Name: Hao (Henry) Fang

Student ID: 301301402

CMPT 310 Surrey Spring 2020

Final Project: Solving the TSP with Genetic Algorithms

# **Introduction:**

This paper is about solving the TSP (Travelling Sales Problem) using GA (Generic Algorithm) with Python. The concept of this algorithm was not using a mathematically guided algorithm, but an algorithm taken from how human and other species evolve through time. Even though the algorithm is mainly the same but there are many different crossover functions, selection methods etc. we can use to get a better result (Using less time and find a better solution which is a shorter distance for this TSP problem)

I used two classes to help me store the information of the cities and each individual:

A “City” class that has three variables

* name: Name of the city which is number 1 to 1000 from the cities list given
* xCoord: x coordinates for the city
* yCoord: y coordinates for the city

A “Individual” class that represent each individual in a population:

* individual: a permutation of the cities
* score: a score that is assigned for this individual

Note: I used the inverse of the distance so that it is easier to sort and visualize each individual

The higher the score, the shorter the distance, the better the permutation.

My GA has the following core functions:

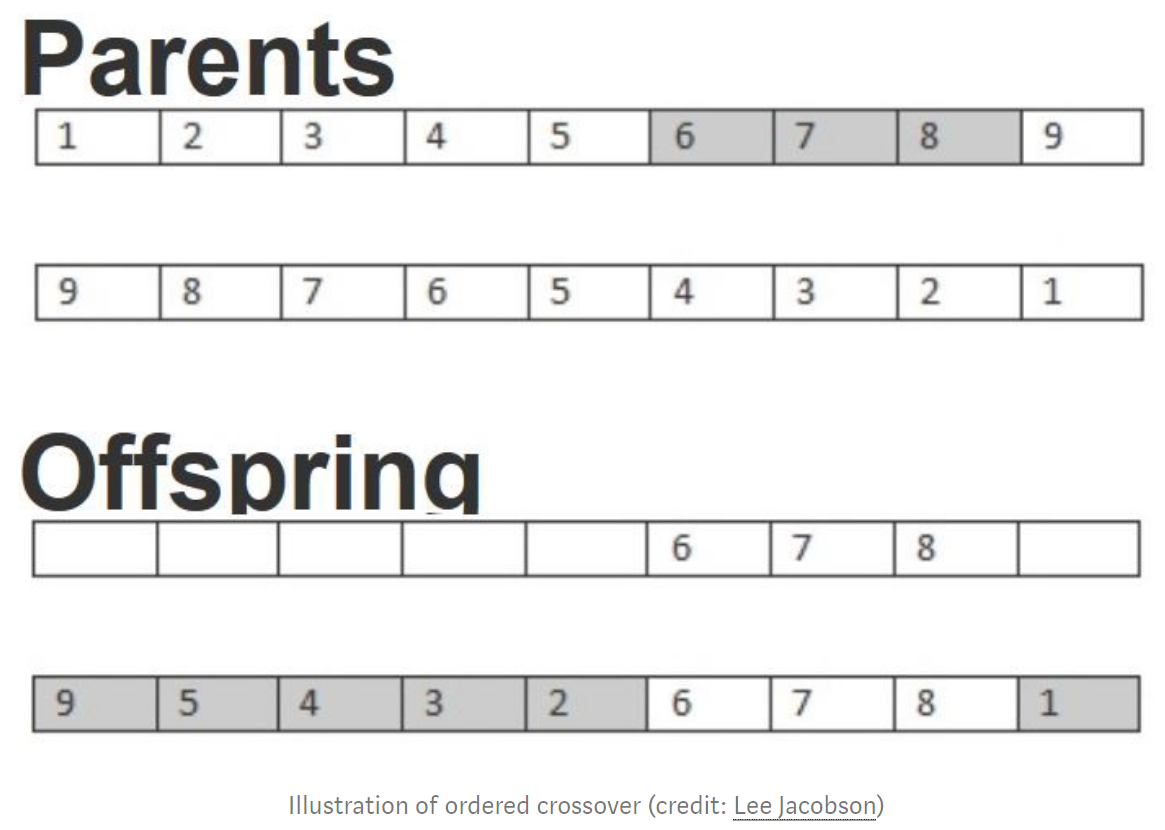
* rankPopulation: This function will rank each individual in the population in an ascending order of their score. The first individual is the best one
* getMatingPool: This will return a list of the individuals that will be the parents for the next generation
* generateChildren(Crossover): This is a crossover function that will generate the children from their parents.
* Mutate: This function will change an individual’s permutation by swap the cities.

# **Ideas and features tried:**

This section will discuss different crossover methods, mutate methods and selection methods used and compare their performance.

1. Ordered crossover

In ordered crossover, we randomly select a subset of the first parent string and then fill the remainder of the route with the genes from the second parent in the order in which they appear, without duplicating any genes in the selected subset from the first parent.



The final GA algorithm will be produced with another mutation methods. See below for details.

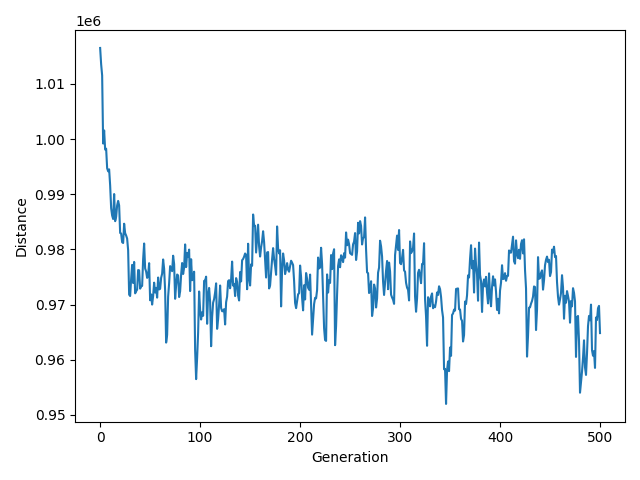
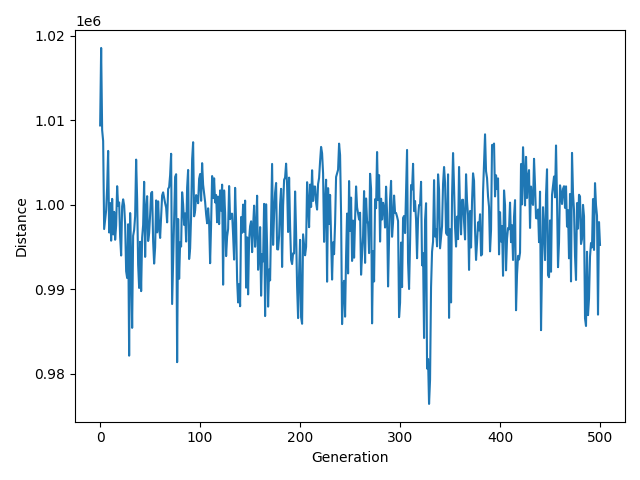
1. Mutation with mutation rate

When running the algorithm, I passed a mutation rate to the GA function. Each individual will have a possibility to “mutate” based on the mutation rate. The higher the mutation rate, the higher chance it will mutate (swap the cities). It turns out that this function is not as efficient.

Here are the results with 500 iterations, the first population is pure randomized:

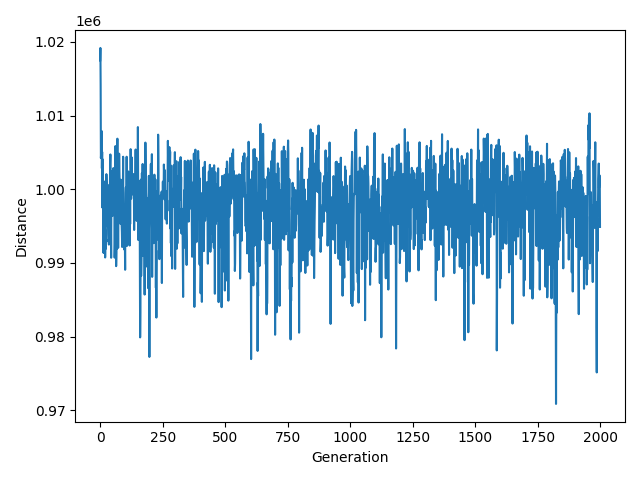
Note: Y-axis means the distance of the permutation. (Lower is better)

**Mutation rate = 0.05 Mutation rate = 0.01**



Here is another run with 1000 iterations:

**Mutation rate = 0.05**

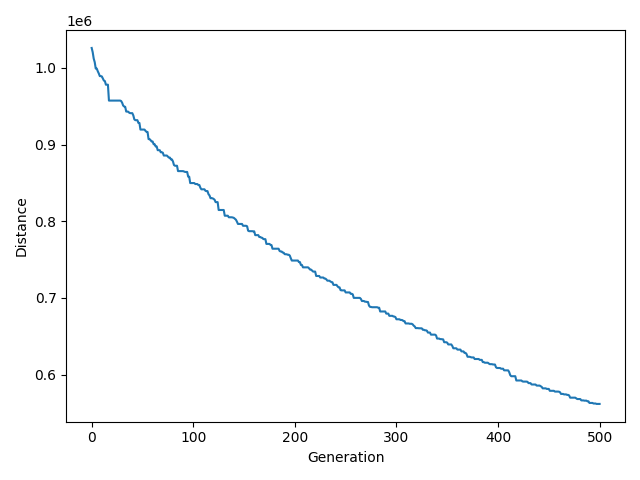


We can see here that the distance goes down at the beginning and then it is bumping up and down constantly.

1. Elite

After doing some research, I added “Elite” to my GA algorithm. In nature, elites are species who have better genes. Those who has better genes will eventually survive. I took this idea and added to my algorithm. Good permutations will be kept to next generation which made sure the distance won’t go up and down like the last method.

Here is a run with GA algorithm which included Elite:



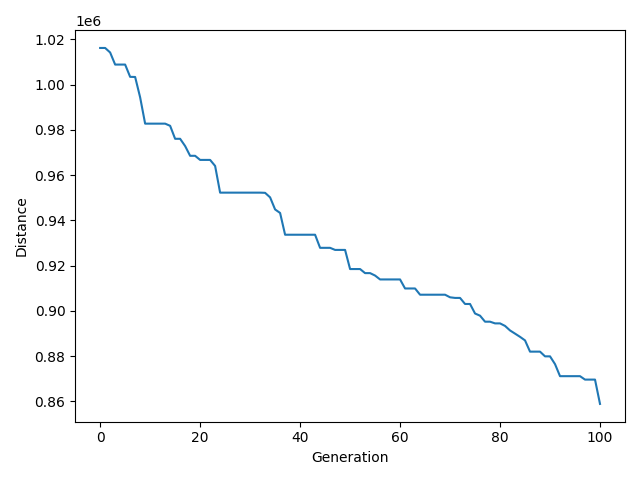
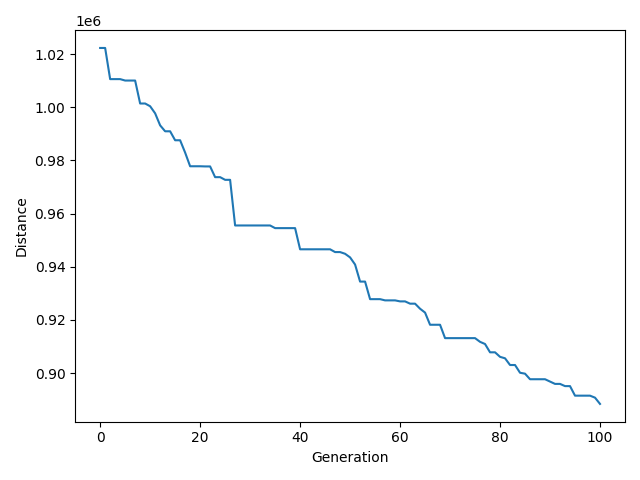
It is clear that this gives a much better result. The distance is constantly falling.

1. Performance analysis with population size and time

Next thing I did is to analyze the performance with different population size. In order to make the algorithm more efficient, I need to find out the relatively good population size with efficient time usage. The following two graphs shows the difference between population size of 100 vs 500 with same iterations.

**Population Size = 100 Population Size = 500**

**Time Elapsed = 14.98 seconds Time Elapsed = 84.04 seconds**



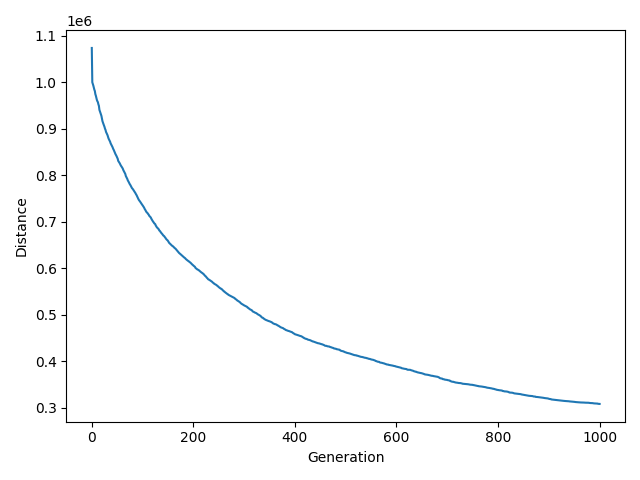
It’s clear that bigger population size will give us a better result, but it takes much more time. After some research and experiments, I found out that more iterations with relatively small population size is better

1. Partial-mapped Crossover (PMX)

# **Challenge Problem results:**

In the end, I have decided to use a combination previous method.

To generation next generation, I will first move the top 40 percent from the last population into the next population. Then use PMX crossover to generate 30 percent of the next generation and use ordered crossover to generate 20 percent of the next generation. For the last 10 percent of the generation, I will pick the best individual from last population and then mutate (randomly swap cities) it. This will make sure that we will not stuck a local minimum.



**Reference**

1. **https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35**
2. **http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/PMXCrossoverOperator.aspx**
3. **https://www.mdpi.com/2078-2489/10/12/390/htm**
4. **http://www.cs.unh.edu/~sc1242/publications/08423877.pdf**