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 (Genetic 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

An "Individual" class that represents 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 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.

Note: The framework is written by myself, some ideas were taken during research. See reference for the actual website. Please see "readme.txt" for how to run the code.

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 the 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 other mutation methods. See below for details.

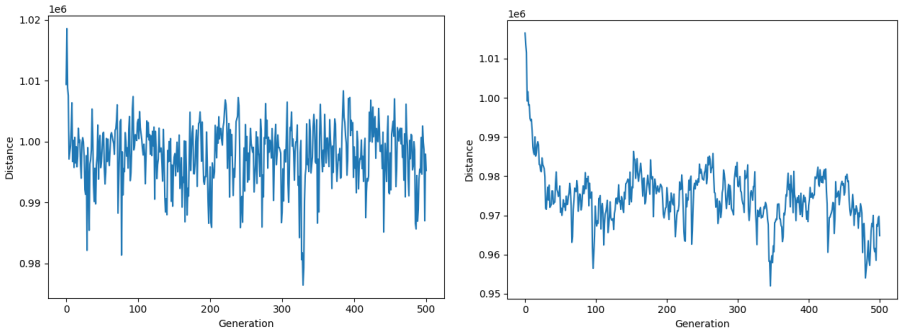
2. Mutation with mutation rate

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:

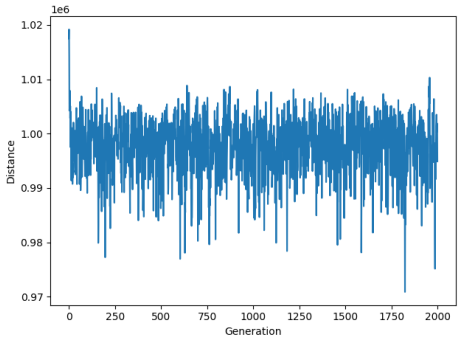
Note: Y-axis is the distance of the permutation. (Lower is better)

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



Result with 1000 iterations:

Mutation rate = 0.05

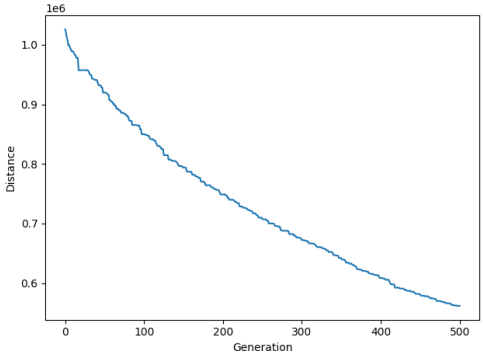


We can see here that the distance goes down at the beginning and then it is bumping up and down constantly.

3. Elite

After doing some research, I added "Elite" to my GA algorithm. Good permutations will be kept to next generation, which made sure the distance will not go up and down like the last method.

Result with GA algorithm which included Elite with 1000 cities:



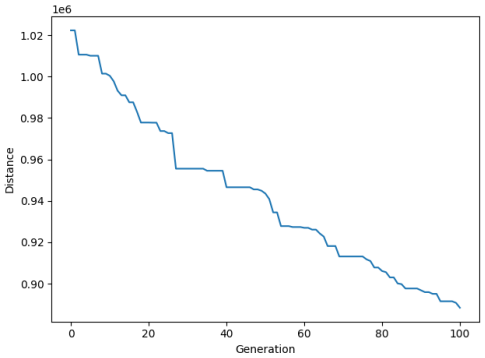
This gives a much better result. The distance is constantly falling.

4. Performance analysis with population size and time (with **Roulette Wheel Selection added**)

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

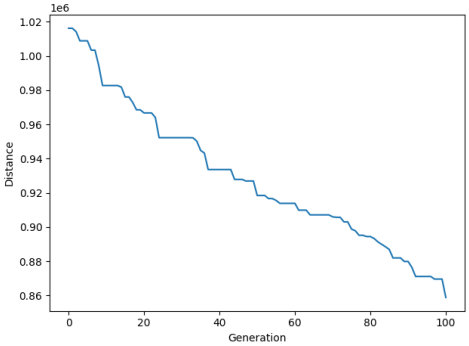
Population Size = 100

Time Elapsed = 14.98 seconds



Population Size = 500

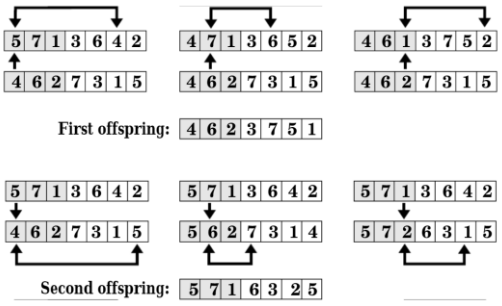
Time Elapsed = 84.04 seconds



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 are better

5. Partial-mapped Crossover (PMX)

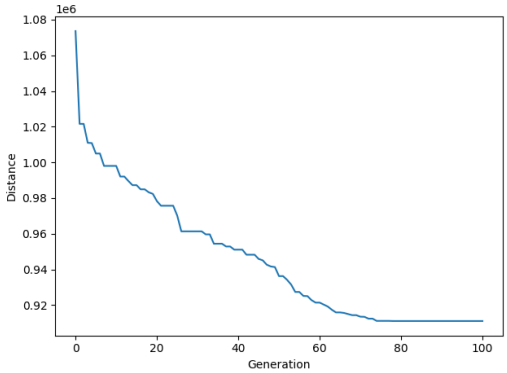
"The algorithm of PPX is that parent 1 donates a swath of genetic material and the corresponding swath from the other parent is sprinkled about in the child. Once that is done, the remaining alleles are copied directly from parent 2."



Performance graph with PMX:

Population Size = 500

Time Elapsed = 74.92 seconds



Challenge Problem results:

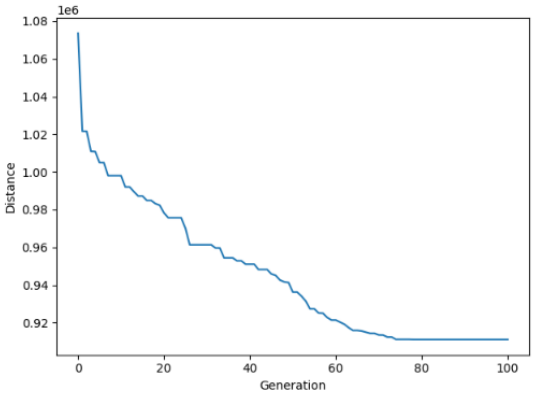
In the end, I have decided to use **a combination of previous method (My own idea)**

In the end, I have decided to use a combination of the previous method. To generation next generation, I will first move individuals at the top 30 percent from the last population into the next population. Then use PMX crossover to generate 40 percent of the next generation and use ordered crossover to generate 20 percent of the next generation (parents are randomly picked from the 40 percent added previously). For the last 10 percent of the generation, I will pick the best individual from the last population and then mutate (randomly swap cities) it. will make sure that we will not stay at a local minimum. These parameters were calculated by myself during experimentation with different numbers.

Here is comparison of different GA algorithms (Same number of iterations):

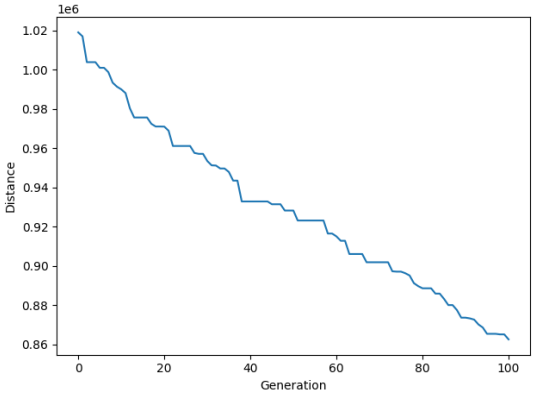
GA with only PMX

Elapsed Time = 74.92 seconds



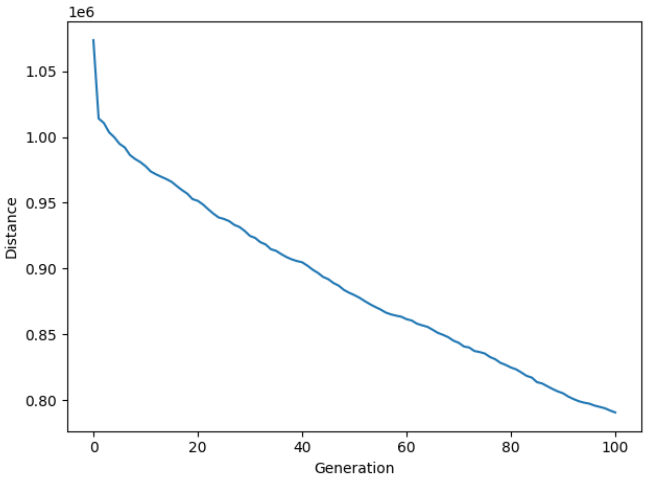
GA with only ordered crossover

Elapsed Time = 85.00 seconds



GA with combined crossover and mutation

Elapsed Time = 61 seconds

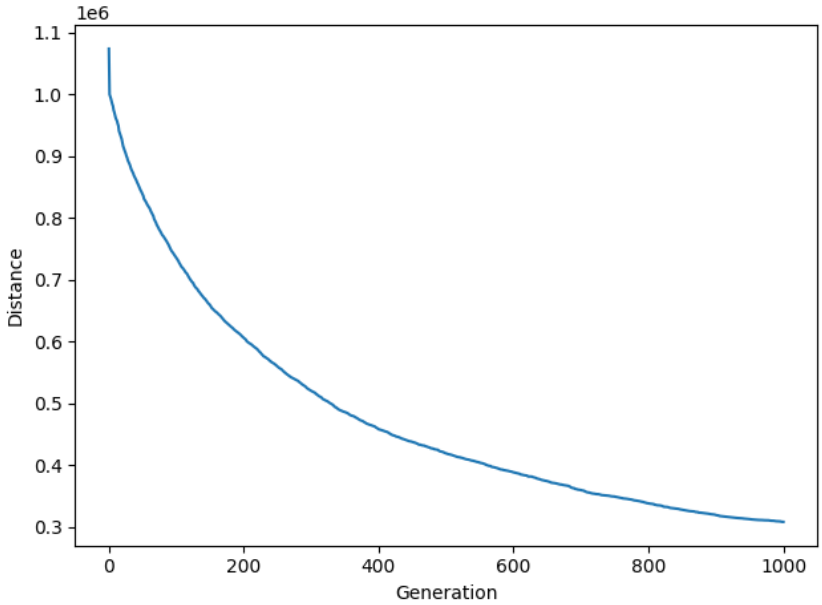


As we can see from the above graph, the GA algorithm with combined crossover and mutation works better than the GA with PMX and GA with ordered crossover. **Most importantly, it has the shortest running time.**

Result:

For the final answer, please see “best-solution1000_fanghaof.txt” for the full permutation. The shortest distance I got is **113272.04050499095**. Its hard to tell how much time exactly was used to get this answer because I used previous best permutation to calculate my next generation. To make a contrast, here is a graph with 1000 iterations. The time used approximately to get the best permutation is 10 hours.

Time Elapsed: 685.02 seconds



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**
5. **https://www.researchgate.net/publication/245746380_Genetic_Algorithm_Solution_of_the_TSP_Avoiding_Special_Crossover_and_Mutation**

Note: I wrote the framework by myself. The PMX code was taken from the code provided by Professor Toby Donaldson. The ordered crossover code was taken from the research paper.