**CECS 545 Project 2 report**

**Ryan Schron**

**October 7, 2019**

**Introduction**

In Project 3 we implemented a greedy approach to solving the Traveling Salesman Problem (TSP). That is, finding the least cost Hamiltonian cycle in a fully connected graph. Due to the complexity of his NP-Hard problem, solutions must generally be optimized for different performance metrics. One such example is to optimize for run-time at the expense of guaranteeing the most accurate result.

This project will implement a genetic algorithm to solve the TSP problem. Such an approach does not guarantee that the most accurate solution is found, but it does allow us to solve much larger graphs than previous approaches in a reasonable amount of time. A greedy approach is deterministic and prone to misreporting the optimal solution by getting suck at local maxima. Genetic Algorithms, on the other hand, are stochastic. This stochasticity allows for the algorithm to not get stuck at a local maximum within the search space, providing a better chance at finding the optimal solution.

Genetic algorithms work by maximizing a fitness score that represents how well it has solved the problem thus far. It does this by first initializing a population of potential solutions that are then “bred” to create new potential solutions. This breeding can happen in many ways, but one typical implementing is to use crossover, combining part of parent one with part of parent two. The progeny is then made to compete based on their fitness for the opportunity to move to the next round of breeding. Once they are selected, random mutations are introduced into the population to ensure that potential best solutions are found. This process is repeated for a fixed number of generations.

This experiment will test different hyperparameters that affects this process, including mutation rates, population size, and ranking methods.

**Running the Program**

This program was developed using IPython due to its ability to display visualizations inline with the code. It is recommended that this program be run in Jupyter Notebooks. Once loaded, you may navigate to cell>run all to run the notebook.

**Code Description**

*Initialization*

The program is initialized by first reading in the coordinates within the Random100.tsp file. Each coordinate is created as a Node class that stores the name of the node, the x position, and the y position. Additionally. this Node class implements a distance() method that compares itself to any other node and returns the Euclidean distance.

Next the collection of nodes is used to initialize a population of individuals. Each individual is an object that contains an ordered collection of nodes, representing a path. It also implements a path\_length() method that traces the path calling each node’s distance method to generate and store the path length. This path length is then inversed (that is, 1/path length) to create a fitness score that may be maximized. Individual also implements a get\_path() method that traces the Node list and returning a python list with the path represented as strings. Each individual’s path contains a copy of each node exactly one time, but in a random permutation. This collection of individuals is implemented by the initialize\_pop() method.

Finally, the find\_fitnesses() method is called to finish initialization. This method browses the collection of individuals within the population and calls their path\_length() method to compute all of the distances and fitness scores. At this point the population is ready for ranking.

*Rank Population*

This project implements two ranking methods: elite ranking and proportional ranking. The first sorts all members of the population and returns the top candidates based on fitness. The rank\_population\_elite() function handles this.

Proportional ranking is achieved by converting each member’s fitness score into a probability to be selected. A random number between 0 and 1 is generated, and then the algorithm moves through the collection of individuals subtracting their probability until a number less than or equal to zero is found. The individual at this point is then removed from the list and selected for breeding, and its selection probability is distributed evenly to the remaining individuals.

*Breed Parents*

After ranking, the determined most fit individuals are bred to create new individuals. This is achieved by first adding the parents to the population pool for the next generation and then selecting two random parents for breeding. A random segment of parent 1’s path is used to form part of a gene. Parent 2’s path is then appended to the end of the selected path from parent 1, omitting any repeats that came in the gene from parent 1. The new individual is then initialized and added to the population pool. This method of one-point crossover was found via Tutorialspoint.com [1].

*Mutate Candidates*

Finally, random mutations are introduced into the new population at a predetermined rate. A swap mutation is implemented, which if triggered, swaps the location in the path of a random pair of nodes. This mutation is implemented at the mutation rate across the population, and new fitness scores are then initialized. At this point, the performance of the population is recorded, and a new round of breeding begins. The process repeats for a number of iterations determined by the user, which in this case, is set to 200 generations. The method of mutation was also found at Tutorialspoint.com [2].

**Code Performance**

Multiple tests of different hyperparameters were performed and are described below. Each test was running 10 times to generated basic summary statistics around the solutions.

*Test1 - Baseline*

Number of Generations: 200

Mutation Rate: 1%

Population Size: 200

Breeding Pool: 10

Ranking Method: Elite Ranking

Distance Statistics

Range: 1301.6723874637084, 1770.6742658571127

Mean: 1545.555654382615

Variance: 17361.718862545917

Time Statistics

Range: 26.36361074447632, 26.761484622955322

Mean: 26.588439631462098

Variance: 0.01901603047579416

Best Path:

[16, 99, 70, 93, 17, 74, 94, 10, 90, 20, 98, 60, 6, 81, 68, 79, 24, 71, 1, 43, 95, 30, 72, 73, 83, 15, 92, 36, 39, 21, 89, 85, 7, 3, 23, 53, 86, 41, 66, 75, 38, 37, 91, 19, 61, 63, 28, 32, 100, 55, 46, 34, 96, 52, 45, 22, 27, 88, 65, 2, 50, 14, 35, 97, 54, 8, 51, 29, 56, 13, 25, 40, 33, 64, 59, 4, 84, 31, 62, 26, 87, 77, 42, 78, 57, 9, 67, 44, 5, 80, 82, 58, 76, 48, 49, 18, 12, 47, 69, 11]

*Test 2 – Lower Population*

Number of Generations: 200

Mutation Rate: 1%

Population Size: 50

Breeding Pool: 5

Ranking Method: Elite Ranking

Distance Statistics

Range: 1514.9525890296225, 2255.7658859374515

Mean: 1904.986468378393

Variance: 54730.98793407257

Time Statistics

Range: 6.433952569961548, 6.558912515640259

Mean: 6.490034890174866

Variance: 0.001621283330224009

Best Path:

[8, 54, 51, 29, 13, 98, 60, 25, 6, 20, 81, 95, 92, 1, 71, 73, 24, 7, 75, 89, 41, 37, 15, 79, 68, 40, 84, 56, 64, 59, 31, 33, 35, 87, 2, 4, 97, 12, 49, 76, 22, 27, 82, 57, 58, 42, 78, 14, 80, 62, 44, 5, 26, 77, 88, 48, 50, 9, 67, 96, 28, 61, 55, 65, 45, 52, 86, 19, 53, 23, 46, 91, 3, 34, 32, 100, 63, 39, 21, 38, 66, 36, 72, 85, 43, 30, 83, 94, 74, 17, 93, 16, 69, 18, 47, 11, 99, 70, 90, 10]

*Test 3 – Proportional Ranking*

Number of Generations: 200

Mutation Rate: 1%

Population Size: 200

Breeding Pool: 10

Ranking Method: Proportional Ranking

Distance Statistics

Range: 4028.640398692472, 4411.945643141621

Mean: 4211.554217009722

Variance: 9113.43749486711

Time Statistics

Range: 48.48357176780701, 99.2311520576477

Mean: 77.44366040229798

Variance: 593.6356662053954

Best Path:

[69, 55, 88, 28, 46, 2, 68, 74, 76, 29, 67, 22, 3, 26, 81, 41, 7, 83, 94, 37, 100, 34, 21, 64, 82, 31, 30, 72, 66, 77, 42, 62, 12, 24, 38, 23, 40, 13, 6, 33, 60, 19, 4, 45, 61, 96, 78, 70, 75, 56, 20, 59, 73, 52, 63, 86, 39, 93, 95, 92, 89, 85, 36, 87, 84, 49, 53, 1, 98, 99, 79, 51, 15, 8, 90, 47, 48, 43, 44, 65, 58, 9, 5, 10, 71, 54, 14, 50, 91, 32, 35, 27, 97, 25, 11, 16, 17, 18, 80, 57]

*Test 4 – Elevated Mutation Rate*

Number of Generations: 200

Mutation Rate: 5%

Population Size: 200

Breeding Pool: 10

Ranking Method: Elite Ranking

Distance Statistics

Range: 1340.091991696275, 1917.2362644675045

Mean: 1601.3489589813594

Variance: 24118.75203129615

Time Statistics

Range: 37.912431716918945, 54.13096761703491

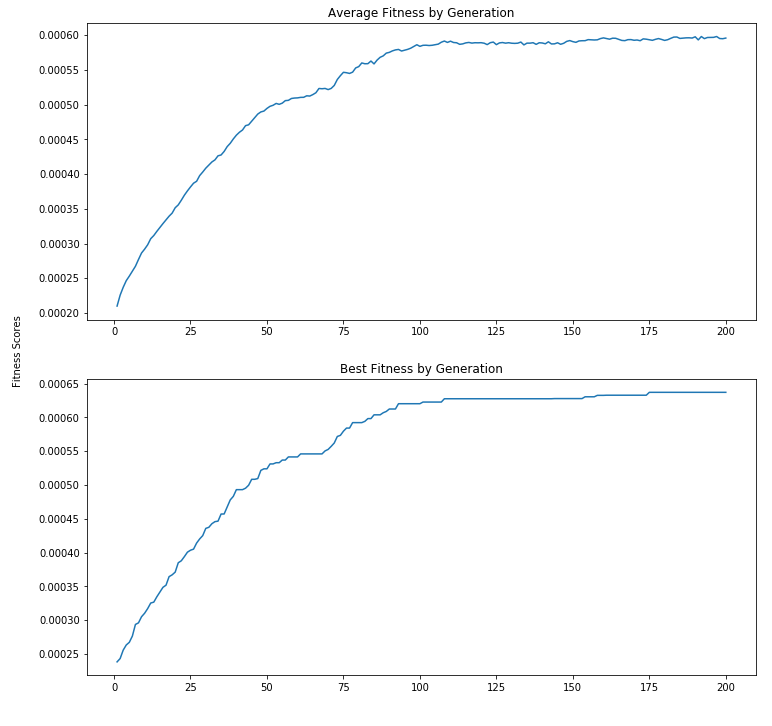
Mean: 50.66906380653381

Variance: 26.014730709286116

Best Path:

[70, 93, 94, 10, 90, 54, 8, 20, 98, 40, 29, 64, 59, 84, 31, 56, 51, 33, 13, 25, 81, 60, 6, 68, 95, 79, 15, 92, 1, 71, 24, 73, 72, 83, 85, 43, 30, 38, 37, 17, 39, 41, 36, 7, 89, 74, 75, 66, 21, 3, 53, 19, 23, 34, 86, 49, 87, 62, 77, 26, 35, 4, 58, 80, 42, 44, 5, 9, 27, 22, 96, 52, 65, 78, 57, 82, 88, 50, 67, 45, 100, 91, 32, 28, 63, 55, 61, 46, 14, 48, 2, 76, 12, 18, 97, 47, 11, 69, 16, 99]

The notebook shows a graph of algorithm performance for each iteration, in addition to the summary statics. As it is 80 graphs, they are left out of this report, but an example one is below:



**Conclusions**

The control set of hyperparameters was chosen based on light experimentation to find a well performing set of conditions and was the best performing set of all options. Test two tried reducing the breeding pool for each generation, and as a result did not perform as well as the control set. It is probably due to a lack of sufficient diversity, and as a result an ideal solution could not be found. Test 3 tried to change the ranking method to a proportional selection, but performed no better than random guesses, which could indicate a fault in the implementation. The final test increased the mutation rate and performed nearly as well as the control conditions with regards to identifying the shortest path, when looking at the minimum found. Unsurprisingly, it also resulted in an increase in the variance among found path lengths, which is again unsurprising as more randomness was introduced in this test.

As far as run time is concerned, the tests also performed fairly predictably. Reducing the breeding pool also decreased the run time to around 6.5 seconds, which is consistent with generating less candidates. The proportional ranking test was the longest at an average of 77 seconds, which could also suggest an implementation problem. Finally, the increased mutation rate was equivalent to the control set at around 26 seconds. This result is reasonable as the increase of mutation rate was only to 5% and the swap method isn’t expected to take much additional computation time.

In reflection a genetic algorithm approach is interesting due to the way in which it blends competitive selection with stochastic behaviors. Its downside, however, is that some cases could involve significant experimentation in tuning the hyperparameters of the method. As far as this project goes, the biggest challenges were finding a workable scheme to implement the solution (that is, how to manage the data structures), and implementing the proportional ranking selection. Looking at the results, I believe it is possible that the latter did not work as intended. I would examine that in the next iteration, and also drop use of the inverse fitness function. I chose to invert the path length because I felt it would be more natural to look at an increasing function, but in reality, I believe it only served to obscure the values.

**Reference**

[1] *Genetic Algorithms – Crossover.* Retrieved from *www*.tutorialspoint.com/genetic\_algorithms/genetic\_algorithms\_crossover.htm

[2] *Genetic Algorithms – Mutation.* Retrieved from *www*.tutorialspoint.com/genetic\_algorithms/genetic\_algorithms\_mutation.htm