Fall 2020 Graph Theory Project Assignment

Colin Greeley, 11567185

Math 453

Introduction

The topic that I will be exploring for this project is the traveling salesperson problem (TSP) [1]. This problem is defined as follows: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city [1]?" In essence, this problem is fairly straightforward and easily understandable but even with 90 years of the brightest minds working toward a generalized algorithm to solve it, there is currently no solution.

The TSP is an NP-hard [2] graph problem in theoretical computer science that was first formulated in the 1930s. NP-hard problems refer to all problems that are solved in nondeterministic polynomial time and are at least as difficult to solve as an NP-problem [2]. The reasoning for the TSP being NP-hard is that time-complexity for linear search increases factorially relative to the number of vertices in the graph. Factorial growth is significantly larger than exponential growth, which makes the TSP seemingly impossible to efficiently find an absolute solution given a large number of vertices. The time-complexity for the linear search of a shortest path Hamiltonian cycle is $O(n!)$, since the graph generated by this problem is an undirected weighted complete graph [4, 5]. A complete graph on n vertices is denoted by K_n where K_n has $n(n-1)/2$ edges and is a regular graph of degree $(n - 1)$ [4].

Technical Background

There are numerous algorithms that have been created in efforts to solve this problem that have proven to run in less than factorial time. The algorithm that interests me the most is a genetic algorithm [3]. A standard genetic algorithm is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover, and selection [3]. What makes this algorithm interesting is that it does not have a defined time-complexity because the law of natural selection makes improvements from mutations which are random. For this reason, this algorithm is good for approximating solutions for the problem, not finding an absolute solution. A solution generated by genetic algorithm is called a chromosome, while collection of chromosomes is referred as a population. A chromosome is composed from genes and its value can be either numerical, binary, symbols, or characters depending on the problem want to be solved [6]. In the case of this problem, the chromosomes will be composed of the vertex numbers where the edges of the graph are defined be adjacent vertices in the chromosome (Figure 1).

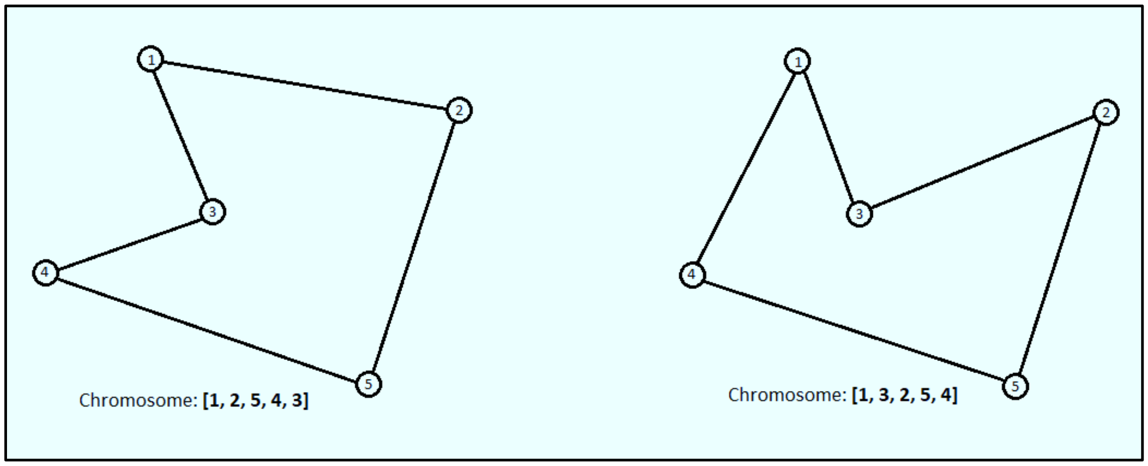


Figure 1 Chromosome illustration for a genetic algorithm on the TSP

There exists an edge for every pair of vertices with a corresponding weight on the complete graph, but the graph generated by each chromosome is Hamiltonian. Additionally, for the implementation of this algorithm, instead of displaying the graph generated on n vertices as a complete graph with weighted edges, the weights will be defined by Euclidean space since the vertices will be randomly generated on a 2-dimentional gird. This will make the visualization for this problem easier to follow.

Development of Results

I will be using the Processing programming language to implement this algorithm because it is very useful for creating visual representations for any type of program. Additionally, the vertices are defined by an xy coordinate and the edges are defined two vertices where the weight of each edge is the Euclidean distance between its respective vertex endpoints, so drawing the graph to the screen should be straightforward.

The algorithm to approximate the shortest path Hamiltonian cycle on the complete graph is defined in the steps below [6]:

(1) Determine the number of chromosomes, generation, and mutation rate and crossover rate value
(2) Generate chromosome-chromosome number of the population, and the initialization value of the gene's chromosome-chromosome with a random value
(3) Process steps 4-7 until the number of generations is met
(4) Evaluation of fitness value of chromosomes by calculating objective function
(5) Chromosomes selection
(6) Crossover
(7) Mutation
(8) Solution (Best Chromosomes or no further improvements)

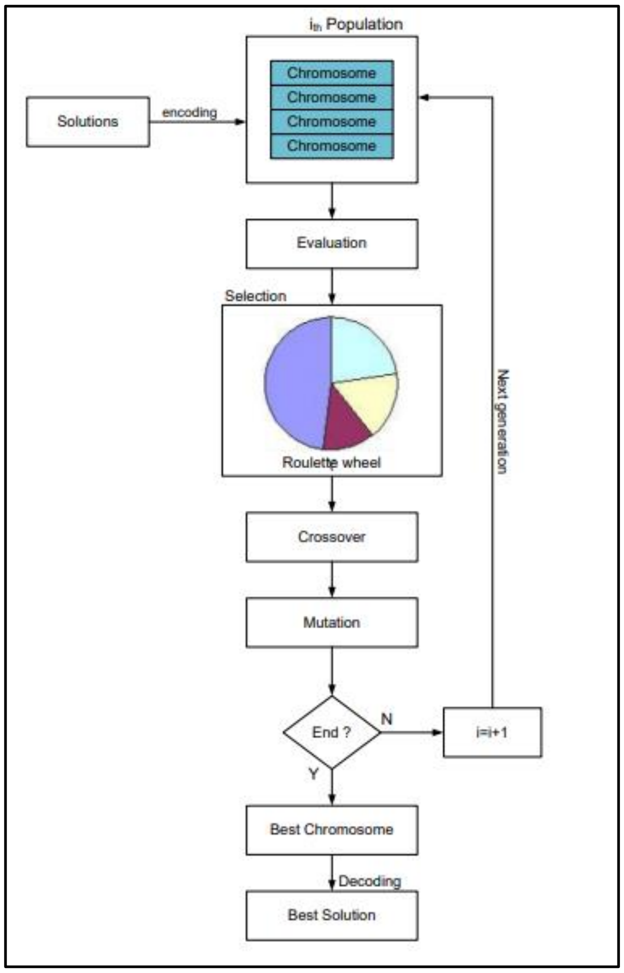


Figure 2 Algorithm flowchart

The first step of this algorithm is to define the hyperparameters for the problem. The number of chromosomes, also known as the population size, is the most influential hyperparameter on the algorithm's performance. The intuition behind this is that the larger a population is, the more likely a "good" mutation will occur for any given generation, this will also lead to further benefits during crossover. Furthermore, the best population size for this algorithm is the maximum array size that will fit in the computer's memory. Since the chromosome is defined as a list of vertices, we can generate a sparse variety of chromosomes by first creating a list with all vertices in order, copying it p times where p is the population size, and finally shuffling each list individually.

Let us start with an example where $n = 10$ so chromosome $c_1 = [1,2,3,4,5,6,7,8,9,10]$. Next, we generate n vertices on the 2-dimentional coordinate plane, in the Processing program, I have defined the sketchpad to the size 1200x800 pixels. The graph G is shown below in Figure 3, which has 10 vertices and $\frac{n(n-1)}{2} \to 45$ edges. Each vertex of the graph was assigned with a random location from a uniform distribution and an edge for every pair of vertices exists. The white path connecting each individual vertex is the Hamiltonian cycle generated from the chromosome $c_1 \to [1,2,3,4,5,6,7,8,9,10]$. From observation, it is clear that c_1 is not the shortest Hamiltonian cycle on the graph G, but the solution is some ordering of the indices of c_1.

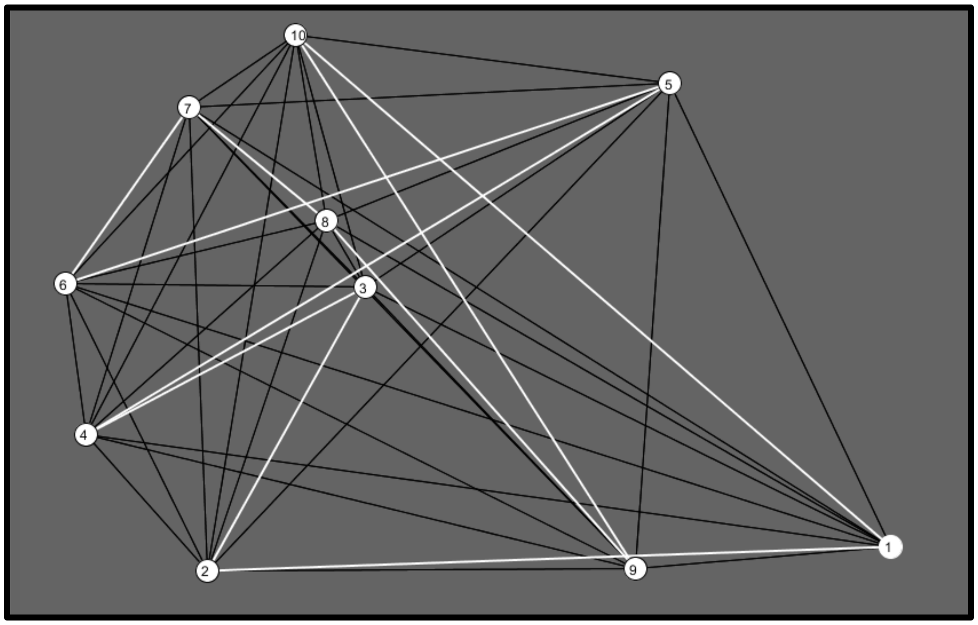


Figure 3 Randomly generated graph where $n = 10$ and c_1 is displayed

Given that $n = 10$ in this example, the are $10! \rightarrow 3.6$ million different combinations of vertex orderings meaning that there are also 3.6 million unique chromosomes, 10 of which will result in the shortest Hamiltonian cycle. There are 10 chromosomes which will result in the correct solution because the starting vertex of the cycle is irrelevant, only the ordering is what matters. With this in mind, if our population size is large enough (i.e., greater than $(n-1)!$) it is likely that we will generate the solution simply from shuffling the chromosomes in the first generation. Furthermore, it is n times more likely to generate the solution from shuffling the chromosomes of a population size of $n!$. This quickly becomes impractical given that a graph with only 15 vertices would result in 1.3 quadrillion unique Hamiltonian cycles on the graph. This is where steps 3-7 of the genetic algorithm take place.

Let us start from the first generation where all chromosomes in the population have just been shuffled. We need to select the most fit genes chromosomes from the entire population, this process is called selection. The next step is to calculate the fitness of each individual chromosome so that only the most fit genes are moving on to the next generation. This is done by calculating the total distance covered by a walk on the cycle generated from each individual chromosome, this value will be denoted by the variable d. Again, the weight of each edge is the Euclidean distance that the edge covers from its one endpoint to the other defined by a pair of vertices on the graph. Once we have d calculated for the entire population, we create a selection list as follows:

$$S_i = \left(\frac{1}{1+d_i}\right) * \left(\frac{1}{\sum d}\right) \text{ for } i \text{ in population size}$$

S is the selection list where each index represents the probability of selecting a chromosome in the population. The first part of the equation above is the fitness calculation for each chromosome. The total distance covered by the walk is on the denominator implying that a smaller distance leads to a higher fitness. Each value in S is then divided by the sum of distances making $\sum_i s_i = 1$, essentially converting the entries of s to probabilities. The next step is to compute the cumulative probabilities denoted by the list C.

$$C_i = \sum_j^i S_j \text{ for } i \text{ in population size}$$

To sample the next population from the list of cumulative probabilities, we generate p random number between 0 and 1 then select the chromosome from C who's probability is closest to the random number. This type of semi-random sampling is called roulette wheel sampling, where chromosomes with a higher fitness are more likely to be randomly selected from the list of cumulative probabilities. This also means that there will be duplicates of the most fit chromosomes since it they are more likely to be randomly selected.

The next step of the algorithm is crossover. For this, we randomly select certain percent of the chromosomes based on the crossover rate and put them in a pool to crossover with each other. The pseudo code below describes the crossover process between two chromosomes:

```
$c_x$ and $c_y$ $a$re two chromosomes selected for crossover
$i =$ random integer between 0 and $n$
$j =$ random integer between $i$ and $n$
$c_{x_k} = c_{y_k} where\ i \le k \le j$
$c_{y_k} = c_{x_k} where\ 0 \le k \le i\ and\ j \le k \le n$
```

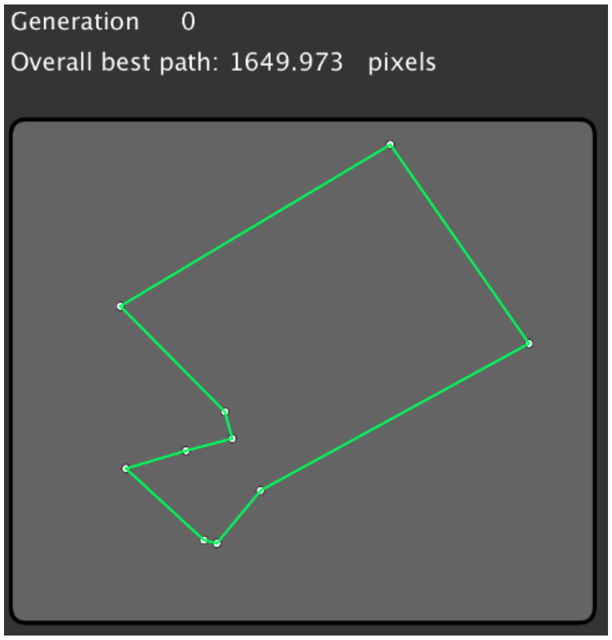
The crossover process promotes "genetic diversity" which means that chromosomes have a chance to replace their own bad genes (long weighted path) with good genes (short weighted path) from another chromosome. Over a large number of generations, this process has proven to be effective in removing bad genes from an entire population.

Now that the algorithm is set up to select the best chromosomes while simultaneously removing bad genes (i.e., optimizing for the shortest path by combining the best parts of all the chromosomes in the initial population) the next step is to implement mutation so that we can explore new, potentially shorter paths. The implementation of mutation for the TSP is quite simple. For each chromosome in the population, there is a change that any given chromosome will be selected for mutation given by the mutation rate. The pseudo code below shows the mutation process for one chromosome:

```
$i =$ random integer between 0 and $n$
$j =$ random integer between 0 and $n$
$c_{x_i} \Leftarrow c_{x_j}$ , $c_{x_j} \Leftarrow c_{x_i}$
```

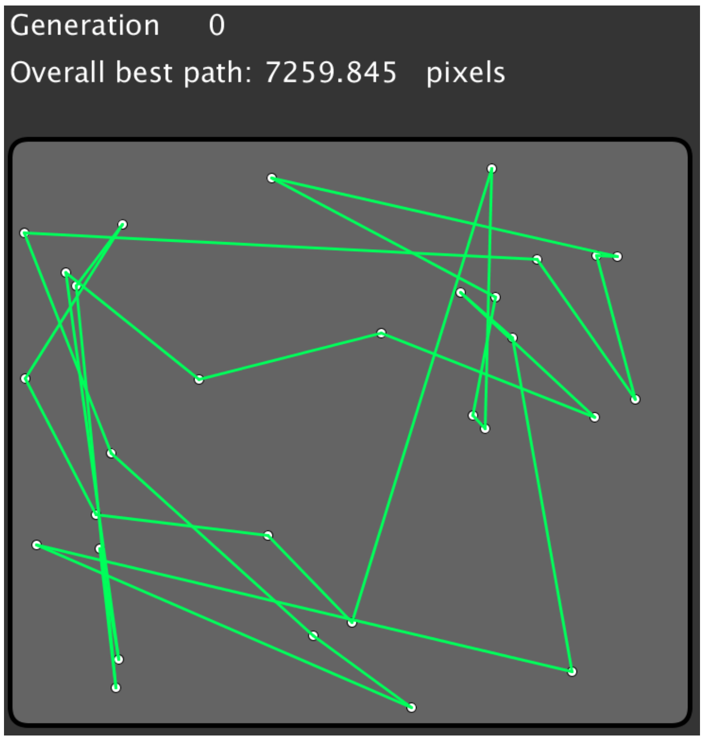
As seen in the pseudo code above, all that is happening during mutation is two random indices of chromosome c_x are being swapped. Even though the process is simple, the result is quite significant since adjacent vertices in the list are connected by an edge, swapping two vertices can result in four edge changes. After mutation occurs, we repeat steps 4-7 until we have found a solution. There is no way to identify if the shortest path has been found because if we knew what the shortest path was, there would be no point in this algorithm. This is also why the result of this algorithm is an approximation and not an absolute solution. Nonetheless, the algorithm stops when no further improvements are made after a certain number of generations.

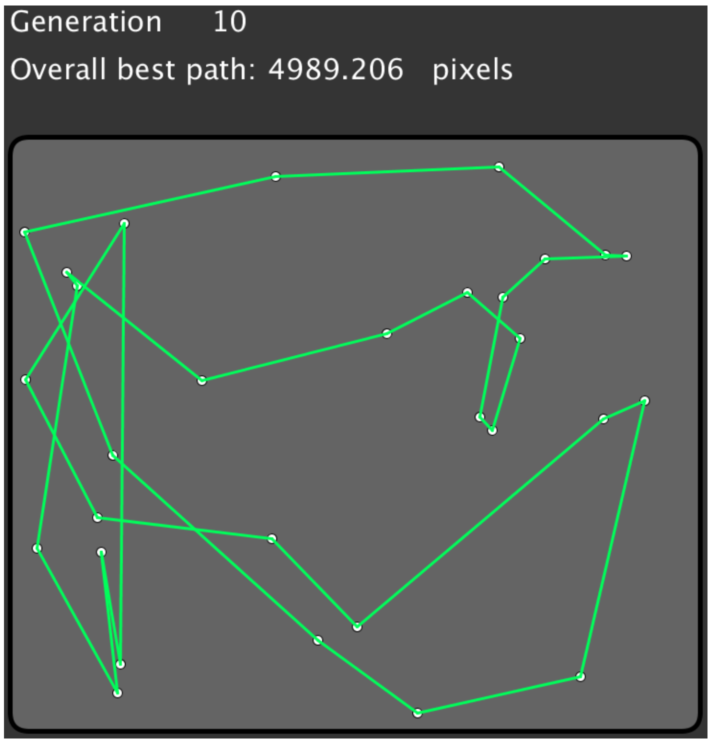
The following test results were generated from a vertex count of 10, 30, and 50, respectively. A population size of 100000, crossover rate of 0.3 and a mutation rate of 0.2.

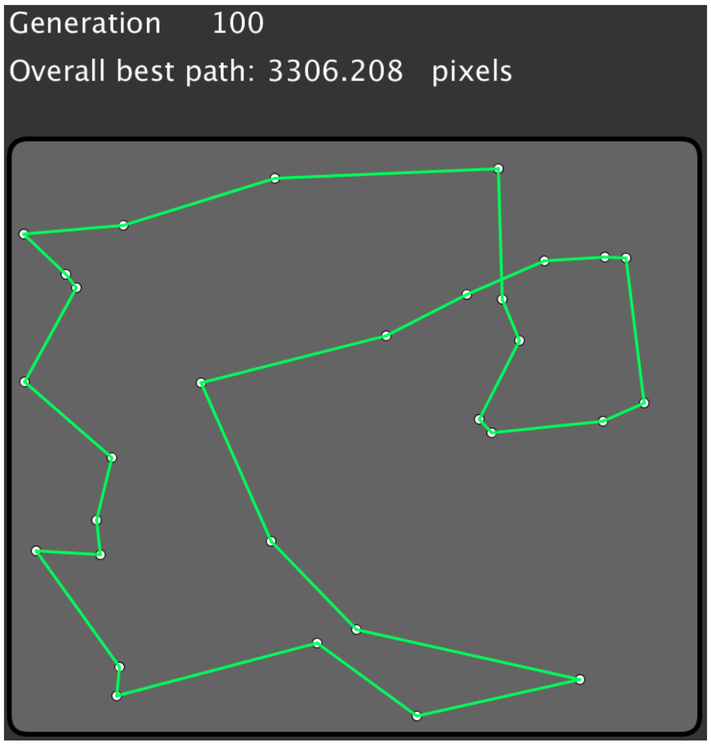

Generation 0
Overall best path: 1649.973 pixels

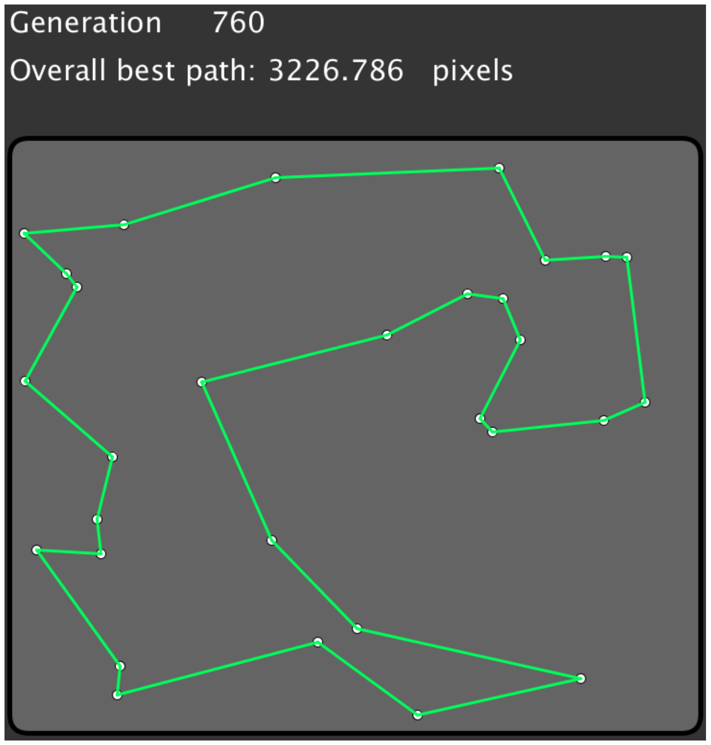
10 vertices
Solution on 1st generation

30 vertices, solution on generation 760

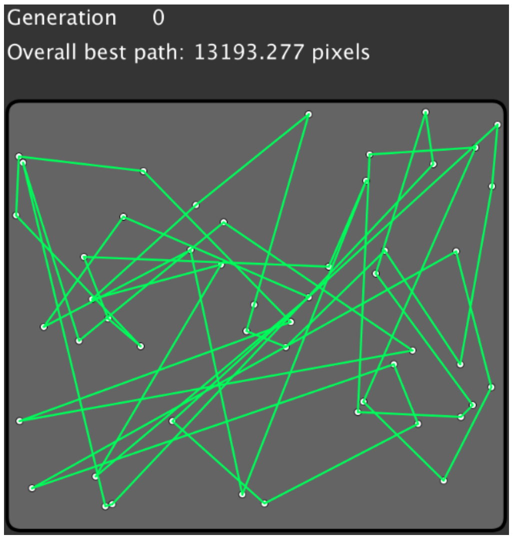

Generation 0
Overall best path: 7259.845 pixels

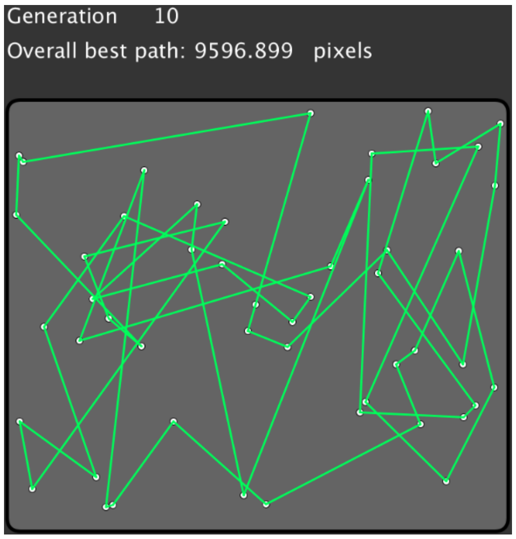

Generation 10
Overall best path: 4989.206 pixels

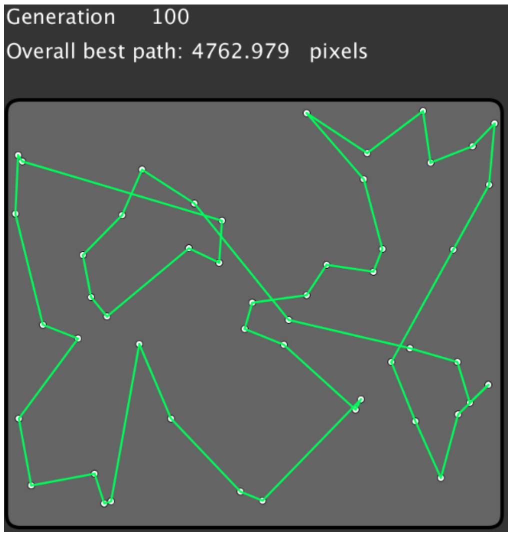

Generation 100
Overall best path: 3306.208 pixels

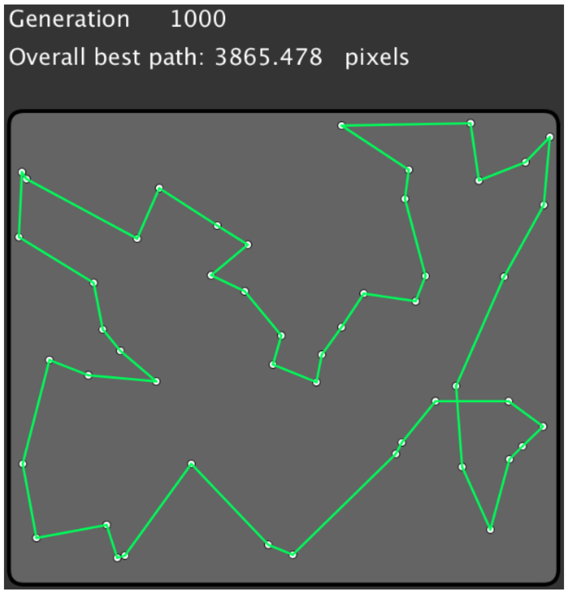

Generation 760
Overall best path: 3226.786 pixels

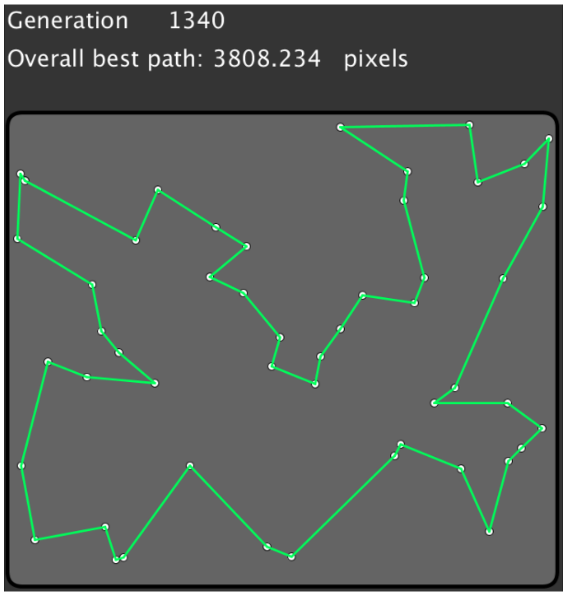
50 vertices, solution found on generation 1340


Generation 0
Overall best path: 13193.277 pixels


Generation 10
Overall best path: 9596.899 pixels


Generation 100
Overall best path: 4762.979 pixels


Generation 1000
Overall best path: 3865.478 pixels


Generation 1340
Overall best path: 3808.234 pixels

As seen from the tests above, this algorithm does a very good job at minimizing the sum of the weighted path during the early generations but seems to have trouble near the end when there is something like a small kink in the path. This simplified genetic algorithm is very ineffective for finding a solution on graphs with more the 50 vertices but still does a good job at minimizing the path in the early states

Observations and Implications

The results from this project solidify that fact that natural selection is great for finding a solution to a problem that is good enough, but not necessarily perfect. This is just one small example on how biology and nature are inspiring and working their way into algorithms. Evolutionary algorithms fit into the category of machine learning, which is a set of algorithms inspired by biological phenomenon's that are very effective for finding patterns and solutions for a wide variety of problems. Artificial neural networks are another example of a graph algorithm inspired by biological neural networks (i.e., a human brain) that have worked their way into almost every aspect of out daily lives. Regarding the TSP, there are many other algorithms that do work for a very large number of vertices that are based on branching. In summary, they basically branch out from each individual vertex and make the path by finding the shortest branches that overlap. I think this approach will be the bases for future algorithms that find solutions for high vertex counts.

Future Directions

The genetic algorithm used in this paper to approximate a solution for the TSP is probably the most basic and naïve implementation of an evolutionary algorithm. There are some extra functions used in other evolutionary algorithms that may have boosted the performance in this case and helped solve the problem much faster. Current state-of-the-art evolutionary algorithms have the ability to accurately simulate behaviors for a group of creatures. The area in which evolutionary algorithms are the most effective is general machine learning and reinforcement learning. Reinforcement learning algorithms require an agent to interact with an environment in order to achieve some sort of goal by incrementally receiving rewards based on the agent's performance. Evolutionary algorithms are great for this because the rules of natural selection that govern them encourage an agent to explore and then remember actions that lead to high reward. This makes sense because simulations that are this advanced are usually mimicking some sort of biological phenomenon. For the case of graph algorithms, this is probably the extent to which evolutionary algorithms will be implemented.

References

[1] "Travelling salesman problem," *Wikipedia*, 01-Dec-2020. [Online]. Available: https://en.wikipedia.org/wiki/Travelling_salesman_problem. [Accessed: 5-Dec-2020].

[2] “NP-Hard Problem,” *from Wolfram MathWorld*. [Online]. Available: https://mathworld.wolfram.com/NP-HardProblem.html. [Accessed: 5-Dec-2020].

[3] A. Gad, “Introduction to Optimization with Genetic Algorithm,” *Medium*, 03-Jul-2018. [Online]. Available: https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b. [Accessed: 6-Dec-2020].

[4] “Complete graph,” *Wikipedia*, 01-Dec-2020. [Online]. Available: https://en.wikipedia.org/wiki/Complete_graph. [Accessed: 6-Dec-2020].

[5] “Graph (discrete mathematics),” *Wikipedia*, 5-Dec-2020. [Online]. Available: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics). [Accessed: 6-Dec-2020].

[6] Denny Hermawanto “Genetic Algorithm for Solving Simple Mathematical Equality Problem,” *Arxiv*. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1308/1308.4675.pdf . [Accessed: 6-Dec-2020].