Chapter 1

Graph Based Genetic Algorithms

Contributed by Kyle M.D. Sweeney

1.1 Introduction

Genetic Algorithms are fundamentally a searching algorithm for finding "good" solutions when the solution space is excessively large. Many problem spaces, such as NP-Hard or NP-Complete problems, are difficult because the possible solution space grows exponentially as the input size of the problem increases. Consequently, there are no guaranteed easy solutions that can be found in reasonable time. Searching algorithms, such as Simulated Annealing and Genetic Algorithms look to nature to find methods of finding reasonably good solutions. In the case of Genetic Algorithms, solutions are found by simulating evolution, pursuing a survival of the fittest approach.

Let's take the classic travelling salesman problem [10] where a saleswoman would like to travel from city to city, visiting each city only once, and taking the shortest route. The problem is classically known to be NP-Hard. There are $N!$ possible combinations of routes to search through. To solve the problem as a genetic algorithm, we can imagine the solution, an ordered list of cities, to be like "DNA", and each city is a gene. When organisms breed, they swap genes, and thus produce new, unique children which may or may not be fitter. Genetic Algorithms require a fitness function in order to sort out which solutions are moving towards a "good" solution, and are better than other solutions. In this case, the fitness function is the cost of the trip, given the ordered list of cities. In each generation, we produce a certain number of children from the solutions in the specimen pool, add them to the pool, and then only keep a certain number which are most fit, according to the fitness function. Eventually, we choose to stop, and the most fit function is our "good" solution.

Of course, while exploring the natural extrema of the solution space, it's possible for our solutions to get stuck around a local extrema. What this means is that our solution specimens have become too homogenized, and there's not enough unique variations to choose from. In genetic terms, there's not enough genetic variation. One possible solution to solve this is via mutations. By introducing mutations during the breeding stage, solutions can jump from one area of the solution curve to another, ideally pulling the rest of the gene pool away from a local extrema, and back on the path towards a better, more optimal solution. But this genetic variation has to be carefully controlled. Too much mutations, and the pool can never stabilize and never travel along the curve. Too little, and mutations don't introduce enough variability.

Another possible solution to this issue is to control the breeding process via graphs. By placing a solution at the node of each graph, then the only possibly breeding partners are those who are neighbors of a given vertex. The idea is to simulate having different groups of solutions preserve different genetic lines. For example, in nature, a single species can be found in many different parts of the planet, but they adapt to their environment via their genetics. Occasionally cross-breeding occurs, refreshing the genepool of both groups by introducing new genetic material.

In this paper, we wished to apply this methodology to the problem of genetic harmonization, expanded upon in section 1.3. Here we have two different species whose codons, that is the group of 3 nucleotides which code for a specific amino acid, produce their amino acids in different rates. We have a DNA sequence which encodes a specific protein in one species, and we wish to find a synonymous DNA sequence in the other species which produces the amino acids at roughly the same rates.

1.2 The Problem as a Graph

The problem of shrinking genetic variation can be partially solved by mutations, but can also be solved by the introduction of graphs into the problem space. In nature, the same species can be found in multiple places around the world, yet are still breed-able with one another. These groups are genetically similar to one another, and distinct from their cousins in different pockets in different environments. For the purposes of solving a problem like the traveling salesman, we can employ graphs by placing a single solution on each node to take advantage of community isolation while permitting limited genetic-crossover. Ideally, this means that each sub-group will develop a unique solution and by crossing over, they can help push the other groups towards more optimal solutions. The effectiveness of this approach in speeding up/improving solutions comes from a combination of the right kind of graph for the problem being solved.

1.3 Some Realistic Data Sets

To demonstrate integrating graphs as helpers in genetic algorithms, the rest of this chapter will focus on the application of Genetic Algorithms in finding ideal complementary codon-sequences to generate protiens in non-human cells at human-rates.

Every protein is comprised of Amino Acids, built inside of cells according to DNA [7]. Inside of a DNA strand, three nucleotides are strung together to form a codon, which then codes for either a specific amino acid, is a stop marker, or is a start marker. Each codon is used with a certain amount of frequency, and these frequencies are species specific. Work done by Clark et al. [2] discuss the implications of these frequencies, and work done by Rodriguez et al. [1] demonstrates an algorithm for harmonizing DNA sequences between humans and a targeted species. These papers discuss a method, where given a DNA sequence, and the frequencies for different species, a series of scores for each codon, based off of those frequencies. These scores can be seen as a function over the codon positions. While the same protein is constructed from each sequence, the "human" function and "bacteria" function could be very different. Harmonizing the DNA, in this case, means altering which codons are chosen in the bacteria so that the resulting "bacteria" function will be as similar to the "human" function as possible.

Solving this harmonization problem via genetic algorithms can be done by imagining the solution space as the chosen sequence of codons which still produce the same protein. The fitness function would then be the difference in the area between the two functions when plotted out. By minimizing the distance between the two functions, a harmonization can be accomplished.

The production rates and specific DNA sequence was obtained via a prior project done in collaboration with Gabriel Wright, one of the authors of the Rodriguez et al paper [1].

For our graphs, we employed two classes of graphs: ones with a variable node size, and one with a fixed number of nodes. In the fixed class, we employed the dodecahedral graph which emulates a dodecahedron [9], as well as the Desaugres graph, which has 20 nodes, each having 3 edges [5].

In the other class, the variable node size, we had five different types. The first was a complete graph where each node connects to every other node [4]. The second type was a 2D grid or lattice graph; each node had on average 4 neighbors, with the exception of the edge nodes [8]. The third type was a Caveman graph which is N clusters of *K-Cliques* [3]. The fourth type was a Windmill graph, a graph with N clusters of *K-Cliques* and where each node is connected to a single, central node [11]. The fifth type was an Erdos-Renyi graph, where each node's edge to every other node has an p chance of existing [6].

1.4 Graph Based Genetic Algorithms-A Key Graph Kernel

When we apply graphs to isolate the breeding pairs of each potential solution, we perform a sort of graph "kernel" by traversing the graph to each of the neighbors from every node. The rough psudeo-code looks something like 1. For each vertex in the graph, breed it with every neighbor that it has. Of all these children, the most fit one will replace it after breeding has finished. Thus, in every round, only the most fit specimens remain.

The evaluation of this would be to test this algorithm on different graphs, measuring for speed and best solution score.

Algorithm 1 Graph Based Breeding:
$G,\ V,\ E$

```
1:  procedure BREED(G, V, E)
2:      $R = \{\}$
3:      for $v$ in $V$ do
4:          $N = Neighbors(v)$
5:          for $n$ in $N$ do
6:              $C = children(n, v)$
7:              for $c$ in $C$ do
8:                  if $fitness(c) < fitness(v)$ then
9:                      $R+ = (c, v)$
10:             end for
11:         end for
12:     end for
13:     for $r$ in $R$ do
14:         $replace(G, r[1], r[0])$
15:     end for
```

In our kernel, we define breeding as the process of combining different parts of two solutions together to obtain many different "children" solutions, each equally valid solutions. In our case of using this process to find a homogeneous DNA sequence in our target species that is most similar to the given DNA sequence in our reference species. Our solution is represented by a list of codons. Thus each node has a single solution, or list of codons. Children are bred by traveling down the list of both parent solutions, and choosing to take the codon from one of the parents for that position in the list. In our implementations, we only breed 10 kids. The first two are chosen by splitting the parents in half and mixing and matching these halves, forming two children. The second two trade off ever other parent's codon. This is done effectively twice. The fourth two do the same, but switch every 3rd item. The fifth two do the same, but every 10th item.

Our fitness function is a modified version of the minmax function from the work done by Rodriguez et.al [1]. This function gives a score to each codon in the DNA sequence. Our fitness function adds up the difference in the score of the reference DNA sequence and our solution sequence, with a scaling factor for easier readability of the score.

As the kernel is a single round of a genetic algorithm, we perform it 50 times, starting at the 3rd line in the psudo-code.

An assumption of the replace function is that no duplicate solutions will be placed into the graph. If the replaced solution is already on a different node, then the replacement doesn't happen.

1.5 Prior and Related Work

Much of this chapter will be applying the work done by Ashlock et al. in their 1999 paper "Graph Based Genetic Algorithms", where they took different kinds of graphs and applied them to three genetic algorithms problems. They explored the time it took to solve different genetic algorithm problems using many different kinds of graphs. This paper focuses on taking that idea and applying it to the specific problem of bioharmonization.

1.6 A Sequential Algorithm

The Kernel proposed above, if taken to be sequential, would have a rather complex execution time, dependent on the execution time of each of the underlying functions, specifically *children* and $fitness$. In the worst case scenario, a fully connected graph, the execution time is $O(V^2SC)$ where V is the number of vertices in the graph, S is the size of a given solution, and C is the number of children produced by *children*. In our specific case, this became $O(V^2S)$ as the number of children generated for each breeding pair was always 10. The pseudo-polynomial nature of the solution is the power of the genetic algorithm, as a good chunk of the solution space is evaluated, but done in an algorithmic manner.

1.7 A Reference Sequential Implementation

Evaluation of this approach was done in Python3, ran using PyPy3 to speed up execution. Our graphs were then generated in NetworkX.

1.8 Sequential Scaling Results

Using a DNA sequence of 155 Codons for a protein in E.coli, we harmonized it in C. elegans, M. musculus, H. sapien, and S. cerevisiae. In our variable node sized graphs, we did a round using 20 nodes, and another with 40 nodes. In our resulting tables, both a lower time and a lower score is better. We tested on a desktop with an Intel Core i7-5960X@3GHz with 16 logical cores, and 32GB of memory on Windows 10, using WSL Ubuntu 14.04.5 LTS. For every test, we ran it 10 times, taking the standard deviation and mean of the results.

As can be seen in section 1.14, the general trend is spending more time running the genetic algorithm, the better the solution. But the results are diminishing returns. For example, in the fully connected case of S. cerevisiae, we get a score of 19127 using 20 nodes, taking 1106s to run.

Bumping that up to 40 nodes, we only get a score of 17748, but it takes 4 times as long with 4408s. Using a shorter running example, using only 20 nodes with 50% chance, the Erdos Renyi graph had a mean runtime of 525s and an average score of 19755, which is hardly worse than the fully connected graph, yet only takes half the runtime. The graphs themselves only limit the possible mating pairs. While this decreases the runtime, often by an order of magnitude when choosing a different graph than a fully connected one, the scores are substantially worse. While the mean score of the fully connected graph is often within 1.5 standard deviations of the other graph types, the reverse is not true.

Most of the tests were undertaken with the pattern of each solution breeding with every partner it could. There are other breeding patterns, such as breeding only with the best partner, or breeding with a random partner. Doing a round of tests using this last pattern, choosing a random partner from the list of available ones, using 40 nodes. Again, here we find that choice of graph had very little impact on either the solution or the runtime. They all ran in roughly the same amount of time and achieved roughly the same score.

1.9 A Parallel Algorithm

1.10 A Reference Parallel Implementation

Discuss here an implementation of the basic parallel code. Include what language/paradigm you used for the code.

1.11 Parallel Scaling Results

Discuss here results from parallel algorithm. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Ideally plots of performance vs BOTH problem size changes AND hardware resources are desired. Did the performance as a function of size vary as you predicted?

1.12 Conclusion

Summarize your paper. Discuss possible future work and/or other options that may make sense.

1.13 Response to Reviews

Second Iteration:

In one review, the reviewer pointed out how there were no graphs mentioned in data set, nor where the data is coming from. That was addressed. The reviewer also commented that breeding and fitness functions were a bit unclear, so that was expanded upon, as well as saying when the overall algorithm ends. I cleared up that this project is applying an idea from an earlier paper to a specific problem, as that was another complaint by the first reviewer. Some bugs were fixed in the pseudo code, as pointed out by the reviewer, and a clarification of uniqueness of solution in the replacement. I also made a small change to 1.2 updating that each solution was placed at a node. I expand on what this means in section 1.4.

The second reviewer had similar complaints, and thus are also addressed.

1.14 Sequential Scaling Results - Results

Fully Connected - 20 Nodes

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	1106.064027	84.105426	19126.919	1283.620401
H. sapien	1025.578735	02.427339	24928.944	1606.4
M. musculus	1020.771278	02.082315	21137.479	2176.747175
C. elegans	1041.69969	00.767798	23789.236	1482.189131

2D Grid - 20 Nodes

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	112.497632	0.603611	25786.073	2065.608277
H. sapien	108.761492	0.569377	28664.414	2154.667222
M. musculus	107.421994	0.119317	26082.867	2162.381269
C. elegans	110.420743	0.108324	26082.867	1815.084957

Windmill Graph - 4,5

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	223.842559	1.001441	23436.453	2572.074629
H. sapien	215.994595	0.466179	28287.147	1813.922553
M. musculus	216.386268	1.20397	24182.217	1931.304768
C. elegans	224.740323	1.584885	27756.876	2479.184106

Caveman Graph - 4,5

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	223.976335	2.085542	25257.561	2122.087532
H. sapien	217.215015	0.891218	30374.471	1695.429033
M. musculus	220.781013	0.234819	26713.461	1856.30301
C. elegans	223.300958	0.234594	28866.525	1547.000103

Erdos-Renyi - 20,0.5

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	525.188143	34.916041	19755.386	1891.456603
H. sapien	510.376475	34.632428	26097.224	1983.388064
M. musculus	525.785302	57.888515	21797.674	1586.296764
C. elegans	602.725836	68.400198	25190.05	1644.452654

Desargues

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	169.624232	0.762033	22348.231	1677.688622
H. sapien	177.160388	0.500322	27590.251	2675.609935
M. musculus	164.078059	0.545794	24807.326	2002.273596
C. elegans	167.43908	0.325993	27069.238	2175.457151

Fully Connected - 40

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	4407.949818	180.660595	17747.837	1073.674007
H. sapien	4166.038358	030.571147	22628.39	1405.021405
M. musculus	4203.88681	108.770135	19826.178	2073.063743
C. elegans	4295.030674	046.220033	20571.038	1319.340475

2D-Grid - 40

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	444.243351	24.377109	21019.449	2326.198972
H. sapien	391.372798	33.135986	23924.596	1550.27023
M. musculus	373.068643	00.282972	23104.234	1864.526685
C. elegans	373.068643	00.593986	24259.694	1947.400487

Windmill - 4,10

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	0981.594376	6.547543	19369.346	2047.492682
H. sapien	0961.863631	2.460456	24557.848	2047.492682
M. musculus	0977.074179	0.66965	21314.179	2339.926058
C. elegans	1003.081456	4.429312	23416.692	2092.55662

Caveman - 4,10

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	1164.003078	122.147287	20447.382	1357.424586
H. sapien	0997.866954	000.791199	25169.657	0849.044711
M. musculus	0984.32995	001.111364	21371.185	1739.21804
C. elegans	1010.327206	000.632495	24271.767	1496.809353

Erdos-Renyi - 40,0.1

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	454.887323	48.710071	20182.799	1593.168119
H. sapien	421.002154	31.075691	25366.729	2271.80358
M. musculus	418.624505	44.980772	21653.777	1181.243829
C. elegans	432.977019	33.668121	24098.38	2721.854925

Random Mate- Fully Connected - 40

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	112.914445	0.692567	25491.094	2011.471445
H. sapien	109.614343	0.530169	27581.154	1917.922058
M. musculus	109.074456	0.139277	25906.26	2120.395358
C. elegans	111.604002	0.19365	28398.73	1873.491964

Random Mate- 2D Grid - 40

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	117.153094	0.237125	25994.986	1969.151356
H. sapien	112.069839	0.209047	27425.349	1155.710915
M. musculus	113.049799	0.12099	26199.627	1720.679499
C. elegans	116.833984	0.141174	29231.478	2798.136781

Random Mate- Windmill Graph - 4,5

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	105.765696	0.399097	25199.198	2081.407954
H. sapien	104.575735	0.573102	27805.829	1347.823895
M. musculus	105.645179	0.501267	25258.195	1135.440949
C. elegans	108.486528	0.629473	29777.532	2081.029823

Random Mate- Caveman Graph - 4,10

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	114.436177	0.55969	25539.219	1705.047569
H. sapien	111.138927	1.669333	29753.6	2519.051726
M. musculus	108.949419	0.258024	26960.741	1125.010804
C. elegans	112.650083	0.096759	28374.057	1614.289676

Random Mate- Erdos Renyi - 40,0.1

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	110.97712	2.371762	27572.692	1414.054499
H. sapien	107.17137	2.95958	28022.321	1545.729126
M. musculus	109.823711	2.074018	26623.335	2014.636918
C. elegans	112.247242	2.680797	28052.558	2186.395163

Bibliography

[1] Scott Emrich Patricia L. Clark Anabel Rodriguez, Gabriel Wright. %minmax: A versatile tool for calculating and comparing synonymous codon usage and its impact on protein folding. *Protein Science*, 1(27):356–362, 2018.

[2] Thomas F. Clarke, IV and Patricia L. Clark. Rare codons cluster. *PLOS ONE*, 3(10):1–5, 10 2008.

[3] Eric W Weisstein. Caveman graph – from MathWorld–a Wolfram Web Resource.

[4] Wikipedia contributors. Complete graph — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].

[5] Wikipedia contributors. Desargues graph — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].

[6] Wikipedia contributors. Erdsrnyi model — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].

[7] Wikipedia contributors. Genetic code — Wikipedia, the free encyclopedia, 2018. [Online; accessed 14-September-2018].

[8] Wikipedia contributors. Lattice graph — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].

[9] Wikipedia contributors. Regular dodecahedron — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].

[10] Wikipedia contributors. Travelling salesman problem — Wikipedia, the free encyclopedia, 2018. [Online; accessed 14-September-2018].

[11] Wikipedia contributors. Windmill graph — Wikipedia, the free encyclopedia, 2018. [Online; accessed 20-November-2018].