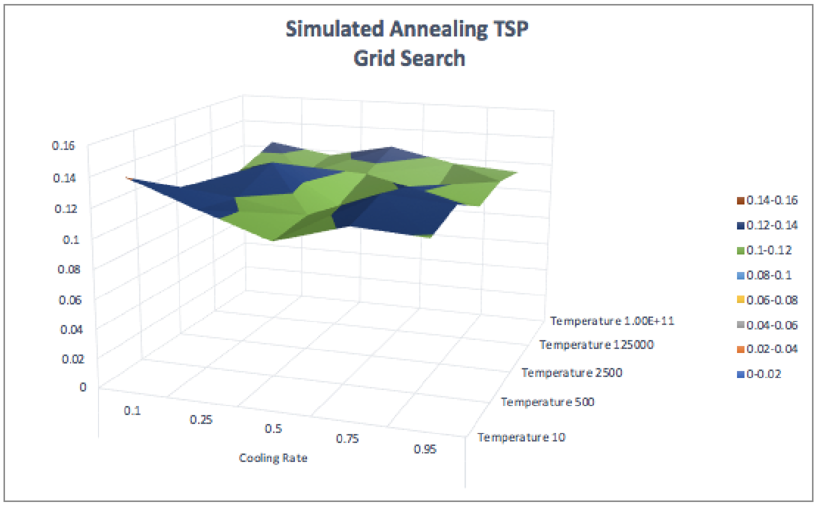
Traveling Salesman

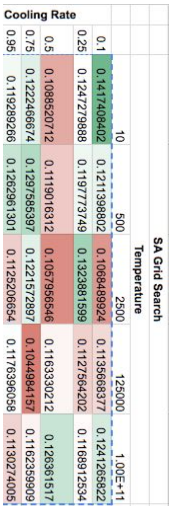
Problem Overview

The Traveling Salesman Problem involves finding the shortest path between n nodes, where each node is visited once and then cycles back to the initial position in the path. This problem is NP-Hard, and is far from trivial thus making it an excellent candidate problem for evaluating each Optimization Algorithm.

Grid Search for Simulated Annealing, Genetic Algorithms, MIMIC

Simulated Annealing: *Grid Search on Initial Temperature and Cooling Rate*

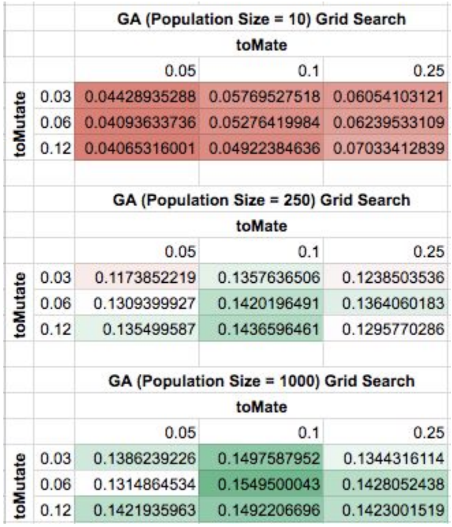




SA Grid Search

Cooling Rate	Temperature				
	10	500	2500	125000	1.00E+11
0.1	0.1417408402	0.1211398802	0.1068489924	0.1135668377	0.1241265822
0.25	0.1247279888	0.1197773749	0.1323881599	0.1127564202	0.1168912534
0.5	0.1088520712	0.1119016312	0.1057956546	0.1163330212	0.126361517
0.75	0.1222466674	0.1297585397	0.1221572897	0.1044984157	0.1162359909
0.95	0.119289266	0.1262961301	0.1125206654	0.1176396058	0.1130274005

Genetic Algorithms: *Grid Search on Population Size, % to Mate, % to Mutate*



GA (Population Size = 10) Grid Search

		toMate		
		0.05	0.1	0.25
toMutate	0.03	0.04428935288	0.05769527518	0.06054103121
	0.06	0.04093633736	0.05276419984	0.06239533109
	0.12	0.04065316001	0.04922384636	0.07033412839

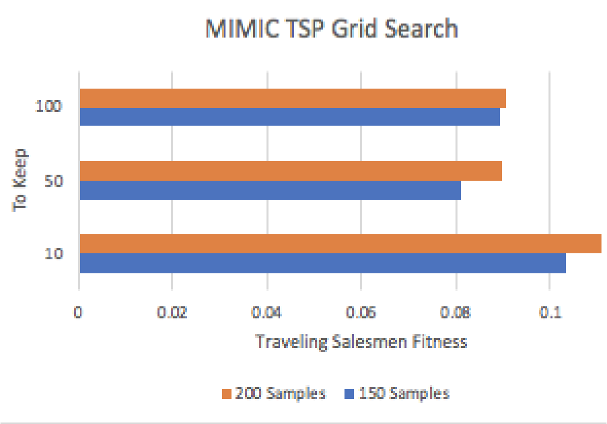
GA (Population Size = 250) Grid Search

		toMate		
		0.05	0.1	0.25
toMutate	0.03	0.1173852219	0.1357636506	0.1238503536
	0.06	0.1309399927	0.1420196491	0.1364060183
	0.12	0.135499587	0.1436596461	0.1295770286

GA (Population Size = 1000) Grid Search

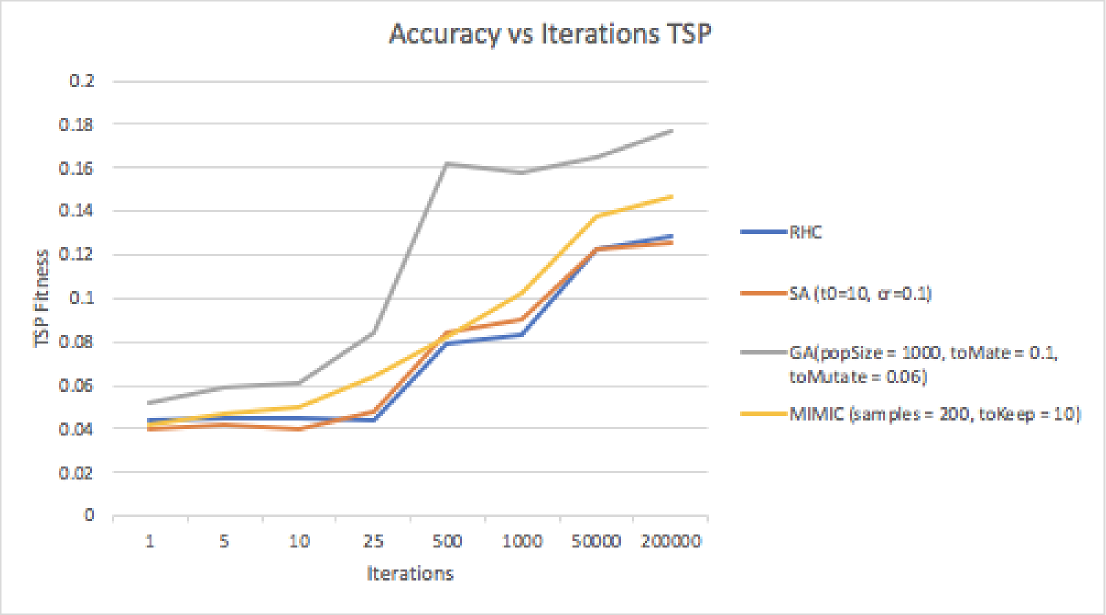
		toMate		
		0.05	0.1	0.25
toMutate	0.03	0.1386239226	0.1497587952	0.1344316114
	0.06	0.1314864534	0.1549500043	0.1428052438
	0.12	0.1421935963	0.1492206696	0.1423001519

MIMIC: *Grid Search on Sample Size, To Keep*



Comparison of Random Optimization Algorithms

Accuracy vs Varying Iterations



We see that Genetic Algorithms were able to solve the Traveling Salesperson problem the best by a large margin when compared to the other optimization algorithms. We can most likely attribute this to the notion that the most optimal path contains many of the same subpaths with other slightly less optimal paths. Therefore, we are able to crossbreed the traversals of the n nodes, which in essence can be changing only a handful of the included edges. Subpaths that are most likely part of the optimal solution will appear across many of the members of the population eventually, resulting in a high likelihood that they remain intact throughout more generations of crossbreeding. Mutation would enable candidate paths to themselves be altered slightly to explore slight variations of what was already found to be optimal at the current iteration. Overall, the problem itself is well tailored to the evolutionary approach that the genetic algorithm takes for this reason.

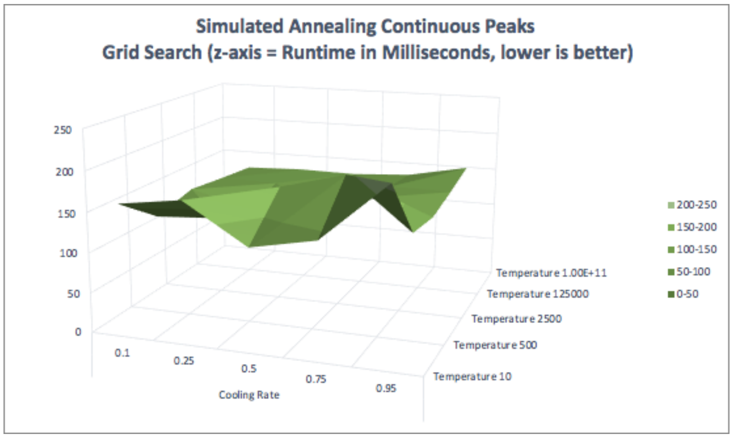
Continuous Peaks

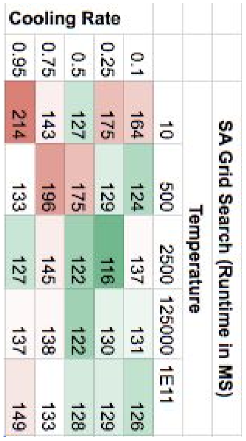
Problem Overview

The Continuous Peaks Problem involves finding the longest substring of continuous 0s or 1s in a n-length bitstring. This is a more complex version of the Four Peaks problem, which has a similar objective but fixes the continuous substring location to be at the beginning or end of the string. For both problems however, a solution constitutes containing a continuous substring of length > T for a given T <= n. More formally, the fitness function is `f(x) = max(#0(x), #1(x)) + (n if (#0(x) > T and #0(x) > T) else 0)` where `#0(x)` is the length of longest substring of 0s in x and `#1(x)` is the length of the longest substring of 1s in x.

Grid Search for Simulated Annealing, Genetic Algorithms, MIMIC

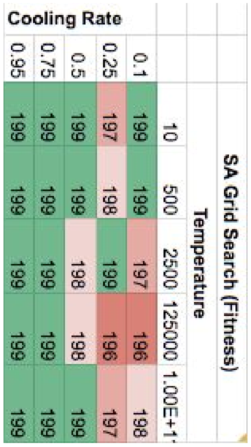
Simulated Annealing: Grid Search on Initial Temperature and Cooling Rate





SA Grid Search (Runtime in MS)

Cooling Rate	Temperature 10	500	2500	125000	1E11
0.95	214	133	127	137	149
0.75	143	196	145	138	133
0.5	127	175	122	122	128
0.25	175	129	116	130	129
0.1	164	124	137	131	126

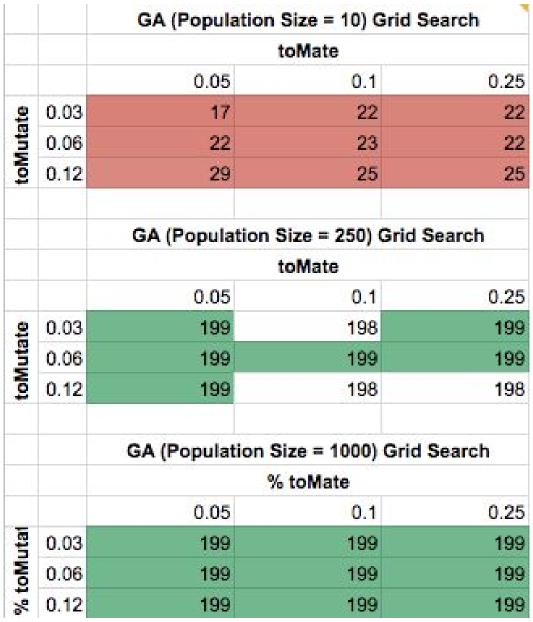


SA Grid Search (Fitness)

Cooling Rate	Temperature 10	500	2500	125000	1.00E+1
0.95	199	199	199	199	199
0.75	199	199	199	199	199
0.5	199	199	198	198	199
0.25	197	198	199	196	197
0.1	199	199	197	196	198

We find that with Simulated Annealing, the fitness function was not able to distinguish different configurations of parameters (see top right). Therefore, we chose our optimal parameters based on those which were able to reach the solution the fastest in terms of wallclock time in milliseconds (see top left).

Genetic Algorithm: Grid Search on Population Size, % to Mate, % to Mutate



GA (Population Size = 10) Grid Search

toMutate	toMate 0.05	0.1	0.25
0.03	17	22	22
0.06	22	23	22
0.12	29	25	25

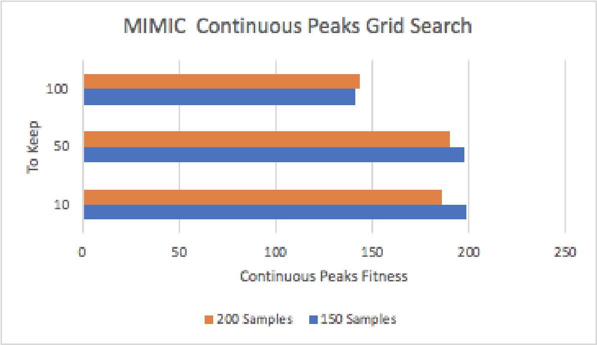
GA (Population Size = 250) Grid Search

toMutate	toMate 0.05	0.1	0.25
0.03	199	198	199
0.06	199	199	199
0.12	199	198	198

GA (Population Size = 1000) Grid Search

% toMutat	% toMate 0.05	0.1	0.25
0.03	199	199	199
0.06	199	199	199
0.12	199	199	199

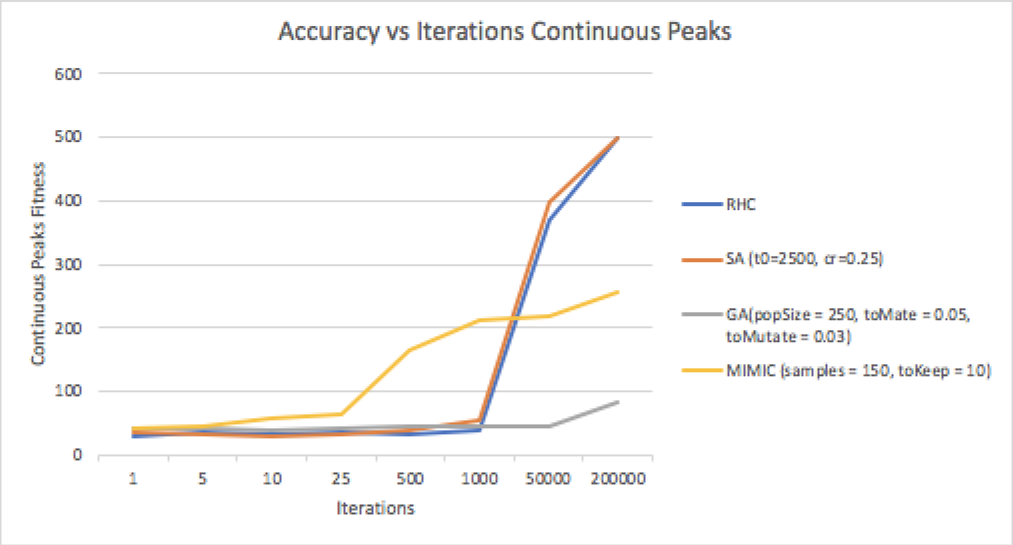
MIMIC: Grid Search on Sample Size, To Keep



Similarly, the Genetic Algorithm Optimizer was also indistinguishable when it came to choosing optimal parameters using solely the fitness function. However, runtimes were largely the same as well so I decided to choose parameters which would reduce the complexity of the problem the most. My reasoning for selecting a population size of 200 with 5% to mate and 3% to mutate was that if mutation and mating rates have little to no effect in the range that I am testing it, then we can mutate the least number of candidates since they all must have traits that enable a well performing algorithm. Likewise, we do not need to crossbreed as many candidates because it appears that most of them already have the necessary traits that make them optimal solutions.

Comparison of Random Optimization Algorithms

Accuracy vs Varying Iterations



For large values of N (N = 500) in this case, we see that Simulated Annealing does exceptionally better than any other candidate optimization algorithm. This can most likely be attributed to the fact that there are many local optima in the problem of Continuous Peaks. For instance, it would seem that a string with the middle-most T+1 bits as a Continuous Peak is better than any of those middle T+1 bits being flipped (see below). However, the global optimum would involve actually setting the continuous peaks of 0s and 1s to be at the beginning and end of the bit string to allow for more fitness.

1. `n = 11, T = 4, f(00011111000) = 5` is a local optimum compared to `f(00011011000) = 3`
2. `n = 11, T = 4, f(00000011111) = 6 + 11 = 17`

This idea of getting out of local optima is where Simulated Annealing would shine, as it's temperature factor would allow it to "jump" out of simply trying to optimize case 1 above and potentially find case 2. Randomized Hill Climbing is in close second but is less reliable on average simply due to its slightly more greedy approach of looking at close by neighbors when trying to iterate to a more optimal solution.

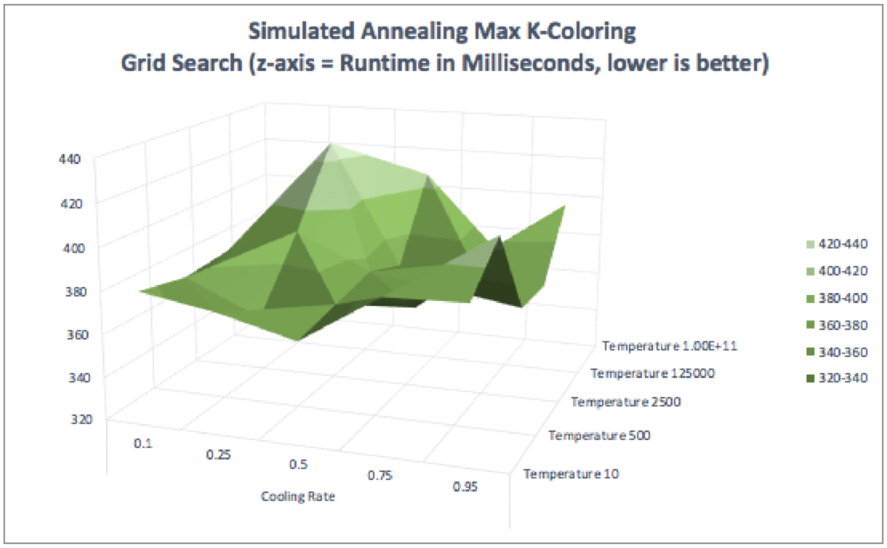
Max K-Coloring

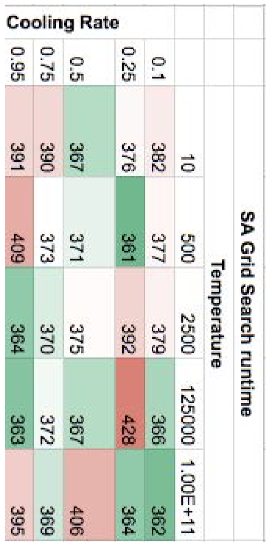
Problem Overview

This is a classic problem in applied combinatorics, where we try to use k colors to label nodes of a graph such that no adjacent nodes (nodes that share an edge) have the same color. The problem of determining whether a graph is k-colorable is NP-Complete, and Max K-Coloring is simply the problem of finding the coloring given K colors and Graph G=(V,E) that minimizes the number of same-colored adjacent nodes.

Grid Search for Simulated Annealing, Genetic Algorithms, MIMIC

Simulated Annealing*: Grid Search on Initial Temperature and Cooling Rate*



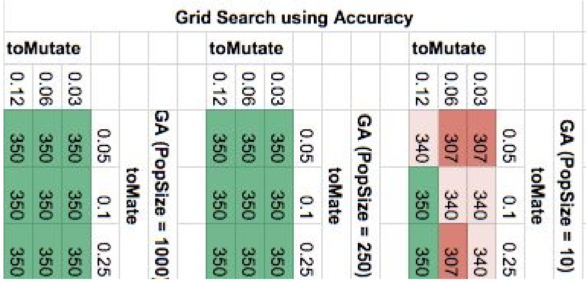


SA Grid Search runtime

Cooling Rate	Temperature 10	500	2500	125000	1.00E+11
0.95	391	409	364	363	395
0.75	390	373	370	372	369
0.5	367	371	375	367	406
0.25	376	361	392	428	364
0.1	382	377	379	366	362

All configurations of the parameters resulted in the exact same scores by the fitness function, 340. As we did with the Continuous Peaks problem, to assess which parameters were optimal I chose to measure the performance of them by their runtime in milliseconds, plotted above.

Genetic Algorithm*: Grid Search on Population Size, % to Mate, % to Mutate*



Grid Search using Accuracy

GA (PopSize = 1000)

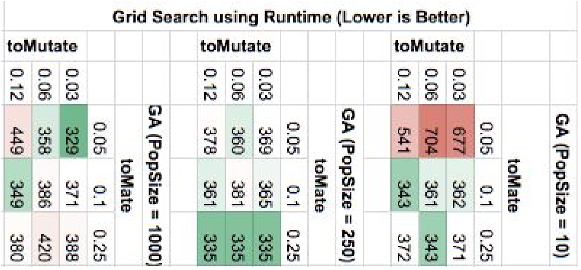
toMutate	toMate 0.05	0.1	0.25
0.12	350	350	350
0.06	350	350	350
0.03	350	350	350

GA (PopSize = 250)

toMutate	toMate 0.05	0.1	0.25
0.12	350	350	350
0.06	350	350	350
0.03	350	350	350

GA (PopSize = 10)

toMutate	toMate 0.05	0.1	0.25
0.12	340	350	350
0.06	307	340	307
0.03	307	340	340



Grid Search using Runtime (Lower is Better)

GA (PopSize = 1000)

toMutate	toMate 0.05	0.1	0.25
0.12	449	349	380
0.06	358	386	420
0.03	329	371	388

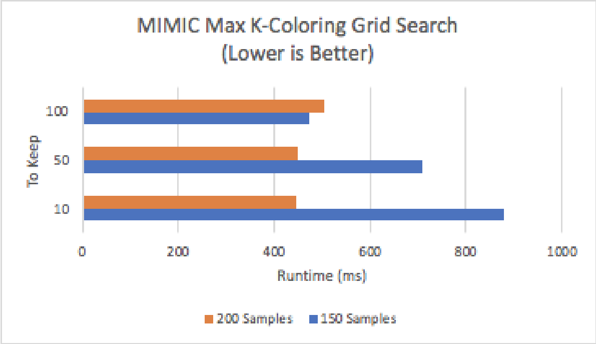
GA (PopSize = 250)

toMutate	toMate 0.05	0.1	0.25
0.12	378	361	335
0.06	360	381	335
0.03	369	365	335

GA (PopSize = 10)

toMutate	toMate 0.05	0.1	0.25
0.12	541	343	372
0.06	704	361	343
0.03	677	362	371

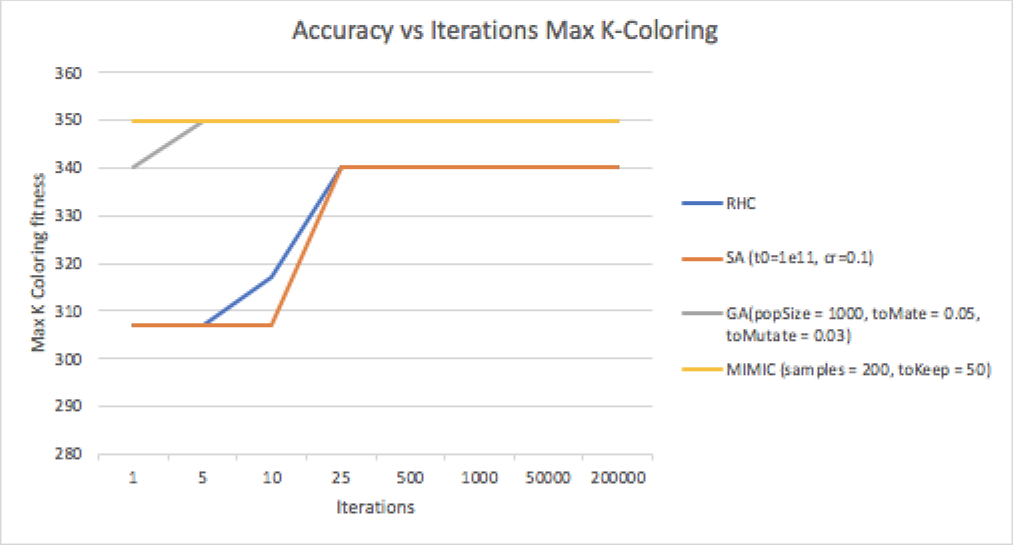
MIMIC: *Grid Search on Sample Size, To Keep*



For Genetic Algorithms, again because the fitness function failed to distinguish the optimal parameters across many configuration, I analyzed the runtime of each of them and found the fastest configurations for the genetic algorithm. In this case, there was still a three-way tie across 3 different mutation rates, which I broke by selecting the configuration with the lowest mutation. My reasoning for this was simple, if mutating the population more and more did not seem to be helping then, leaving our population and it's mating process as is would be the simpler route rather than adding the complexity of more unhelpful mutation. Similarly, the evaluation function did not yield conclusive results of which parameters were most optimal for MIMIC, so I once again assessed each configurations performance based off of runtime. Thus, once again, lower runtimes are better in the above bar chart.

Comparison of Random Optimization Algorithms

Accuracy vs Varying Iterations



Here we can clearly see that MIMIC does a great job of quickly finding the solutions within few notorious. Some might argue that it is too computationally expensive each iteration to be considered the best algorithm in this case, but in practice it only ran approximately 100 ± 4.7 ms slower than the other algorithms for this problem. More importantly, MIMIC accomplishes much more in fewer iterations- we see that within a single iteration it already found an answer more optimal than anything Random Hill Climbing and Simulated Annealing would ever find even after 200000 iterations. In fact, both Randomized Hill Climbing and Simulated Annealing completely failed to find a valid coloring (one where no adjacent vertices have the same color). Meanwhile, the Genetic Algorithm optimizer and MIMIC did a much better job of retaining previous progress either by evolution or an updated probability distribution of well performing candidates. Intuitively, this can be because there most likely exists some underlying structure to the graph coloring- in fact, we know that any K-complete subgraph within the graph has to be colored with at least K colors because each node in a K-complete graph is connected to all other K-1 nodes. Solving subproblems like this can be why Genetic Algorithms do better, but MIMIC still outperforms because it is able to understand more parts of the common underlying structure of optimal solutions than the evolutionary process of the Genetic Algorithm.

Conclusions

This case study has illuminated some very useful aspects about the Randomized Optimization algorithms we studied- for one I have come to realize that Genetic Algorithms and MIMIC tend to perform similarly well and similarly poorly due to their common ability to identify many optimal candidates in each iteration of the algorithms. This class of algorithms does well when understanding the underlying structure of optimal solutions is key to finding the global optimum. In contrast, Simulated Annealing and Random Hill Climbing appeared to perform similarly well and similarly poorly because they both evaluate one potential candidate at a time before probabilistically jumping to another candidate. This class of algorithms does well when there are very prominent local optima dispersed throughout the hypothesis space of a problem. When it comes to things I would like to change, I do wish that the ABAGAIL implementation of Random Hill Climbing had random restarting, which would in turn help avoid getting stuck in local optima once the neighborhood range becomes too narrow. Frankly apart from the hyperparameter tuning via grid search for each of the algorithms, I don't believe there is all that much I would alter however in each algorithm to improve performance. It would be worthwhile to see how these randomized optimization algorithms perform when using Scikit-Learn's Library and normalized data, as the combination of these two appeared to perform much better overall in baseline backpropagation.

Citations

Algorithm Implementations

1. https://github.com/pushkar/ABAGAIL
2. http://scikit-learn.org/stable/index.html

Papers

3. https://www.cc.gatech.edu/~isbell/papers/isbell-mimic-nips-1997.pdfl
4. https://www.cc.gatech.edu/~isbell/tutorials/mimic-tutorial.pdf

Dataset

1. https://www.kaggle.com/nsrose7224/crowdedness-at-the-campus-gym