Computer Science 3201 - Final Project Report EA - TSP Competition

Mackenzie Peyton 201530995
David MacInnis 201538949

Introduction

The Traveling Salesman Problem (TSP)

The traveling salesman problem is a very popular problem in the field of graph theory. It tasks the problem solver to find the shortest route for a salesman to visit a certain number of cities where each city is only visited once. The Idea was first studied in the 1800's by Irish mathematician, Sir William Rowan Hamilton when he created a game that required players to complete a tour through 20 points using only specified paths. This type of path is now known as a Hamiltonian path in respect to Hamilton's contribution. Throughout the years this optimization problem has built quite a reputation with mathematicians and computer scientists alike. This project takes a computer science look at the problem and looks to optimize the path utilizing an evolutionary algorithm to modify a randomly generated path to become an optimal path through mutation and recombination of multiple instances.

Our Task and Goal

For this program our goal was to design and implement an evolutionary algorithm (EA) to try and solve the Traveling Salesman Problem. For our EA, cities locations are represented as coordinates (x and y points) on a graph. The distance between two cities is the Euclidean distance of their coordinates. Our fitness function which evaluates how good an individual is, finds the sum of the total distance between all the cities. In this case the lower the sum of the distances the better, as it is less distance to travel.

Methods

General outline of algorithm:

1. Initialize population
2. Select parents
3. Make offspring from parents
4. Mutate offspring
5. Select the next generation from both parents and offspring

Initialization

Upon starting the algorithm,a set number of individuals are created and stored in an array. Each individual is represented as a list of the cities visited in order of which they are visited. This is accomplished by using the python function random.sample() on an initial list of the cities. Each city is then represented by their city number and their x and y coordinates.

Fitness

To test the fitness of an individual, we take the sum of the distances traveled between all the cities(further description in “Pre-Calculating all of the distances”). For each generation the fitness is corrected in proportion to the best performing individual. This is done by subtracting the difference, between the best performer and the current fitness, from the value of the best performer and then adding a buffer for each individual in the generation. This ensures that we have a well balanced distribution and the fitness is in ascending order from worst performer to best performer instead of the other way around. The buffer ensures the worst performing individual has a value of 1.

Scramble Mutation

As a mutation method we are using Scramble mutation. To accomplish this we pick two random points in a single individual, we then copy all cities located at indexes that fall between the two points to a list, the list is then randomized and reinserted back to the individual between the two indexed points. The new individual is then added to the offspring list to be tested for it’s fitness.

Randomized N point crossover (Advanced method)

For our recombination method we are using a Randomized N point crossover. Instead of having a fixed N like a regular N point crossover we are using a random number of crossover points. This is to mitigate any location bias between points as there is a roughly equal likelihood of having an even or an odd number of points. First, two individuals are selected for mating and labeled parent1 and parent2, after that the random N is calculated for every crossover instance to be between 0 and half the length of the parents. Then a list is created to contain N randomly generated crossover points. This list is then sorted and has any duplicates removed. From there two children are created, offspring1 and offspring2, and all cities between an even and odd indexed crossover point are copied directly to the child with the same number as the parent and every city between an odd and an even indexed crossover point is copied to the child with the opposite number as the parent. The two offspring are then added to the offspring list to have their fitnesses tested. Recombination has a 90% chance of occuring.

Pre-Calculating all of the distances

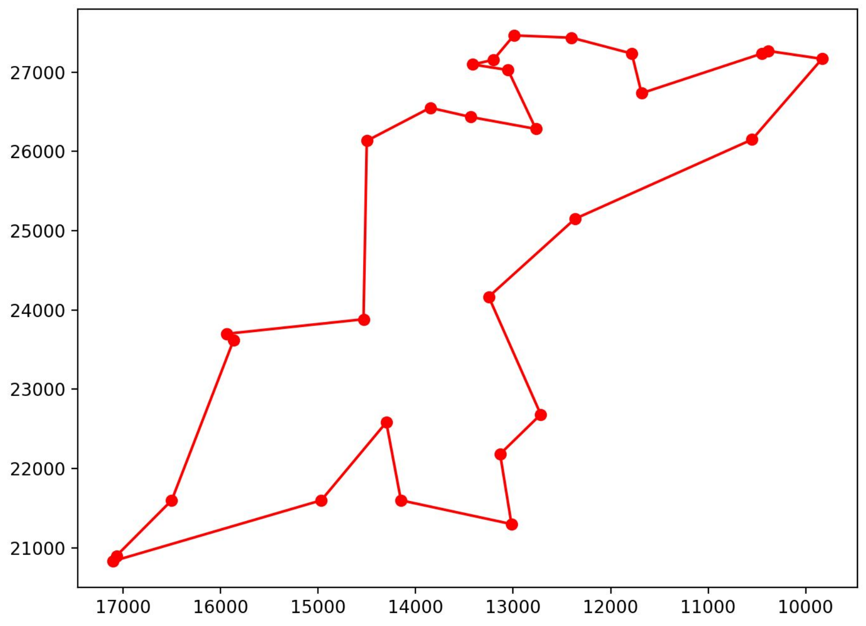
Before the algorithm starts all the xy coordinates are taken in from the given text file. These xy coordinates are put into a function that creates a dictionary of all possible distances between each city. By doing this, each possible distance calculation only has to be done once at the beginning which saves time later once the algorithm starts. Then later when the algorithm needs a distance between two cities it is simply a lookup in the dictionary using the city numbers.

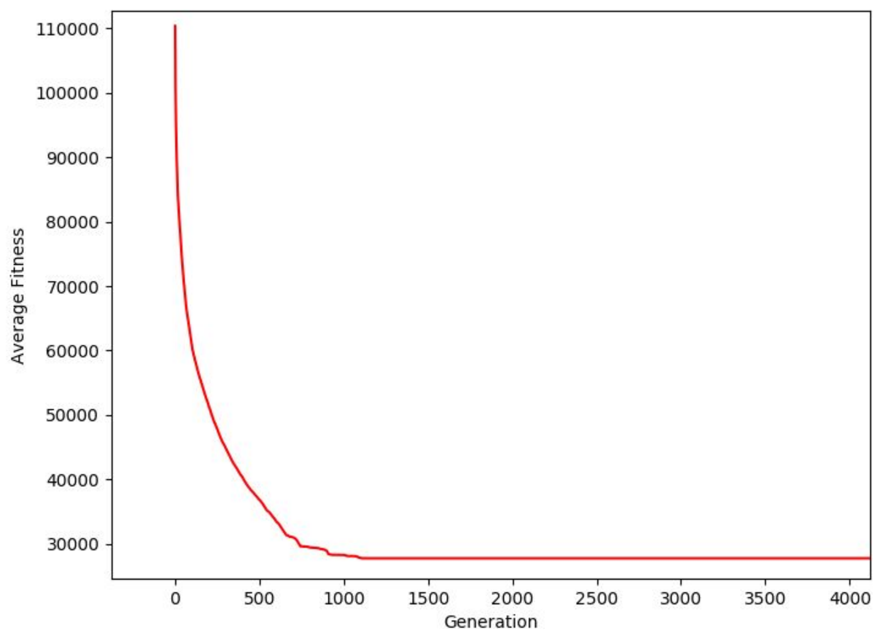
Simple stall detection mitigation

To prevent local maxima in our algorithm we added a stall check after each generation, if the average fitness stays the same for 10 generations in a row the algorithm will switch from a scramble mutation to a random swap mutation. In our case the random swap mutation picks two random points in the given individual and swaps them, this is helpful when the algorithm gets to a local maxima. This change will stay in effect until the average fitness changes, then it will revert back to a scramble mutation again until it stalls again.

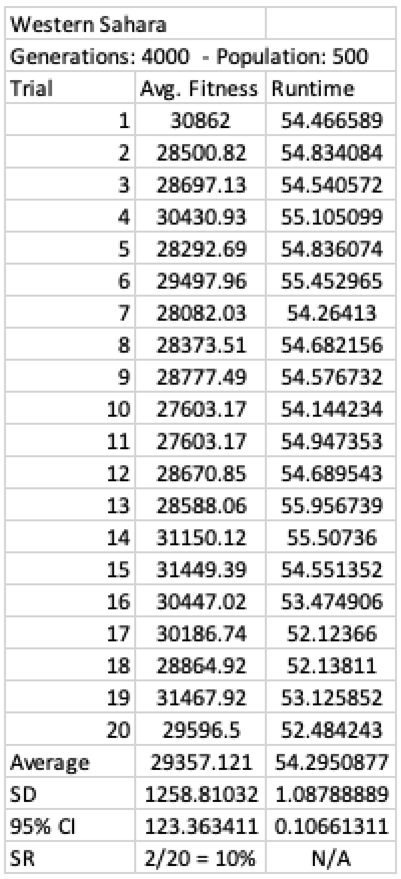
Results

According the University of Waterloo the optimal tour for Western Sahara is 27603 and using our algorithm we were able to achieve this tour (see below)





Below is a table of 20 sample runs done using the Western Sahara dataset (the file 'WesternSahara_20_runs.txt' with all the information on the runs can be found in the 'figure_routes' folder).

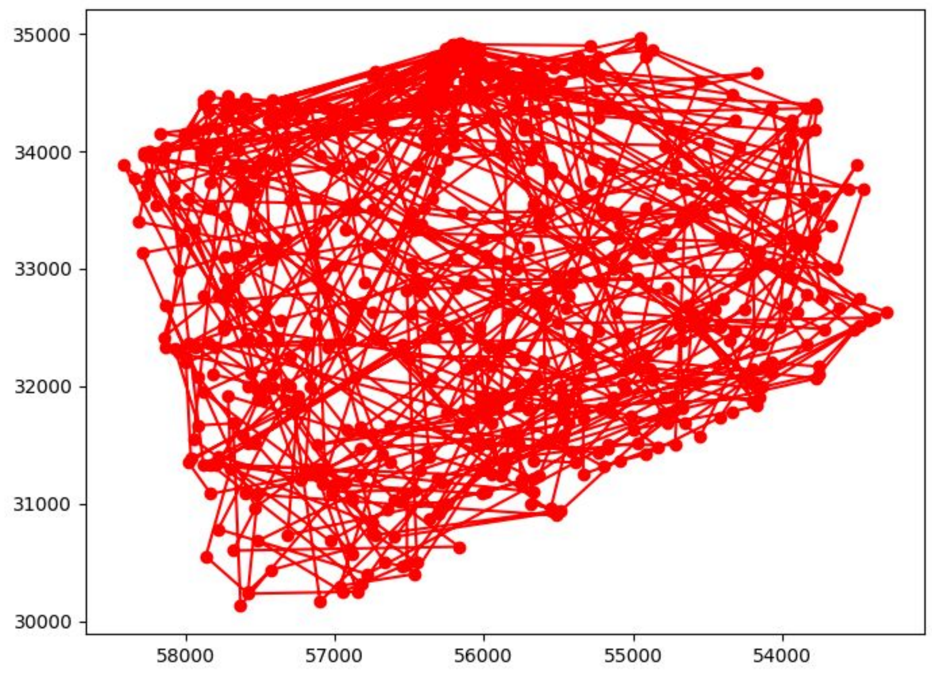


Western Sahara		
Generations: 4000 - Population: 500		
Trial	Avg. Fitness	Runtime
1	30862	54.466589
2	28500.82	54.834084
3	28697.13	54.540572
4	30430.93	55.105099
5	28292.69	54.836074
6	29497.96	55.452965
7	28082.03	54.26413
8	28373.51	54.682156
9	28777.49	54.576732
10	27603.17	54.144234
11	27603.17	54.947353
12	28670.85	54.689543
13	28588.06	55.956739
14	31150.12	55.50736
15	31449.39	54.551352
16	30447.02	53.474906
17	30186.74	52.12366
18	28864.92	52.13811
19	31467.92	53.125852
20	29596.5	52.484243
Average	29357.121	54.2950877
SD	1258.81032	1.08788889
95% CI	123.363411	0.10661311
SR	2/20 = 10%	N/A

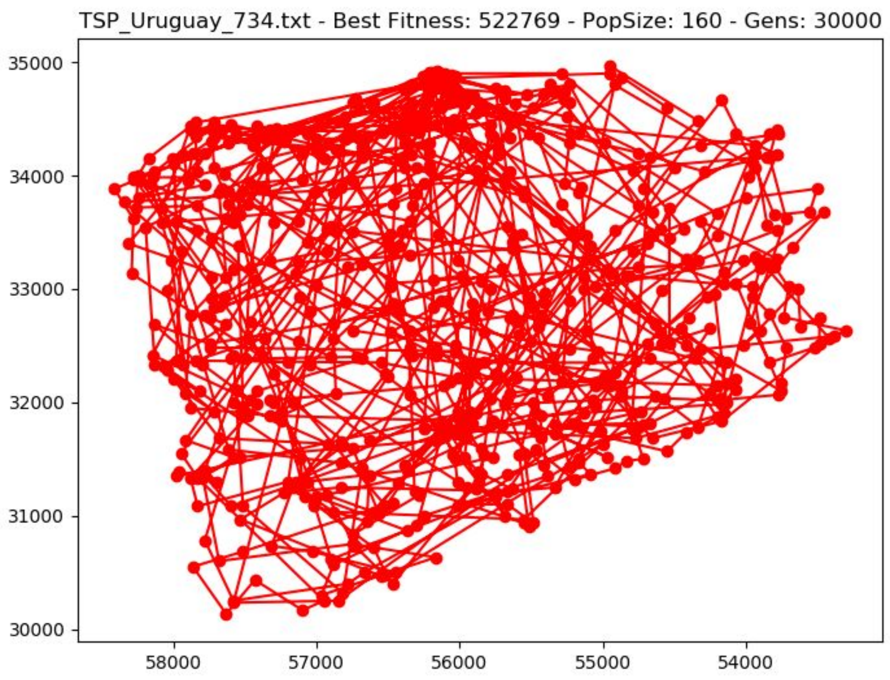
Local maxima are an issue for our algorithm (as can be seen in the source file for the table above and when running the algorithm itself), we would likely need to use some form of population management in order to help combat this better if we were to further develop this algorithm. The random swap does help after the algorithm stalls for a couple generations but a better solution is needed.

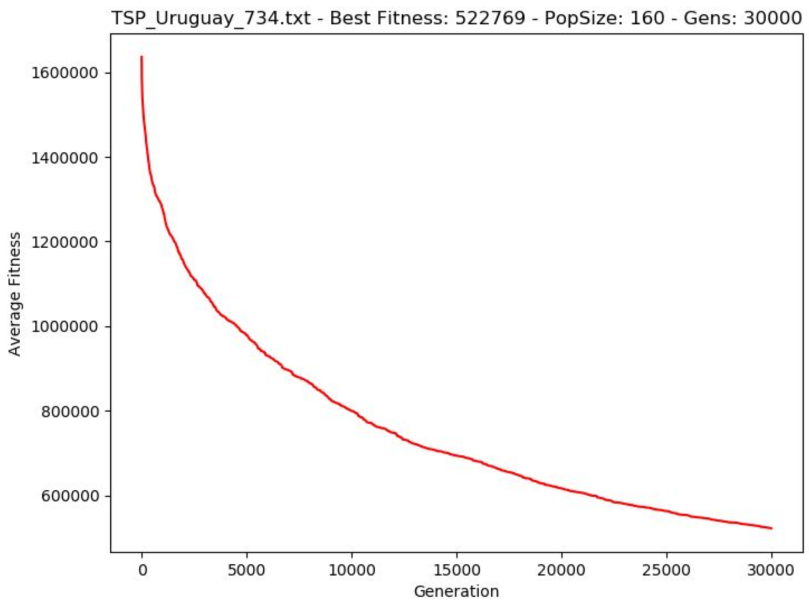
Below are two runs using the Uruguay dataset. It is clear that our algorithm needs more optimization as it just takes too long from the starting fitness in the first generation to get to something even remotely close to the optimal tour, which in this case according to the University of Waterloo is 79114.

Run 1:
This test was run early on without many optimizations, it had a best fitness of 687,072 after a starting best fitness of 1,570,118 and 30,000 generations with a population of 160.



Run 2:
This run was run later after optimizations and other features like stall mitigation were added. It had a best fitness of 522,769 after a starting best fitness of 1,553,480 and 30,000 generations with a population of 160.





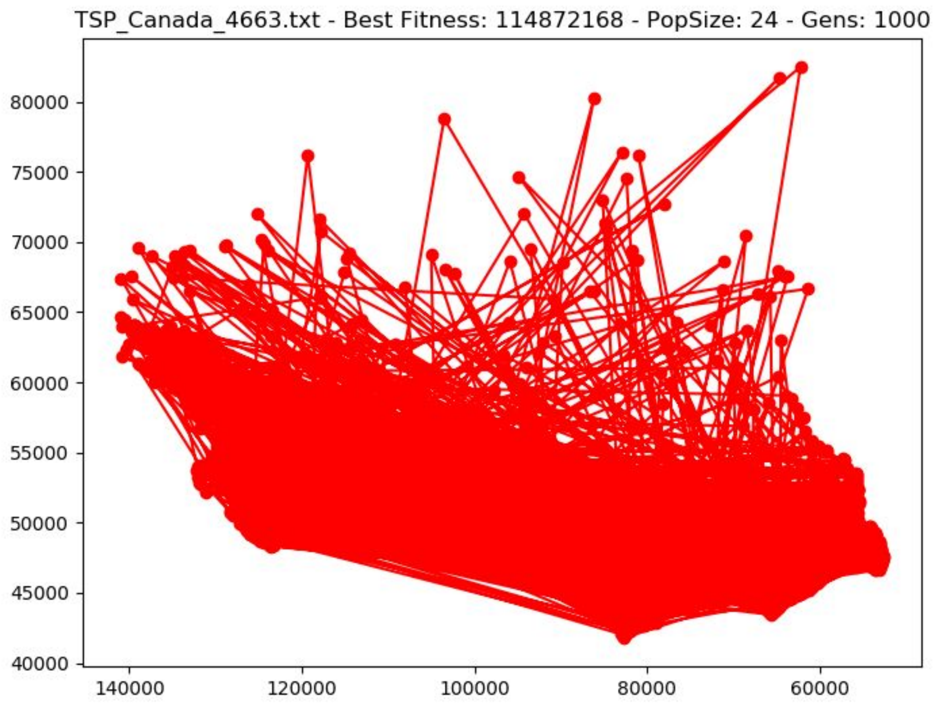
There is big improvement from run 1 to run 2 but clearly one of the biggest problems with the algorithm is the initialization, a random initialization works fine on small datasets but is not great for larger datasets like that of Uruguay. Some sort of heuristic would most likely be a lot better as it would greatly reduce the starting best fitness and average fitnesses, which in turn would reduce the total time it would take to get to the optimal tour of the given dataset.

Canada Dataset

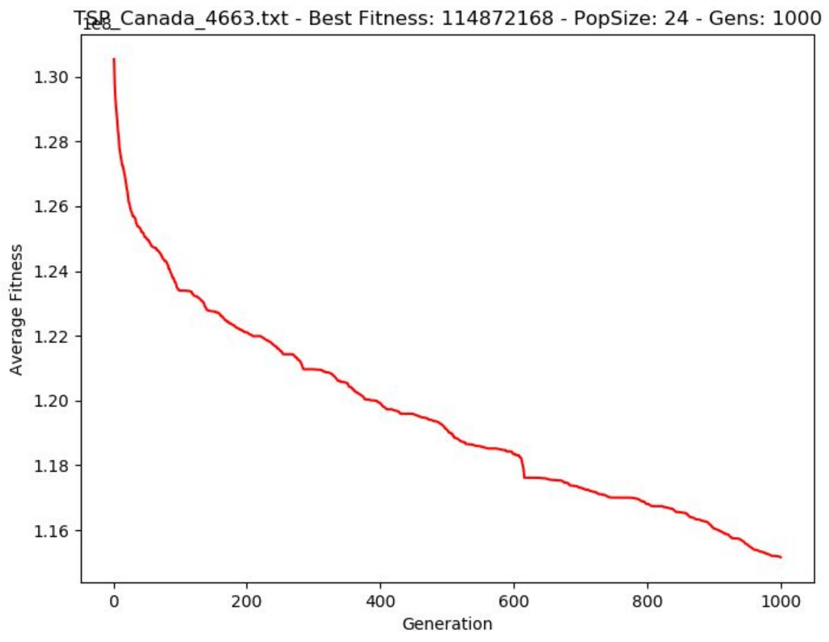
We tried running the Canada dataset through our algorithm here are some of the runs:

Run 1:

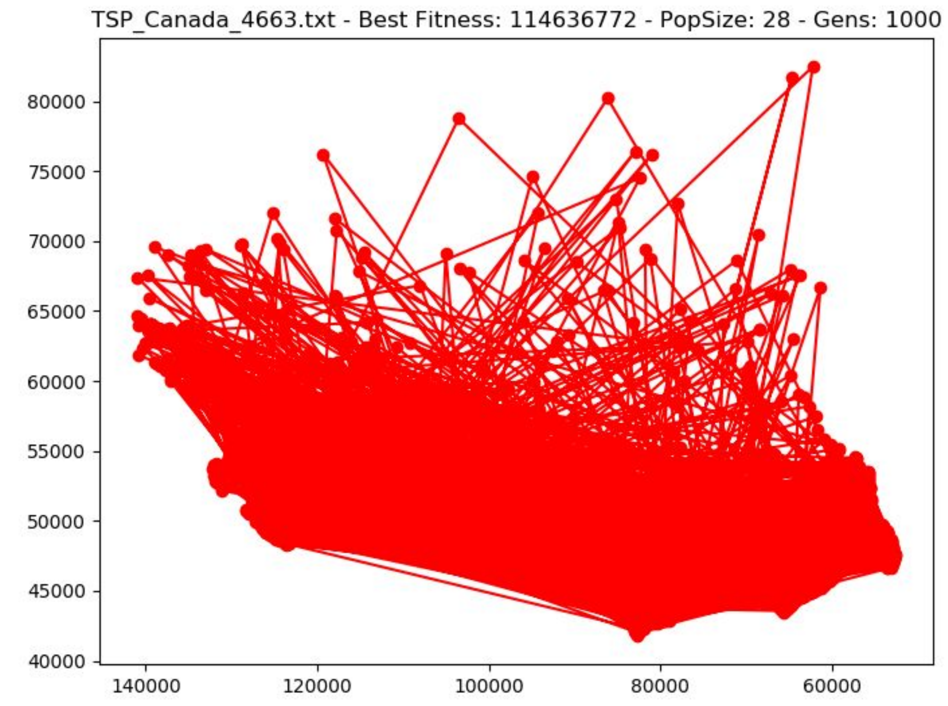
Starting best fitness of 127,654,902 with a final best fitness of 114,872,168 after 1000 generations with a population of 24

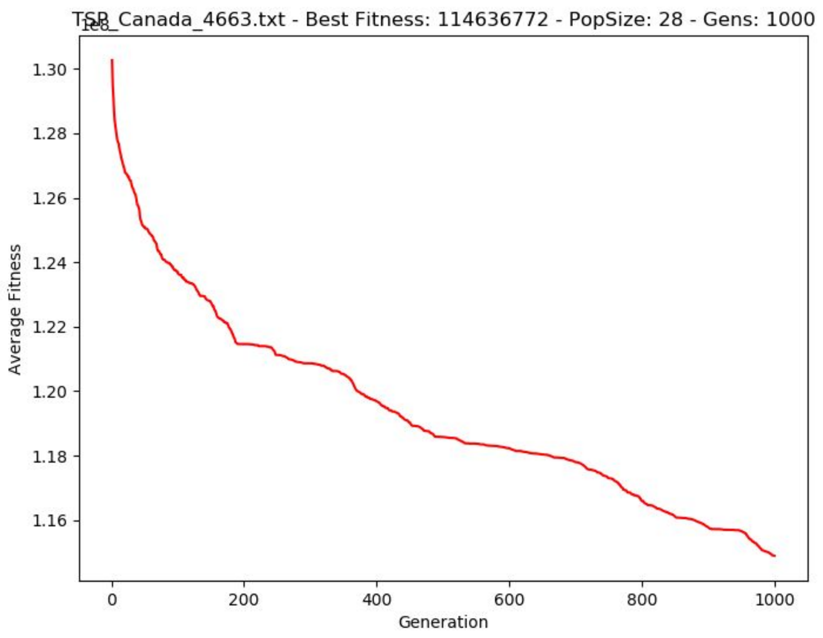


.



Run 2:
Starting best fitness of 127,386,739 with a final best fitness of 1146,636,772 after 1000 generations with a population of 28.





Discussion

What needs to change

For smaller data sets like that of the Western Sahara, our algorithm has minimal problems getting to the optimal tour or close to it. As described in the results section, when our algorithm gets close to the optimal tour local maximums become an issue for it, especially the cities that are very close together.

Potential improvements to the algorithm

Numpy arrays are something we wanted to add but did not have time to implement, if these were used instead of normal python arrays the algorithm would be more efficient due to how numpy arrays work. If we had more time we would have also looked into python multithreading as well as different python compilers like Numba, which could have made our algorithm run faster. As the code is currently written it all runs on one thread, so by spreading the workload around to different threads the calculations could potentially be done faster. Another method that looked promising if we had more time was instead of using random initialization would be to use a clustering method for the initialization.

References

Random Multipoint Crossover:
De Jong, K.A. & Spears, W.M. Ann Math Artif Intell (1992) 5: 1.
https://doi.org/10.1007/BF01530777

University of Waterloo. (2017, March 10). National Traveling Salesman Problems. Retrieved from http://www.math.uwaterloo.ca/tsp/world/countries.html

University of Waterloo. (2007, January). History of the TSP. Retrieved from http://www.math.uwaterloo.ca/tsp/history/index.html

Anaconda. (2018). Numba: A High Performance Python Compiler. Retrieved from http://numba.pydata.org/