Chapter 1

Graph Based Genetic Algorithms

Contributed by Kyle M.D. Sweeney

1.1 Introduction

A Genetic Algorithm is 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 [12] 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 vertex 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 [2]. 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 gene pool 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 an real DNA sequence (as compared to the "DNA" sequence a genetic algorithm uses) which encodes for 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 environments. For the purposes of solving a problem like the traveling salesman, we can employ graphs by placing a single solution on each vertex 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. Edges only representing a possible breeding pair.

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 proteins in non-human cells at human-rates.

Every protein is comprised of Amino Acids, built inside of cells according to DNA [9]. 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. [3] discuss the implications of these frequencies, and work done by Rodriguez et al. [1] demonstrates an algorithm for harmonizing DNA sequences between a source and a targeted species. These papers discuss a method where given a DNA sequence, and the frequencies for a species, a score can be calculated for each codon based off of those frequencies. We will interpret these scores as a function over the codon positions. While the same protein is constructed from each sequence, the "source" function and "target" function could be very different. Harmonizing the DNA, in this case, means altering which codons are chosen in the "target" so that the resulting "target" function will be as similar to the "source" 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 graph selection, we employed two classes of graphs: ones with a variable vertex size, and one with a fixed number of vertices. In the fixed class, we employed the dodecahedral graph which emulates a dodecahedron [11], as well as the Desaugres graph, which has 20 vertices, each having 3 edges [7], see figure 1.1.

In the other class, the variable vertex size, we had five different types. The first was a complete graph where each vertex connects to every other vertex [6]. The second type was a 2D grid or lattice graph; each vertex had on average 4 neighbors, with the exception of the edge vertices [10]. The third type was a Caveman graph which is N clusters of *K-Cliques* [5] are connected together by moving one edge from clique to connect to another clique, see figure 1.3. The fourth type was a Windmill graph, a graph with N clusters of *K-Cliques* and where each clique shares a single, central vertex [14], see figure 1.2. The fifth type was an Erdos-Renyi graph, where each vertex's edge to every other vertex has an p chance of existing [8].

The Dodecahedral and Desaugres graphs are chosen as they are classic graph types, and we wished to see if they had an impact on the genetic algoirthm. The Erdos-Renyi was similarly chosen. The Lattice was chosen as it majorly restricts the breeding partners, but it doesn't necessarily have a corresponding real-world example for breeding, thus we wished to see if it would have an interesting effect. The Windmill and Caveman graphs were chosen specifically for employing tight cliques. One hypothesis we had was to create miniature groups which would go after different extrema points in the solution space and then cross over DNA every once in a while.

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 Evolution

```
procedure EVOLVE-SINGLE-GENERATION(G, V, E)
    $R = \{\}$
    for $v$ in $V$ do
        $N = Neighbors(v)$
        for $n$ in $N$ do
            $C = children(n, v)$
            $C.sort()$
            if $fitness(C[0]) < fitness(v)$ then
                $R += (c, v)$
        end for
    end for
    for $r$ in $R$ do
        $replace(G, r[1], r[0])$
    end for
```

In our kernel, we define breeding as the process of combining different parts of two solutions

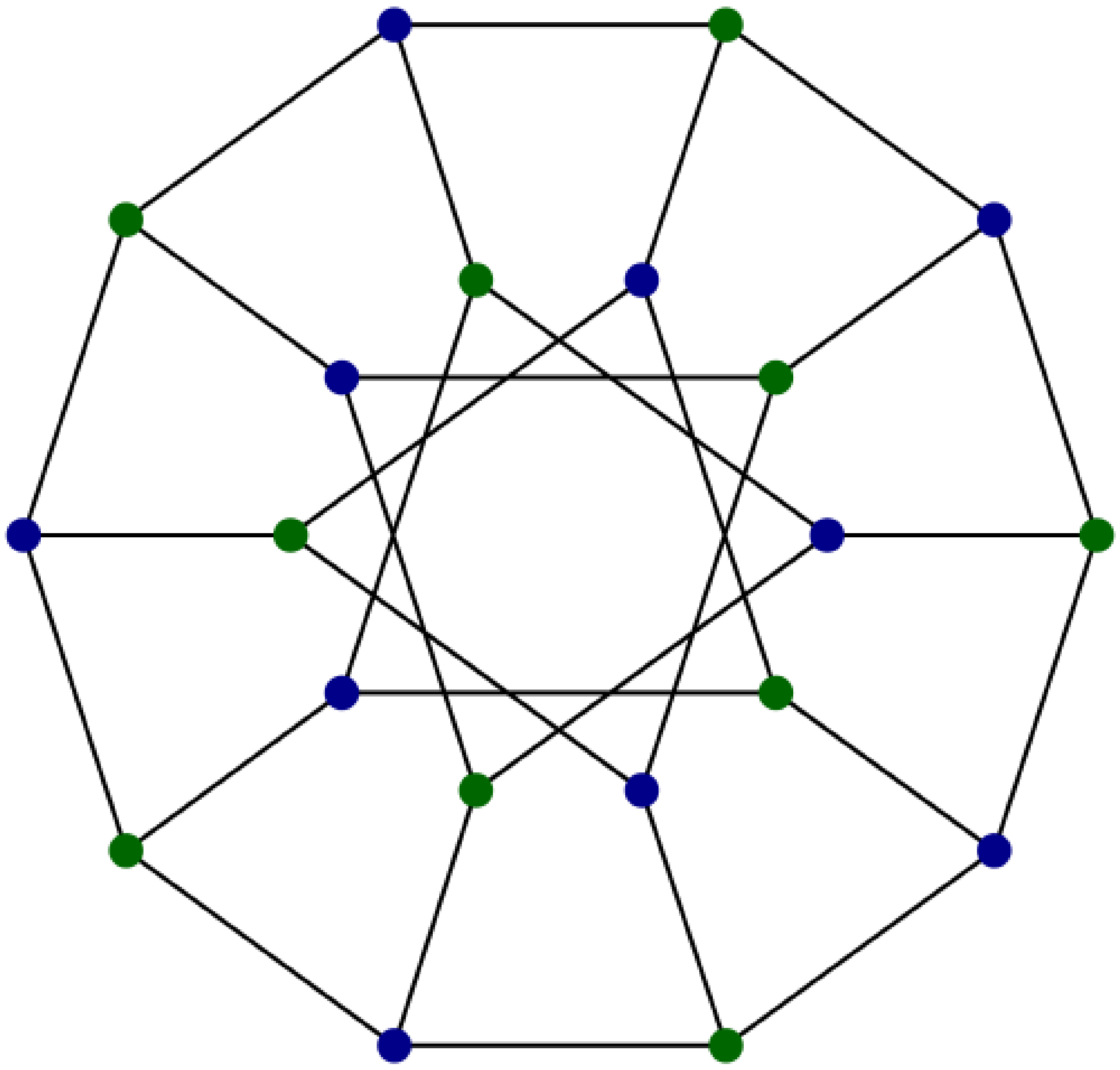


Figure 1.1: The Desargues graph. By David Eppstein - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2258499

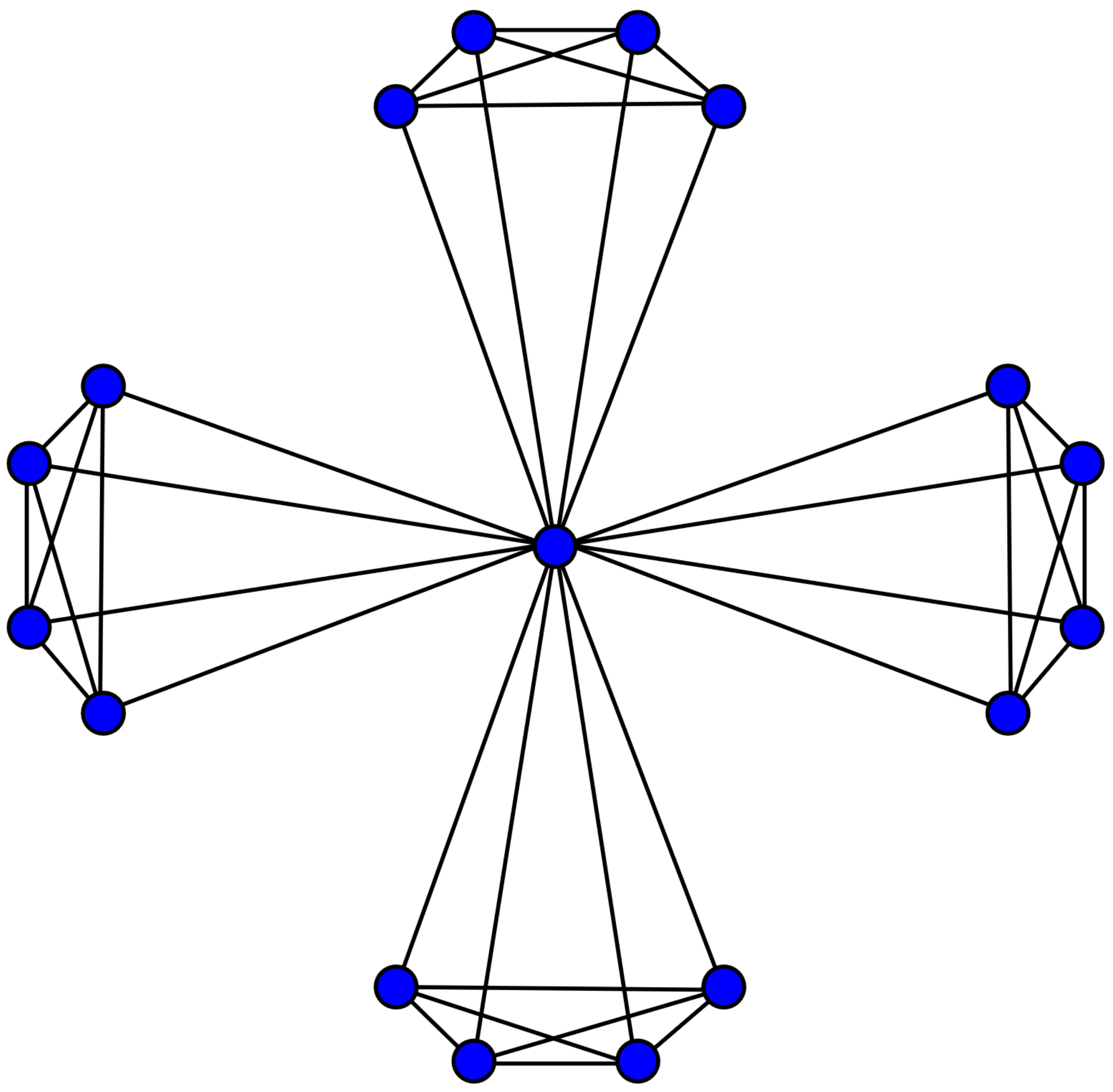


Figure 1.2: An example windmill graph of 4,5. By Koko90 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7833133

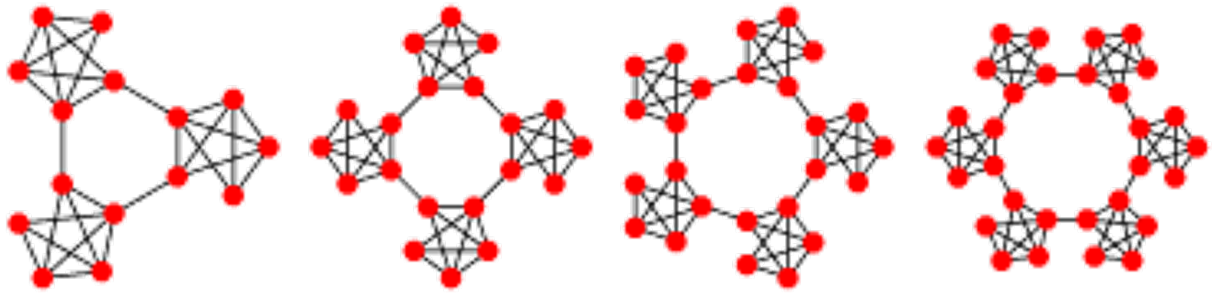


Figure 1.3: Sample Caveman graphs. Image from [5]

together to obtain many different "children" solutions, each equally valid solutions. In general, this process is up to the user of Genetic Algorithms, as it can be application specific. Thus we don't include pseudocode as the actual process of breeding is not what is being explored here, only controlling how and whom can breed.

In our application, 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 when the solution DNA sequence has a different slope sign than the reference DNA sequence's minmax score. Figure 1.4 demonstrates the actual output of both the source species minmax function and the target minmax function. The area between the two lines, labeled "Orig" and "New" respectively, is what is being minimized.

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" [2], 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.

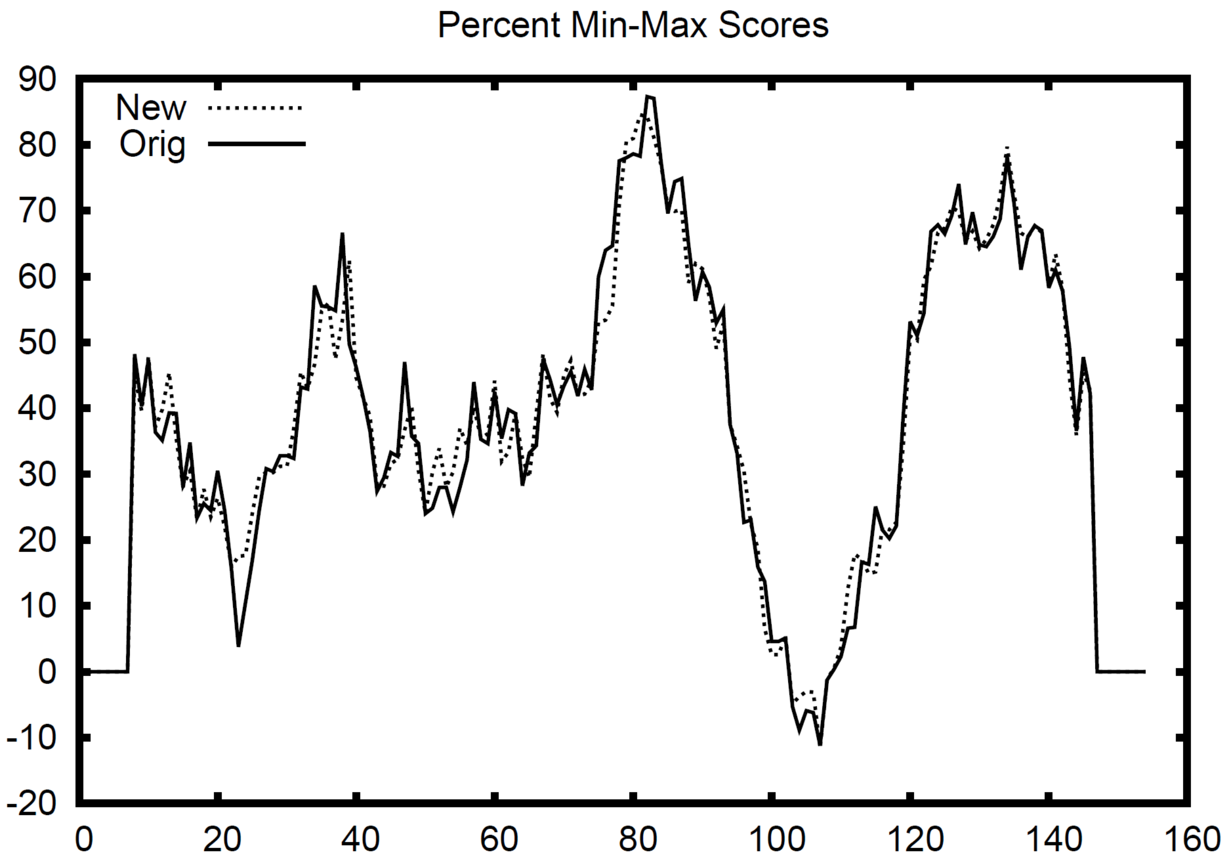


Figure 1.4: Example Resulting bioharmonization graph. Solution generated by Wattz-Strogats with 40 vertices, K=10, P=0.1, with a fitness score of 19429.47

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 with C. elegans, M. musculus, H. sapien, and S. cerevisiae. In our variable node sized graphs, we did a round using 20 vertices, and another with 40 vertices. 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.

For reference, a first generation solution could have a score of 270,000.

As can be seen in the tables 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 vertices, taking 1106s to run. Bumping that up to 40 vertices, we only get a score of 17748, but it takes 4 times as long with 4408s. Using a shorter running example, using only 20 vertices 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 being used 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.

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 vertices. 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

To modify the sequential algorithm, we used a simple process pool to execute a vertex program. The vertex program seen in algorithm 3 takes in the current state of the graph and a vertex. The neighbors of the vertex are then found and bred with the given vertex. The best child out of all breedings is then returned if it is better than the parent. Once all vertices have been processed,

Algorithm 2 Parallel Graph Based Evolution

```
procedure EVOLVE-SINGLE-GENERATION-PARALLEL(G, V, E, K)
    $R = \{\}$
    $NodeTupes = \{\}$
    for $v$ in $V$ do
        $nodeTuple+ = (G, v)$
    end for
    $P = pool(K)$
    $P.map(VertexProgram, NodeTupes)$
    $R = P.results()$
    for $r[1]$ in $R$ and not $r$ in $G$ do
        $replace(G, r[1], r[0])$
    end for
```

Algorithm 3 Vertex Program for Parallel Graph Based Evolution

```
procedure VERTEXPROGRAM(G, v)
    $N = G.neighbors(v)$
    $C = \{\}$
    for $n$ in $N$ do
        $C+ = children(n, v)$
    end for
    $C.sort()$
    if $fitness(C[0]) < fitness(v)$ then
        $return(C[0], v)$
    $returnNULL$
```

then the returned generation is applied to the graph, with parents being replaced by their best child.

1.10 A Reference Parallel Implementation

Our parallel implementation was built off of our sequential one, extending it by using a process pool. Again we targeted Python3 as our language, using PyPy3 to speed up execution. Graphs were generated via NetworkX. Parallelization came from having a python process pool, thus achieving true parallelism in Python.

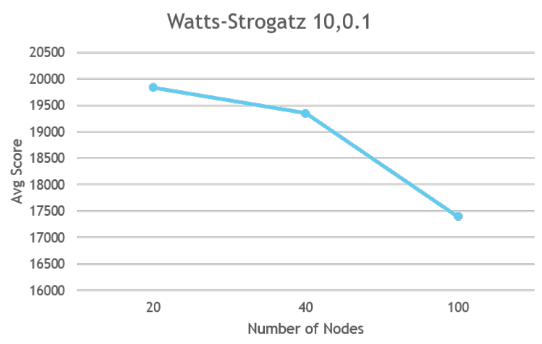
1.11 Parallel Scaling Results

To test, we again used the same computer as in our sequential implementation. The computer is a desktop with an Intel Core i7-5960X@3GHz with 16 logical cores, 32GB of memory running Windows 10. Testing was done using the Windows Subsystem for Linux Ubuntu 14.04.5 LTS. Every test was ran 10 times to obtain the mean and standard deviation. We used the exact same data and harmonization as in the sequential scaling as well.

We introduced a new type of graph to use in our Parallel testing: the Wattz-strogatz small world model [13]. This graph is generated by first putting N vertices in a ring, have each connect to the K closest neighbors, and then have each edge possibly rewire with probability P [4]. The main benefit of this graph is that it has more rational clusters by behaving like the common phrase "six-degrees of separation" where people have their own cluster of people they know, but everyone is only 6 people away from anyone else [13].

Again for reference, a first generation solution could have a score of about 270,000.

Figure 1.5 demonstrates that as we increase the number of processes in our processor pool we do get a speed up, however the gains become smaller and smaller every time we double the number of processes. Going from around 1400s with one process in the pool down to about 300s with 16 processes in the pool we achieve around 4.7x speed up.



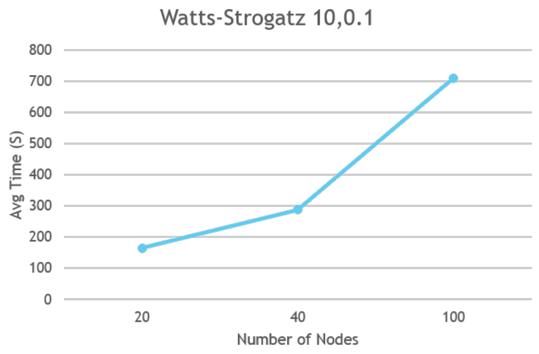


Table 1.1: Here we show the score and time in harmonizing E.coli and Brewer's Yeast vs increasing the number of vertices. K was 10 and P was 0.1.

In figure 1.1 we show the average score and average time of the tests as we increase the number of vertices in the graph. There's a significant increase in time going from 20 vertices up to 100 nods, but there isn't a corresponding decrease in score. The score does improve by about 13%, however the time is almost 3x as long.

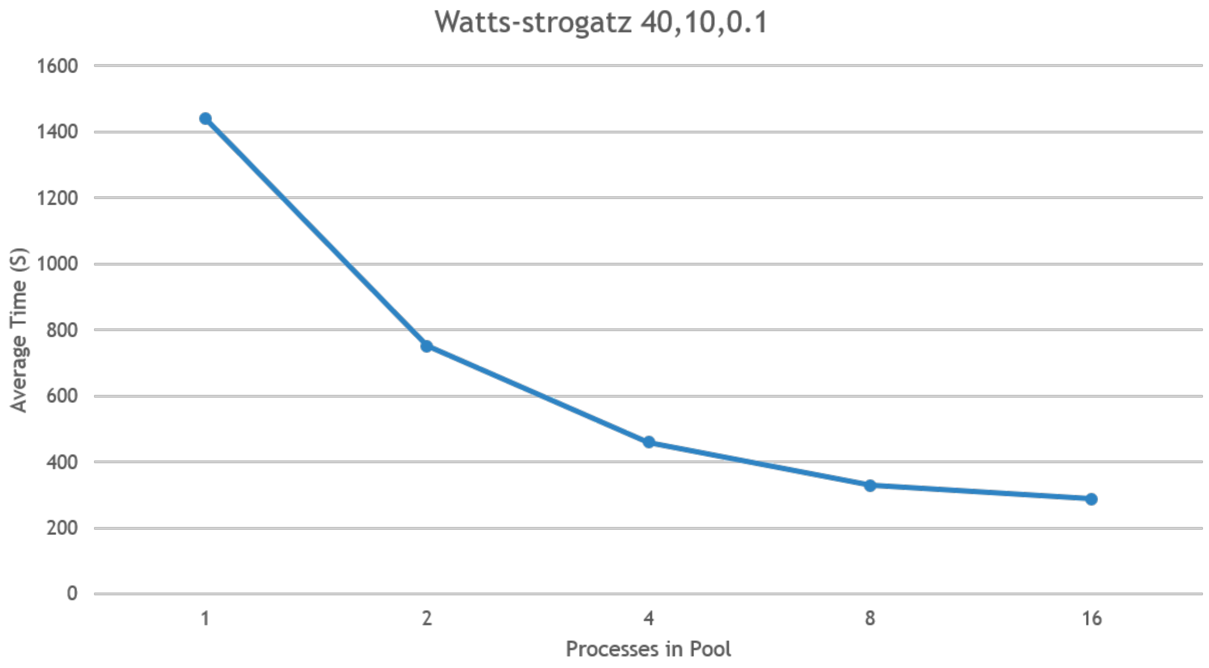


Figure 1.5: In this test, we show the runtime in harmonizing E.coli and Brewer's yeast vs increasing the number of processes in the process pool. The number of vertices was 40, K was 10 and P was 0.1.

Figure 1.2 shows a similar story as figure 1.1 where there is a dramatic increase in the amount of time, but the improvement is only about 22%. Both solutions are roughly within an order of magnitude of each other for an almost 7x cost.

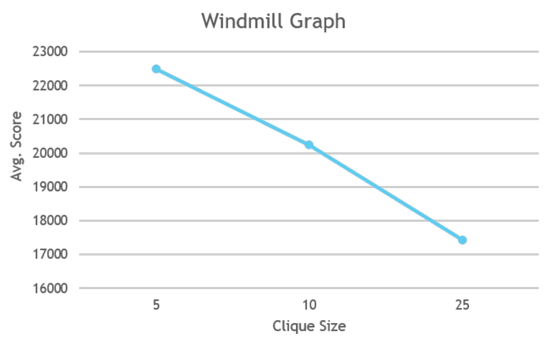
Figure 1.3 shows a similar story as the previous figures as well. Going from 20 vertices to 100 vertices takes 7x as long, with roughly 33% improvement.

Section 1.15 details a comparison of Windmill, Caveman, and Wattz-Strogatz graphs all running at 16 processes and 40 vertices. However they both have different numbers of edges. Windmill and Caveman achieve this by swapping the number of queues and the size of queues. The Wattz-Strogatz graph changes the number of K neighbors being connected to. The data demonstrates that it's not necessarily the number of vertices in the graph which alters the runtime, but rather the number of edges. However the charts do show remarkable similarity in scores between the two different configurations, suggested that the number of vertices does directly impact the score.

1.12 Conclusion

In the end, our work demonstrates that employing graphs can improve the runtime of a genetic algorithm, especially if the correct type of graph is used. The Fully Connected graphs ended up with some of the best scores, but at a rather large cost in terms of time elapsed. By carefully choosing our graphs, we can take a small hit on our score while achieving our roughly equally good result in significantly less time. Similarly, by parallelizing our application, we can make further improvements to our score, taking less time and achieving good scores.

A surprising result came from our parallel implementation when we employed the Wattz-Strogatz graph model. It was able to achieve some of the best scores in the shortest amount



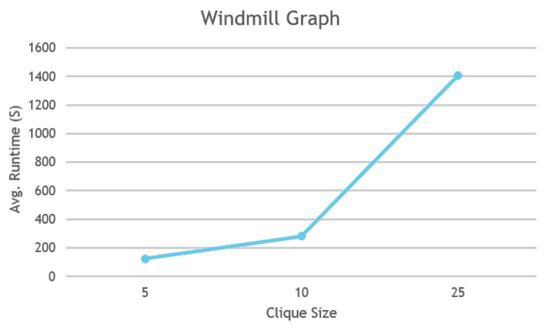
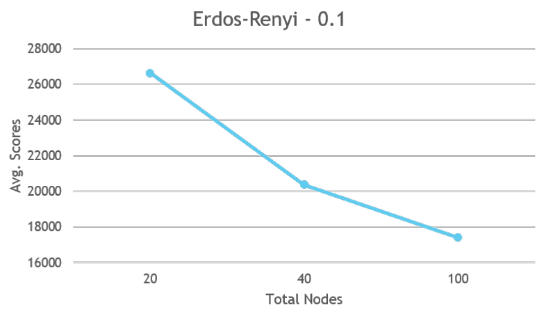


Table 1.2: Here we show the score and time in harmonizing E.coli and Brewer's Yeast vs increasing the number of vertices. Here we had 4 cliques of increasing size, running with 16 processes in the pool.



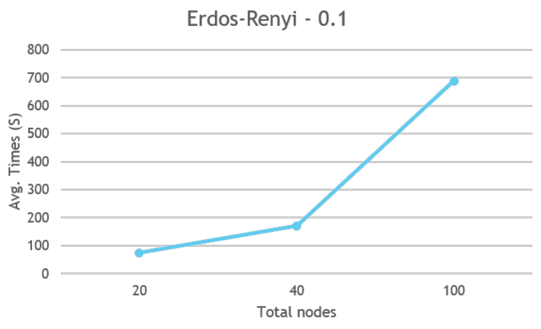


Table 1.3: Here we show the score and time in harmonizing E.coli and Brewer's Yeast vs increasing the number of vertices. Here we had 10% probability of any vertex connecting to any other vertex, running with 16 processes in the pool.

of time compared to the other graph methods. This is possibly due to it being a more natural and organic network model and thus beneficial to genetic algorithms. Examining why this is would be an ideal direction for future work, as well as further research into the current literature on the usage of graphs to help guide genetic algorithms.

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.

Third Iteration:

One reviewer mentioned how I needed to put a citation in section 1.5, so I addressed that. The reviewer also was confused by the DNA terminology at the end of the introduction, thus I made a little clarification to distinguish between the DNA in our problem, and DNA as a data structure for genetic algorithms. They also wanted some justification for the graphs chosen, thus I wrote a paragraph in the basic datasets section explaining why. They also wished for some expansion into different mutation rates, but measuring that was a little beyond the scope of this project. They additionally had a few minor changes in grammar, and thus addressed. They also pointed out a potential bug in Algorithm 1 of replacing a parent multiple times. This was fixed so only the child with the smallest score was looked at by a parent and possibly replaced. They also pointed out in appendixA there were some uneccesary leading 0s and in one instance a stdev with only one item after the decimal point. These were fixed. I also added an example output to demonstrate what is actually being generated and minimized.

Another reviewer had a few comments that I felt were not fully actionable. For example, they said that it wasn't clear what the metrics being evaluated are, yet the kernel section specifically states we're looking for low runtime and better scores. They also stated that I should include the multiple generations in the pseudocode. I feel as if the algorithm itself is about how to use a single generation with a graph. Most genetic algorithms and other evolutionary algorithms use multiple generations, thus including that feels redundant. Two reviewers wanted more clarification on what vertices and edges meant, so I addressed this in the as-a-graph section.

The final reviewer had some similar complaints, as well as a few grammatical errors pointed out.

Several reviewers wanted to see some visualizations of the graphs, so I added three images for graphs which are harder to imagine but make better visualization.

1.14 Sequential Scaling Results - Results

Fully Connected - 20 vertices

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	1106.064027	84.105426	19126.919	1283.620401
H. sapien	1025.578735	2.427339	24928.944	1606.458714
M. musculus	1020.771278	2.082315	21137.479	2176.747175
C. elegans	1041.69969	0.767798	23789.236	1482.189131

2D Grid - 20 vertices

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	30.571147	22628.39	1405.021405
M. musculus	4203.88681	108.770135	19826.178	2073.063743
C. elegans	4295.030674	46.220033	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	0.282972	23104.234	1864.526685
C. elegans	373.068643	0.593986	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	0.791199	25169.657	849.044711
M. musculus	0984.32995	1.111364	21371.185	1739.21804
C. elegans	1010.327206	0.632495	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

1.15 Parallel Scaling Results - Results

Windmill Graph - 4,25 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	1408.068770	8.699657	17428.424	1111.514012
H. sapien	1423.719129	10.158205	21811.427	1624.714577
M. musculus	1439.245480	26.072518	18648.071	841.697471
C. elegans	1404.060382	2.853025	20139.532	1090.583855

Windmill Graph - 25,4 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	469.180785	9.633604	20847.582	1406.669398
H. sapien	495.512529	3.884894	25048.655	1934.238458
M. musculus	466.830144	13.005787	23982.811	1560.534256
C. elegans	456.034971	2.652946	26518.976	2373.275543

Caveman Graph - 4,25 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	1397.356041	3.315249	17379.616	559.891502
H. sapien	1417.976462	3.782410	21811.167	1155.276183
M. musculus	1421.361858	3.749360	19056.142	905.003859
C. elegans	1406.844888	12.047170	21306.424	1237.06641

Caveman Graph - 25,4 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	365.723654	6.032161	24305.905	1874.371394
H. sapien	363.854922	0.903344	29282.458	1242.619206
M. musculus	366.806325	2.963881	25348.137	1583.727080
C. elegans	375.012293	3.576178	27629.355	957.840066

Wattz-Strogatz - 40,20,0.1 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	471.01908	1.914677	17617.438	1750.497095
H. sapien	477.796109	5.220690	23033.722	1135.957858
M. musculus	473.214193	2.311429	19182.586	1310.839698
C. elegans	470.439868	6.602775	21481.47	1274.895345

Wattz-Strogatz - 40,10,0.1 - 16Proc

Genes	*Time (s)*		*Score*	
	mean	stdev	mean	stdev
S. cerevisiae	287.126895	4.592964	19348.722	1898.003673
H. sapien	288.292479	2.427174	23230.660	880.696387
M. musculus	288.253836	2.194757	20261.781	1514.381834
C. elegans	284.449048	1.707648	22243.756	1684.05801

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] D. Ashlock, M. Smucker, and J. Walker. Graph based genetic algorithms. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1362–1368 Vol. 2, July 1999.

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

[4] Network X. watts_strogatz_graph, 2018.

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

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

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

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

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

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

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

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

[13] Wikipedia contributors. Wattsstrogatz model — Wikipedia, the free encyclopedia, 2018. [Online; accessed 12-December-2018].

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