**Travelling Salesman Problem**

**Using Genetic Algorithms**

Section 2 group 204

Team member:

Yunan Shao(001818832)

Yuchen Qiao(001293335)

Vinod Thiagarajan(001237129)

# Abstract

Genetic algorithm (GA) is a met heuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

In this project, we will create a genetic algorithm that calculate best path. The path is represented by the order of city, the fitness is 1/the total distance from start city to each other cities once then return back to start point (\*1000 for reading purpose).

# Implementation concepts

1. Seeding: When importing the city list into the original population as individual, the array list that for each individual will be shuffled for randomization.

2. Evolve: Produce next generation by eliminating the second half of population after sorting by fitness. Then mating/breeding to have children by crossover and mutation. And fill the rest space using the selected/survivor pool to reach the maximum population.

3. Culling: Select the best half of the population. Individual class implements comparable, compareTo function is used to reverse the order when sorting.

4. Crossover: Select part of the gene from parent1 and fill the rest using missing parts from parent2.

5. Mutate: Randomly swap the order of gene (city) for each child, have two implementations. In this project, we have two kinds of mutate.

5.1 Mutate: Every city has a chance decided by mutation rate to swap position with one of the rest cities

5.2MutateAlt: Alternative method, only swap once.

\*ReportWriter class works as a logger to record the results for each generation

# Pseudo Code for Basic Genetic Algorithm

The pseudo code for a basic genetic algorithm is as follows:

1: generation = 0;

2:population[generation]=initializePopulation(populationSize, citylist);

3: evolve(population[generation]);

3: while generationCount<maxGeneration do

4:parents = selectParents(population[generation]);

5:population[generation+1] = crossover(parents);

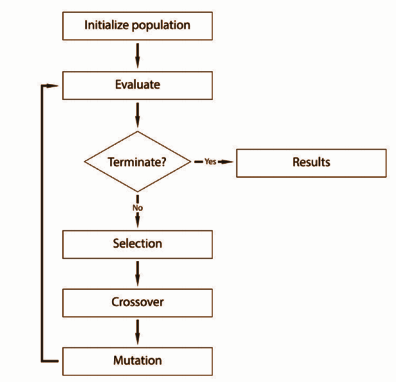
6:population[generation+1]=mutate(population[offspring]);

7: fillPopulation(population[generation]);

8: generation++;

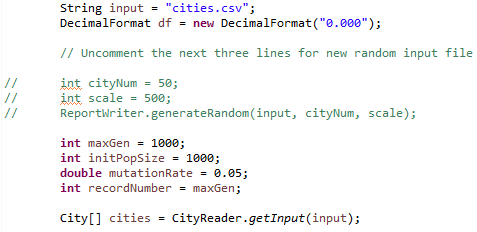
9: End loop;

**Layout**



# Parameters Setting

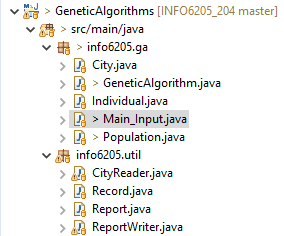
In this project, we set default parameters as follow:



The initial population is 1000 and max generation is 1000. When reach max population, the program will stop and print result.

# Program Structure

Src folder contains ga and util parts.



ga part is the core of this project, including:

-City.java

-GeneticAlgorithm.java

-Individual.java

-Main\_Input.java

-Population.java

util part mainly read file and output log files, including:

-CityReader.java

-Record.java

-Report.java

-ReportWriter.java

# Work flow

Basic step of genetic algorithm

1. Main\_Input read the cities.csv file in the input folder under application root path. (If a new set of cities need to be created, the code in ReportWriter can be used. Read the instructions in Read.me file). The different permutations of cities will be stored in population as individuals which represent different routes.

2. Then, the population is assessed by assigning fitness values ​​to each individual in the population. At this stage, we often pay attention to the most suitable solution at present, as well as the average fitness of the population.

3. The main function will start to evaluate the population and stop after the max generation has been reached, which means the evolve function will be called maxGen times.

4. For each evaluation, the group goes through a selection phase in which individuals from the group are selected based on their health score - the higher the fitness, the greater the chance that the individual will be selected.

5. The next stage is to apply crossover and variation to selected individuals. And then apply mutations depends on the mutation rate on offspring. This stage is where new individuals are created for the next generation.

6. At this point, the new population will return to the assessment step and start the process again. We call each cycle of this loop a generation.

7. When the maximum generation number has been reached, the best candidate will be print to console and stored in the report log file.

Here are steps our evolve function works:

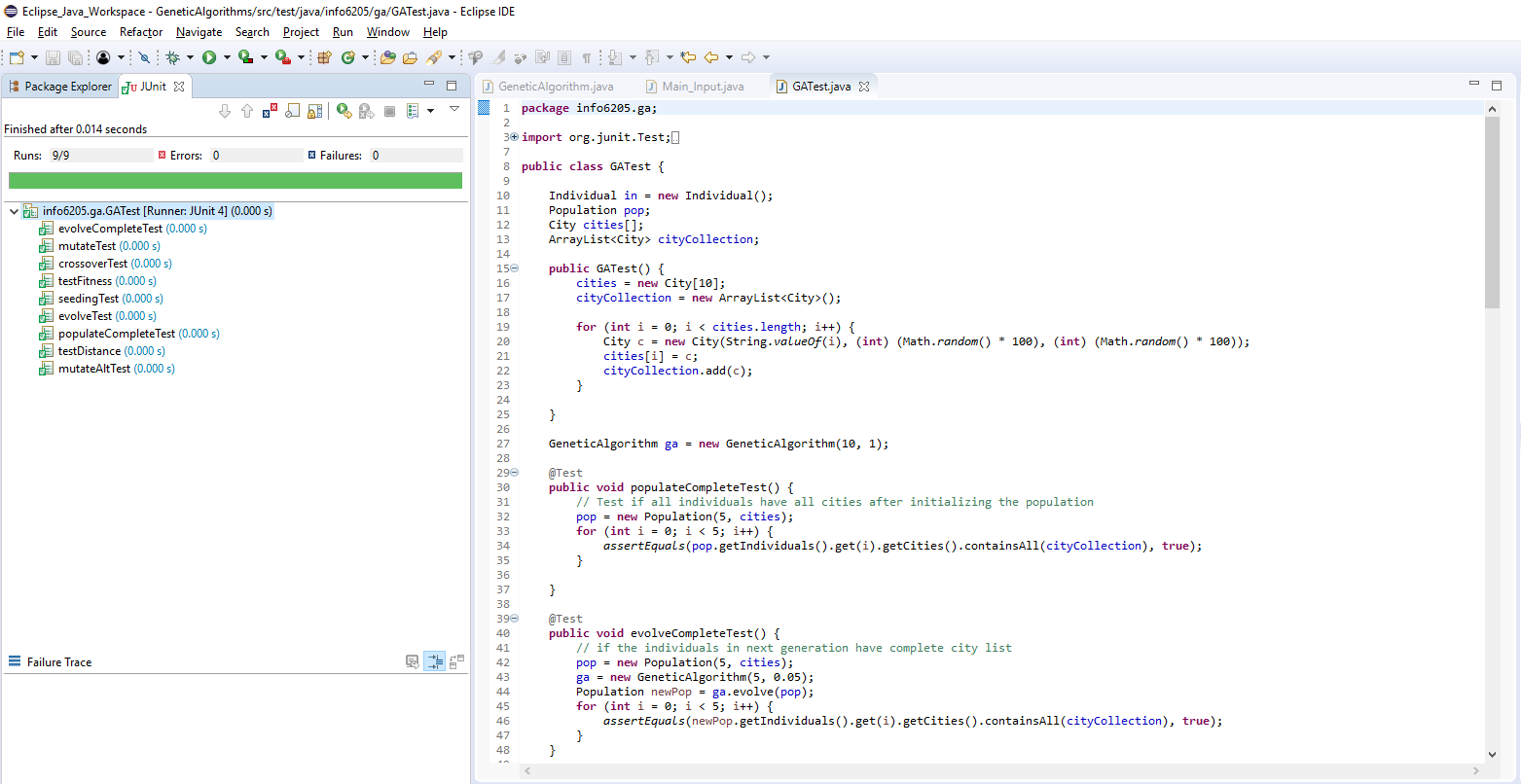
1. First, select the best half of the generation to create the survivor pool, where off springs are generated from selected individuals in this pool. In fitness calculation, we simply use 1/totalDistance of the route as number of fitness.
2. In crossover part, we randomly select two individuals as parents from survivor pool. The constraint is that parents can't be the same individual. The offspring group will be stored into next generation first.
3. Then, fill the next generation with survivor pool until it reaches maximum number of individuals.

# JUNIT TEST CASES

All the test case successfully run.

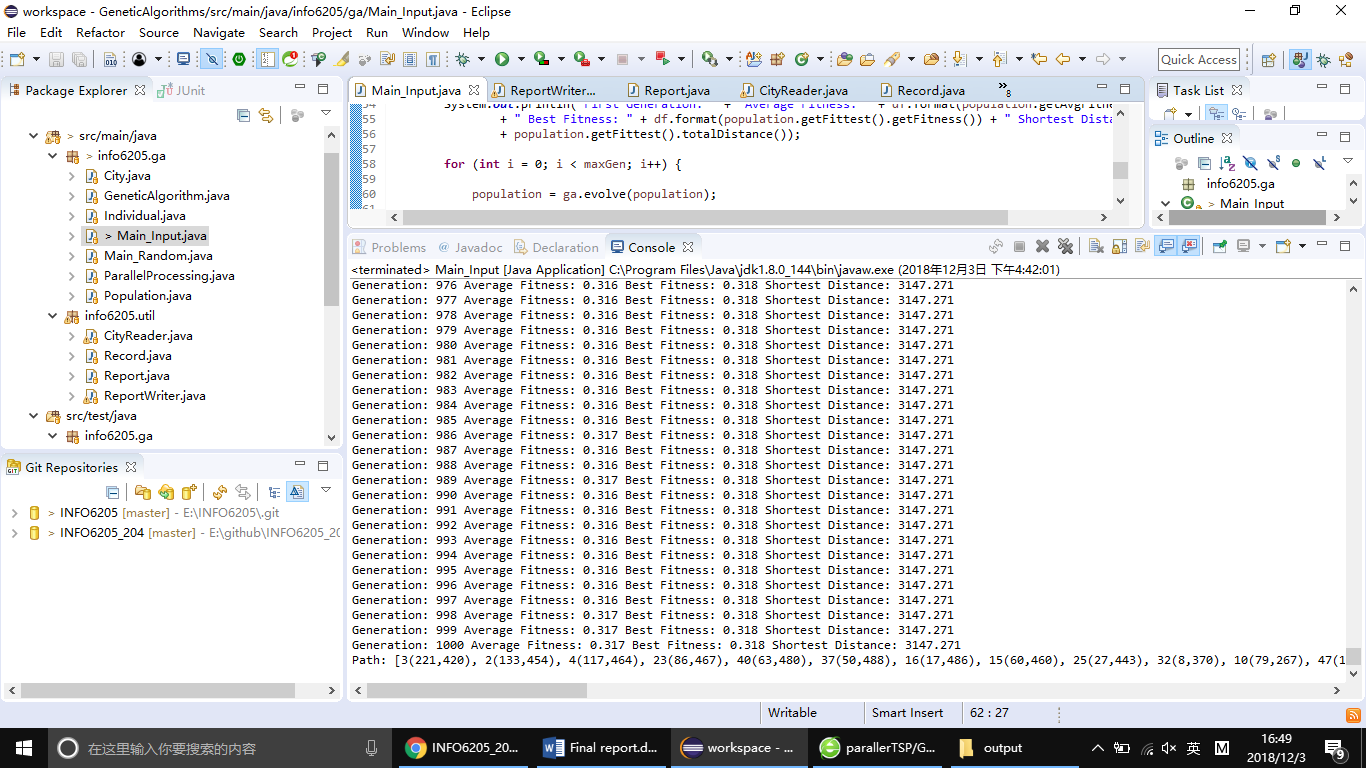
Our test cases are as follows:

1. If all individuals have cities after initializing population
2. If all individuals in next generation have complete city list
3. If new population has better fitness as previous one
4. Testing mutating function
5. Testing if seed process for first generation randomizes every individual
6. Testing mutate alt functions
7. Test crossover by comparing child with parents
8. Fitness test
9. Distance test



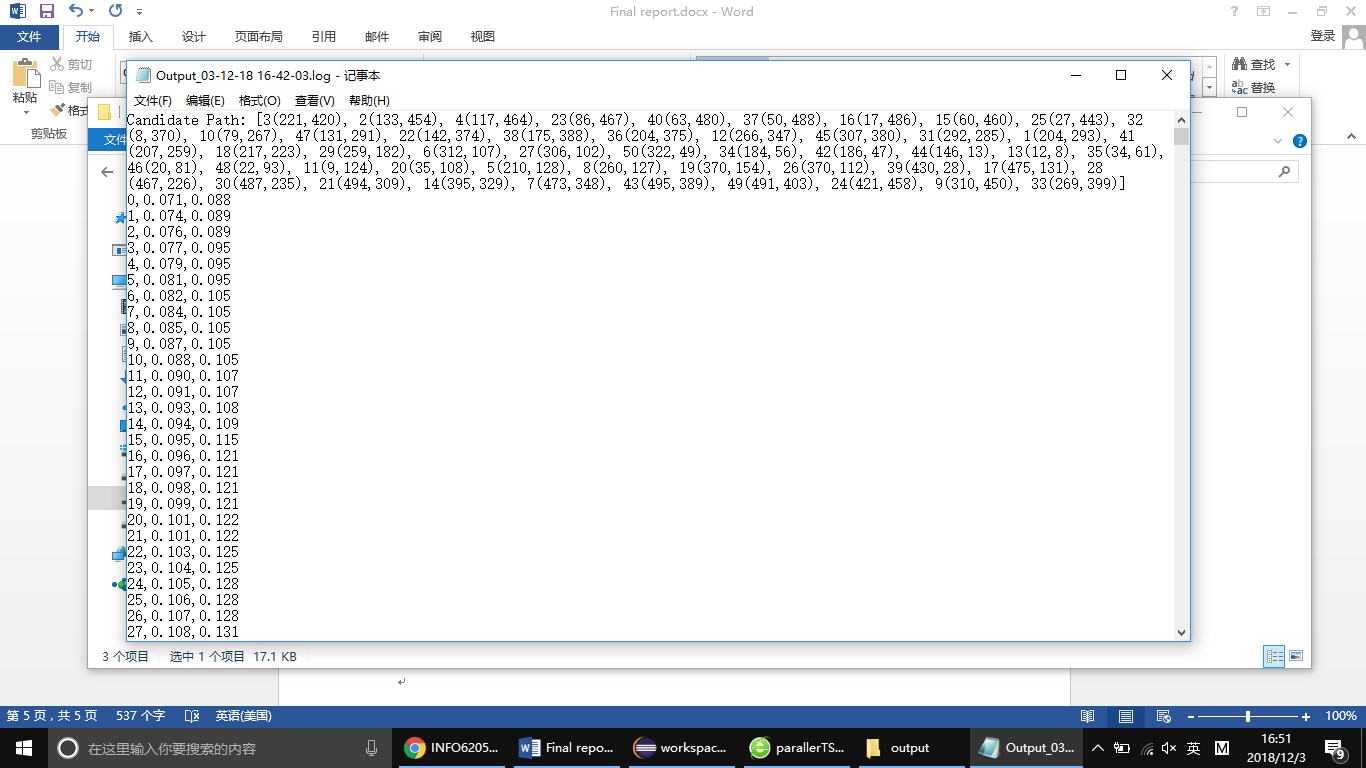
# Running results

Here is the running result for this project:

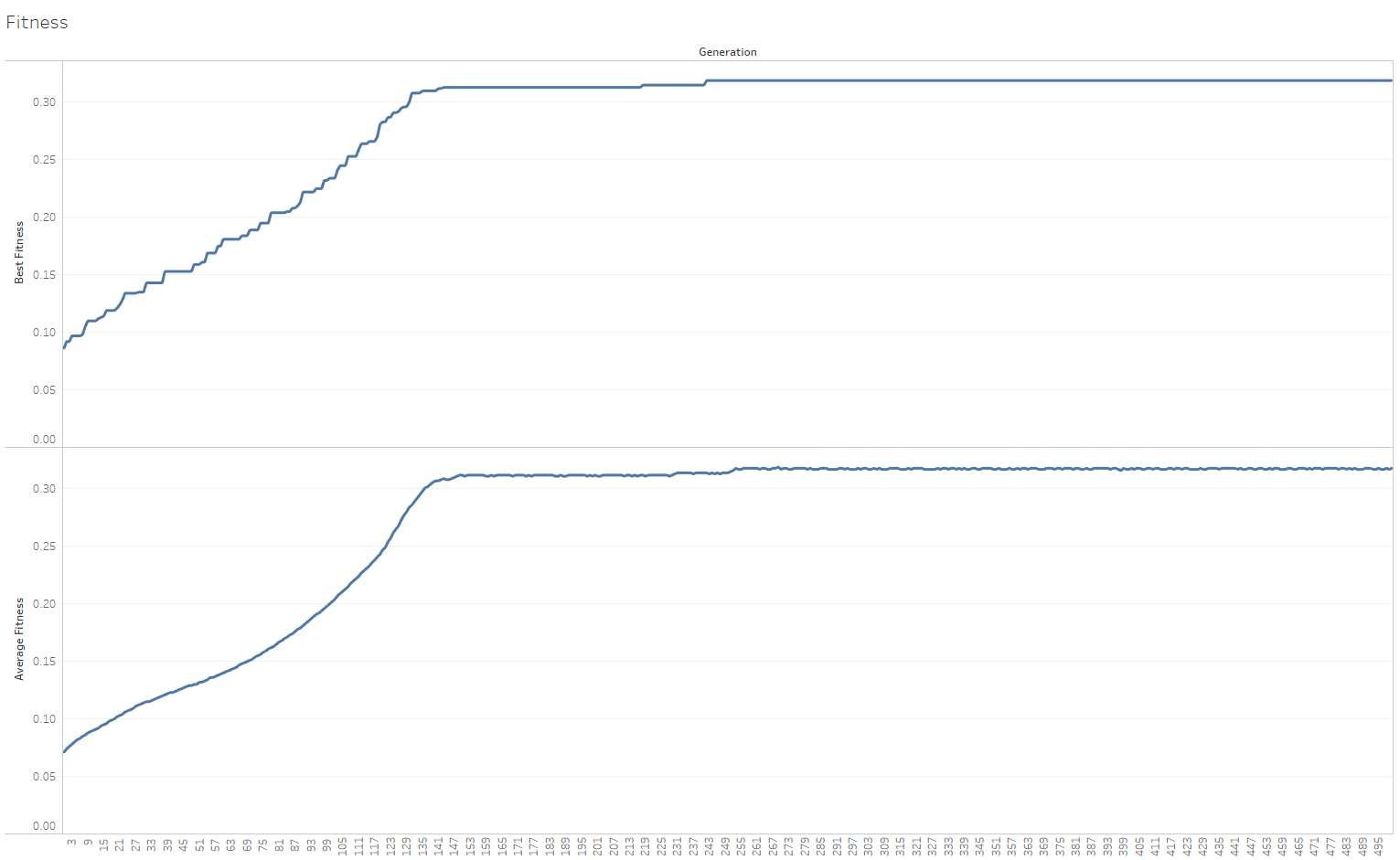


The shortest distance, average fitness and best fitness of each generation will be printed. The best fitness and shortest distance don’t always have better result comparing to previous generation because the best half of previous population is still inside the current generation. In this situation, it means the offspring group doesn’t produce the better children which have better fitness than parents.

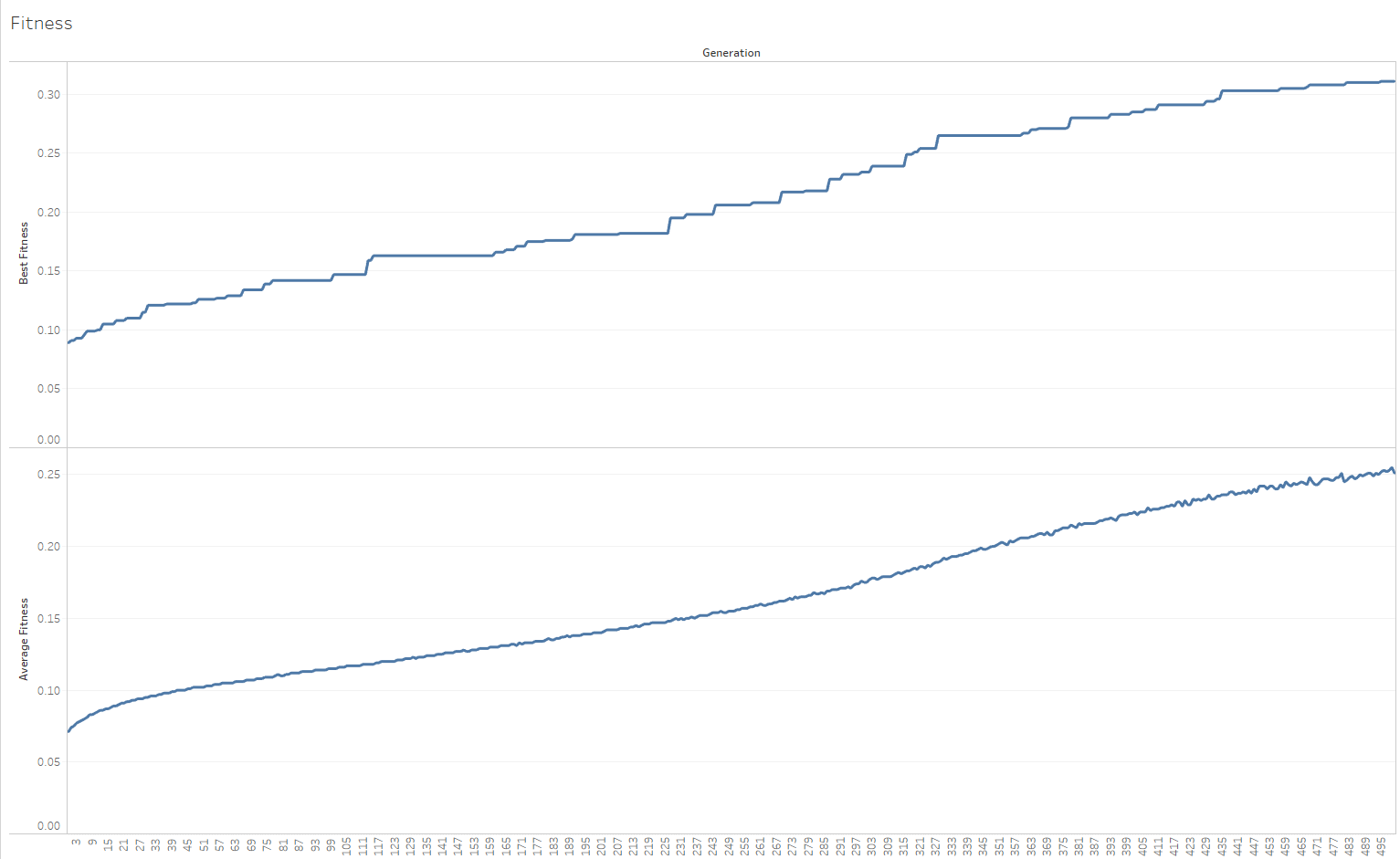
We also generate log file as follow:



Best candidate is stored first and the rest is all statistics of each generation. We only keep numbers and split them with ‘,’ for plotting the results.

**Result**

Fitness vs Generation (mutateAlt method, swap once)



Fitness vs Generation (mutate method, do swap for each city)

Our mutateAlt function shows better performance finding the relatively optimal solution in fewer generations. Our genetic algorithm doesn’t always produce better solution in next generation because the best candidate is already in it (Best individuals in the survivor pool).

Our genetic algorithm gets first possible optimal candidate in about 150 generations (50 cities, 0.05 mutation rate for mutateAlt funtion). With fewer cities the generations taken will be decreased (tested 10 generations for 10 cities) because there are fewer number of permutations. We would recommend a mutation rate under 0.1 in our genetic algorithm since our breeding strategy is producing offspring from the survivor pool. Higher mutation rate will increase the possibility of decreasing fitness of the offspring.

The crossover function is pretty tricky. We set the selection range under 70% of the city list size since we want to ensure the crossover will give us different routes in children. We found the crossover might produce the same permutation as parents for children with smaller or larger range when writing the test cases.

# Reference

[1]. <https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications>

[2]. <https://github.com/Apress/genetic-algorithms-in-java-basics/tree/master/GA%20in%20Java>

[3]. <https://www.apress.com/us/book/9781484203293>

[4].<http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5>