

# **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 meta 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
  - 5.2 MutateAlt: Alternative method, only swap once.

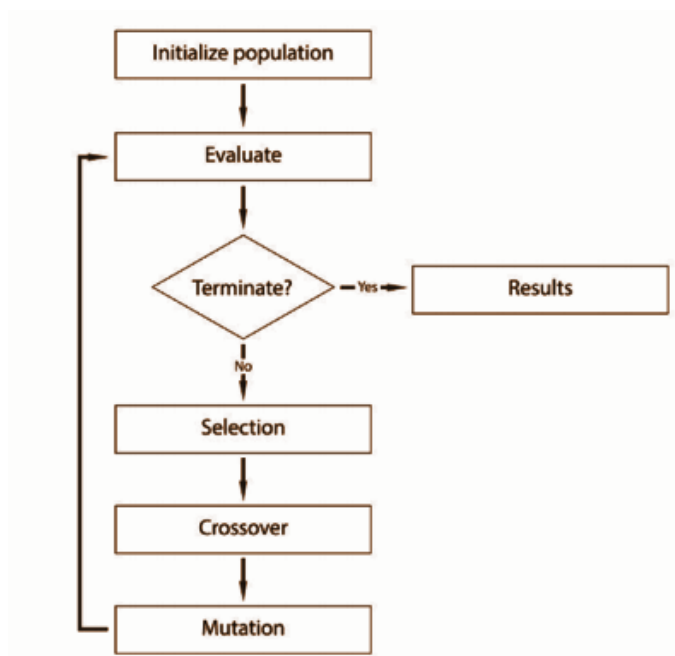
\*ReportWriter class works as a logger to record the results for each generation

# 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:

```
String input = "cities.csv";
DecimalFormat df = new DecimalFormat("0.000");

// Uncomment the next three lines for new random input file

// int cityNum = 50;
// int scale = 500;
// ReportWriter.generateRandom(input, cityNum, scale);

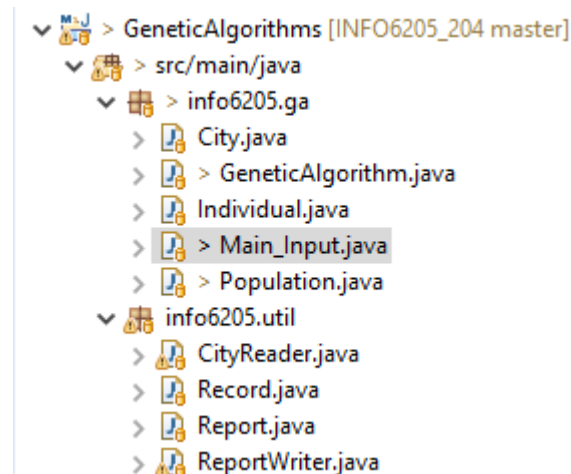
int maxGen = 1000;
int initPopSize = 1000;
double mutationRate = 0.05;
int recordNumber = maxGen;

City[] cities = CityReader.getInput(input);
```

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:

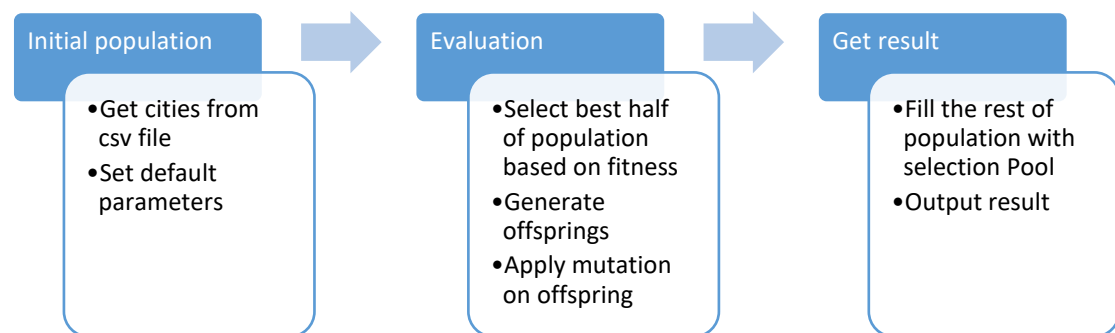
- City.java
- GeneticAlgorithm.java
- Individual.java
- Main\_Input.java
- Population.java

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 offsprings are generated from selected individuals in this pool. In fitness calculation,

we simply use  $1/\text{totalDistance}$  of the route as number of fitness.

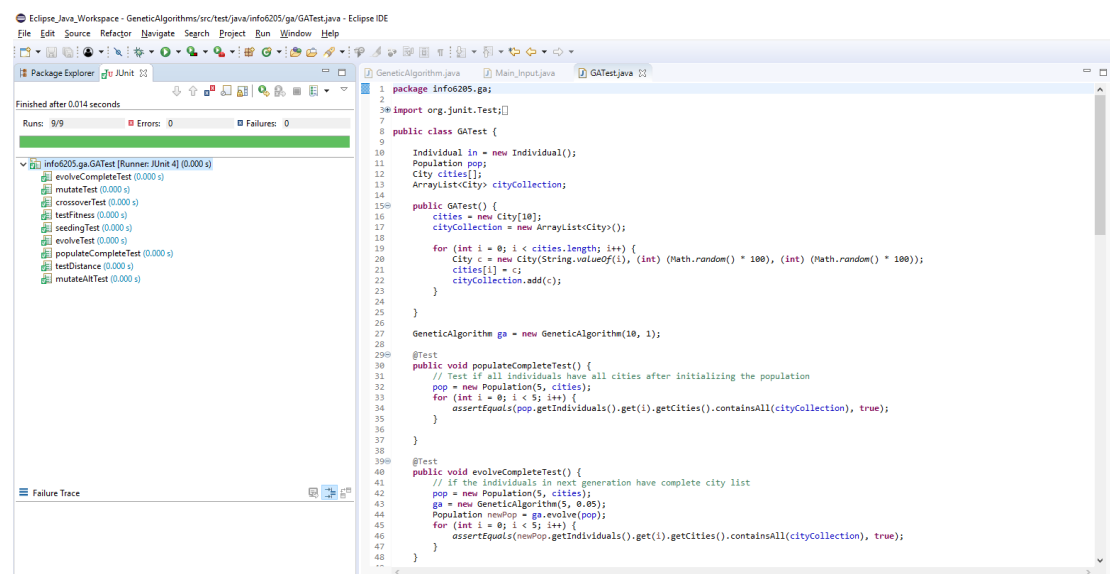
2. In crossover part, we randomly select two individuals as parents from survivor pool. The constraint is that parents can't be the same individual. The offspring group will be stored into next generation first.
3. Then, fill the next generation with survivor pool until it reaches maximum number of individuals.

## 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



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the test results for 'info6205.ga.GATest'. The tests passed, with a total time of 0.014 seconds. The main editor shows the source code for 'GA.java' and 'GATest.java'. The 'GATest.java' file contains several JUnit test methods: 'populateCompleteTest()', 'evolveCompleteTest()', 'mutateTest()', 'crossoverTest()', 'testFitness()', 'seedTest()', 'testDistance()', and 'mutateAltTest()'. The code is written in Java and uses the 'org.junit' package for testing.

```
1 package info6205.ga;
2
3 import org.junit.Test;
4
5 public class GATest {
6
7     Individual in = new Individual();
8     Population pop;
9     City cities[];
10    ArrayList<City> cityCollection;
11
12    public GATest() {
13        cities = new City[10];
14        cityCollection = new ArrayList<City>();
15
16        for (int i = 0; i < cities.length; i++) {
17            City c = new City(String.valueOf(i), (int) (Math.random() * 100), (int) (Math.random() * 100));
18            cities[i] = c;
19            cityCollection.add(c);
20        }
21    }
22
23    GeneticAlgorithm ga = new GeneticAlgorithm(10, 1);
24
25    @Test
26    public void populateCompleteTest() {
27        // Test if all individuals have all cities after initializing the population
28        pop = new Population(5, cities);
29        for (int i = 0; i < 5; i++) {
30            assertEquals(pop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
31        }
32    }
33
34    @Test
35    public void evolveCompleteTest() {
36        // If the individuals in next generation have complete city list
37        pop = new Population(5, cities);
38        ga = new GeneticAlgorithm(5, 0.05);
39        Population newPop = ga.evolve(pop);
40        for (int i = 0; i < 5; i++) {
41            assertEquals(newPop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
42        }
43    }
44
45    @Test
46    public void mutateTest() {
47        // Test if the mutate function works correctly
48        pop = new Population(5, cities);
49        ga = new GeneticAlgorithm(5, 0.05);
50        Population newPop = ga.evolve(pop);
51        for (int i = 0; i < 5; i++) {
52            assertEquals(newPop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
53        }
54    }
55
56    @Test
57    public void crossoverTest() {
58        // Test if the crossover function works correctly
59        pop = new Population(5, cities);
60        ga = new GeneticAlgorithm(5, 0.05);
61        Population newPop = ga.evolve(pop);
62        for (int i = 0; i < 5; i++) {
63            assertEquals(newPop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
64        }
65    }
66
67    @Test
68    public void testFitness() {
69        // Test if the fitness function works correctly
70        pop = new Population(5, cities);
71        ga = new GeneticAlgorithm(5, 0.05);
72        Population newPop = ga.evolve(pop);
73        for (int i = 0; i < 5; i++) {
74            assertEquals(newPop.getIndividuals().get(i).getFitness(), true);
75        }
76    }
77
78    @Test
79    public void seedTest() {
80        // Test if the seed function works correctly
81        pop = new Population(5, cities);
82        ga = new GeneticAlgorithm(5, 0.05);
83        Population newPop = ga.evolve(pop);
84        for (int i = 0; i < 5; i++) {
85            assertEquals(newPop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
86        }
87    }
88
89    @Test
90    public void testDistance() {
91        // Test if the distance function works correctly
92        pop = new Population(5, cities);
93        ga = new GeneticAlgorithm(5, 0.05);
94        Population newPop = ga.evolve(pop);
95        for (int i = 0; i < 5; i++) {
96            assertEquals(newPop.getIndividuals().get(i).getDistance(), true);
97        }
98    }
99
100    @Test
101    public void mutateAltTest() {
102        // Test if the mutateAlt function works correctly
103        pop = new Population(5, cities);
104        ga = new GeneticAlgorithm(5, 0.05);
105        Population newPop = ga.evolve(pop);
106        for (int i = 0; i < 5; i++) {
107            assertEquals(newPop.getIndividuals().get(i).getCities().containsAll(cityCollection), true);
108        }
109    }
110}
```

# Running results

Here is the running result for this project:

```
Generation: 976 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 977 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 978 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 979 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 980 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 981 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 982 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 983 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 984 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 985 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 986 Average Fitness: 0.317 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 987 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 988 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 989 Average Fitness: 0.317 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 990 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 991 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 992 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 993 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 994 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 995 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 996 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 997 Average Fitness: 0.316 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 998 Average Fitness: 0.317 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 999 Average Fitness: 0.317 Best Fitness: 0.318 Shortest Distance: 3147.271
Generation: 1000 Average Fitness: 0.317 Best Fitness: 0.318 Shortest Distance: 3147.271
Path: [3(221,420), 2(133,454), 4(117,464), 23(86,467), 40(63,480), 37(50,488), 16(17,486), 15(60,460), 25(27,443), 32(8,370), 10(79,267), 47(1
```

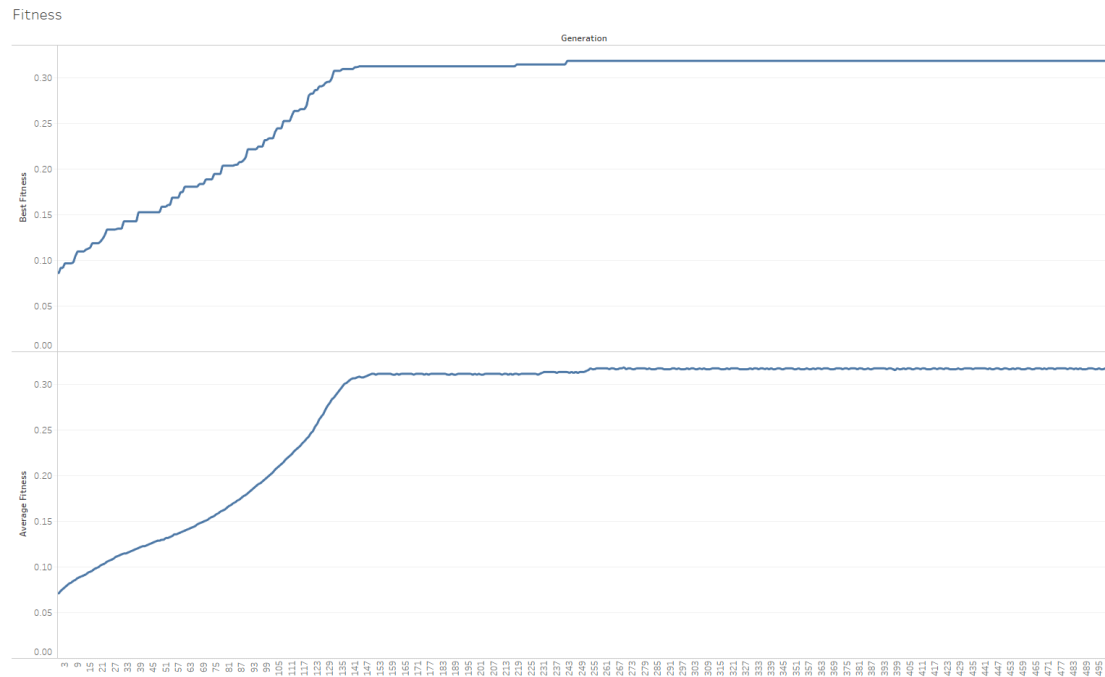
The shortest distance, average fitness and best fitness of each generation will be printed. The best fitness and shortest distance don't always have better result comparing to previous generation because the best half of previous population is still inside the current generation. In this situation, it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

We also generate log file as follow:

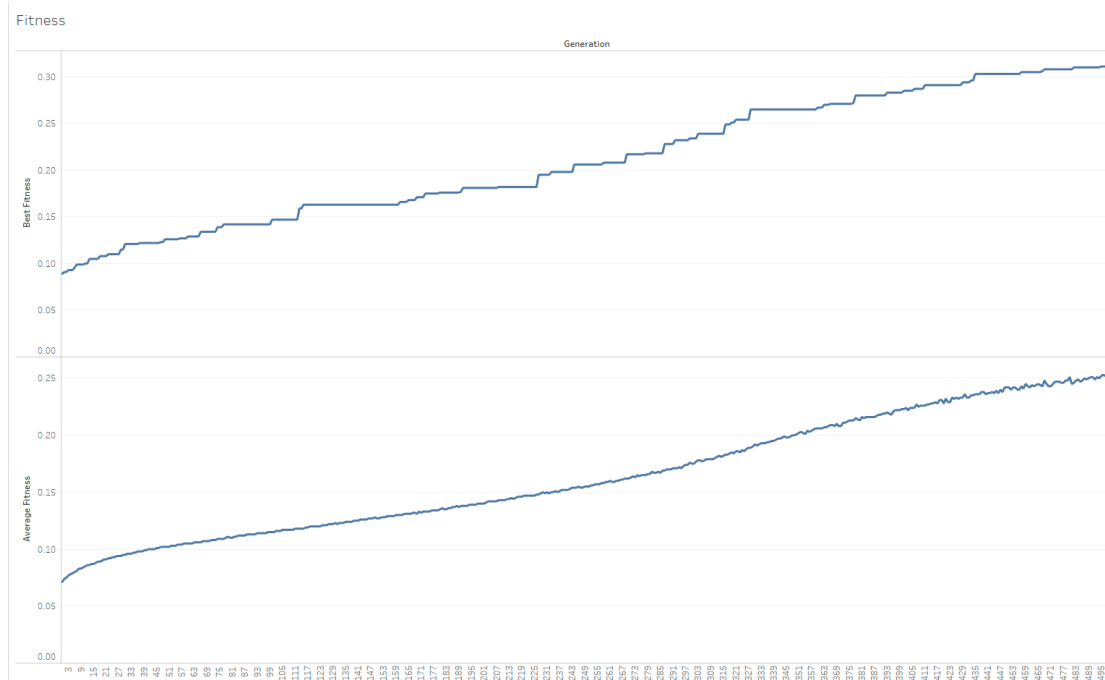
```
candidate Path: [3(221,420), 2(133,454), 4(117,464), 23(86,467), 40(63,480), 37(50,488), 16(17,486), 15(60,460), 25(27,443), 32(8,370), 10(79,267), 47(131,291), 22(142,374), 38(175,388), 36(204,375), 12(266,347), 45(307,380), 31(292,285), 1(204,293), 41(207,259), 18(217,223), 29(259,182), 6(312,107), 27(306,102), 50(322,49), 34(184,56), 42(186,47), 44(146,13), 13(12,8), 35(34,61), 16(20,81), 48(22,93), 11(9,124), 20(35,108), 5(210,128), 8(260,127), 19(370,154), 26(370,112), 39(430,28), 17(475,131), 28(467,226), 30(487,235), 21(494,309), 14(395,329), 7(473,348), 43(495,389), 49(491,403), 24(421,458), 9(310,450), 33(269,399)]
), 0.071, 0.088
1, 0.074, 0.089
2, 0.076, 0.089
3, 0.077, 0.095
4, 0.079, 0.095
5, 0.081, 0.095
6, 0.082, 0.105
7, 0.084, 0.105
8, 0.085, 0.105
9, 0.087, 0.105
10, 0.088, 0.105
11, 0.090, 0.107
12, 0.091, 0.107
13, 0.093, 0.108
14, 0.094, 0.109
15, 0.095, 0.115
16, 0.096, 0.121
17, 0.097, 0.121
18, 0.098, 0.121
19, 0.099, 0.121
20, 0.101, 0.122
21, 0.101, 0.122
22, 0.103, 0.125
23, 0.104, 0.125
24, 0.105, 0.128
25, 0.106, 0.128
26, 0.107, 0.128
27, 0.108, 0.131
```

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 function). 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](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>