TRAVELING SALESMAN PROBLEM

Team 510

Hetashavi Shah

Chitra Paryani

Traveling-Salesman Problem

In Traveling Salesman Problem(TSP), it consist of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain city (e.g. hometown) and return back to the same city.

The challenge in the travelling salesman problem is to minimize the total cost of the tour.

The traveling salesman problem is NP-complete and cannot be solved exactly in polynomial time.

Approach – GENETIC Algorithms

In this approach, we have set of cities and we are measuring distance between the two cities.

Then, we are calculating the total distance between all the cities and finding fitness score of every individual route.

After this, we are sorting the routes based on fitness score and picking top 2 elite routes to pass it to new generation.

Then, evolving new population using crossover and mutation and repeating the steps until we get best possible solution.

Implementation

Below are the main classes which we have created to implement traveling salesman problem

City Class – This class consist of city name, its latitude and longitude, and method to measure distance between two cities.

Route Class – This class consist of list of cities and a method to calculates the total distance between all the cities and check its fitness.

Population – This class consist of list of routes and a method to sort routes by fitness

Genetic Algorithm – In this class we are selecting fittest parents to give to the next generation using crossover and mutation methods.

INFO6205_510 (class containing main() method)

```
private static final Logger log = Logger.getAnonymousLogger();

public ArrayList<City> firstRoute = new ArrayList<City>(Arrays.asList(
        new City("Boston", 42.3601, -71.0589),
        new City("Houston", 29.7604, -95.3698),
        new City("Austin", 30.2672, -97.7431),
        new City("San Francisco", 37.7749, -122.4194),
        new City("Denver", 39.7392, -104.9903),
        new City("Los Angeles", 34.0522, -118.2437),
        new City("Chicago", 41.8781, -87.6298),
        new City("New York", 40.7128, -74.0059),
        new City("Dallas", 32.7767, -96.7970),
        new City("Seattle", 47.6062, -122.3321),
        new City("Sydney", -33.8675, 151.2070),
        new City("Tokyo", 35.6895, 139.6917),
        new City("Cape Town", -33.9249, 18.4241)

));
```

```
public static final double MUTATION = 0.25;
public static final int CROSSOVER = 3;
public static final int POPULATION_SIZE = 8;
public static final int ELITE_ROUTES = 1;
public static final int GENERATIONS =30;
```

Initial Set UP

```
public double measureDistance(City city){

    double deltaLongitude = city.getLongitude() - this.getLongitude();
    double deltaLatitude = city.getLatitude() - this.getLatitude();

    double dist = Math.pow(Math.sin(deltaLatitude/2D), 2D) +
            Math.cos(this.getLatitude()) * Math.cos(city.getLatitude()) *
            Math.pow(Math.sin(deltaLatitude/2D),2D);

    return CONVERT_KM_TO_MILES * EARTH_EQUATORIAL_RADIUS * 2D * Math.atan2(Math.sqrt(dist),Math.sqrt(1D-dist));

}
```

Measuring Distance between two Cities

Calculating Fitness for Each Route

```
public double getFitness(){
    if(isFitnessChanged == true){
        fitness = (1/calculateTotalDistance())*10000;
        isFitnessChanged = false;
    }
    return fitness;
}

public double calculateTotalDistance(){
    int size = this.cities.size();
    return (this.cities.stream().mapToDouble(x -> {
    int cityIndex = this.cities.indexOf(x);
    double returnValue = 0;
    if(cityIndex < size - 1) returnValue = x.measureDistance(this.cities.get(cityIndex + 1));
    return returnValue;
    }).sum() + this.cities.get(0).measureDistance(this.cities.get(size - 1)));
}
```

```
public void sortRoutesByFitness(){
    routes.sort((route1, route2) -> {
        int flag = 0;
        if(route1.getFitness() > route2.getFitness()) flag = -1;
        else if(route1.getFitness() < route2.getFitness()) flag = 1;
        return flag;
    });
}
```

Sorting Routes By Fitness

Selection, crossover, mutate and evolve methods to generate new population

```
Route crossoverRoute(Route route1, Route route2){
    Route crossoverRoute = new Route(this);
    Route tempRoute1 = route1;
    Route tempRoute2 = route2;
    if(Math.random() < 0.5){
        tempRoute1 = route2;
        tempRoute2 = route1;
    }
    for(int x = 0; x < crossoverRoute.getCities().size()/2; x++)
        crossoverRoute.getCities().set(x, tempRoute1.getCities().get(x));
    return fillNullsInCrossoverRoute(crossoverRoute, route2);
}
```

```
Route mutateRoute(Route route){
    route.getCities().stream().filter(x -> Math.random() < MUTATION).forEach(cityX -> {
        int y = (int) (route.getCities().size() * Math.random());
        City cityY = route.getCities().get(y);
        route.getCities().set(route.getCities().indexOf(cityX), cityY);
        route.getCities().set(y, cityX);
    });
    return route;
}
```

```
public Population evolve(Population populate){
    return mutatePopulation(crossoverPopulation(populate));
}
```

Crossover

- Route1: [Denver, San Francisco, Houston, Austin, New York, Boston, Chicago, Los Angeles]
- Route2: [Chicago, Boston, Los Angeles, Austin, San Francisco, Houston, Denver, New York]
- Intermediate crossoverRoute: [Denver, San Francisco, Houston, Austin, New York, null, null, null]
- Final crossoverRoute: [Denver, San Francisco, Houston, Austin, New York, Chicago, Boston, Los Angeles]

Mutation

- Original route: [Austin, Boston, Los Angeles, Denver, Houston, San Francisco, Chicago, New York]
- Mutated route: [Los Angeles, Boston, Austin, Denver, Houston, San Francisco, New York, Chicago]

Elitism

- A basic genetic algorithm will often lose the best individuals in a population between generations because of the crossover and mutation operators. However, we need these operators to find better solutions.
- One simple optimization technique used to tackle this problem is to always allow the fittest individual, or individuals, to be added unaltered to the next generation's population. This way the best individuals are no longer lost from generation to generation.
- This process of retaining the best for the next generation is called *elitism*.

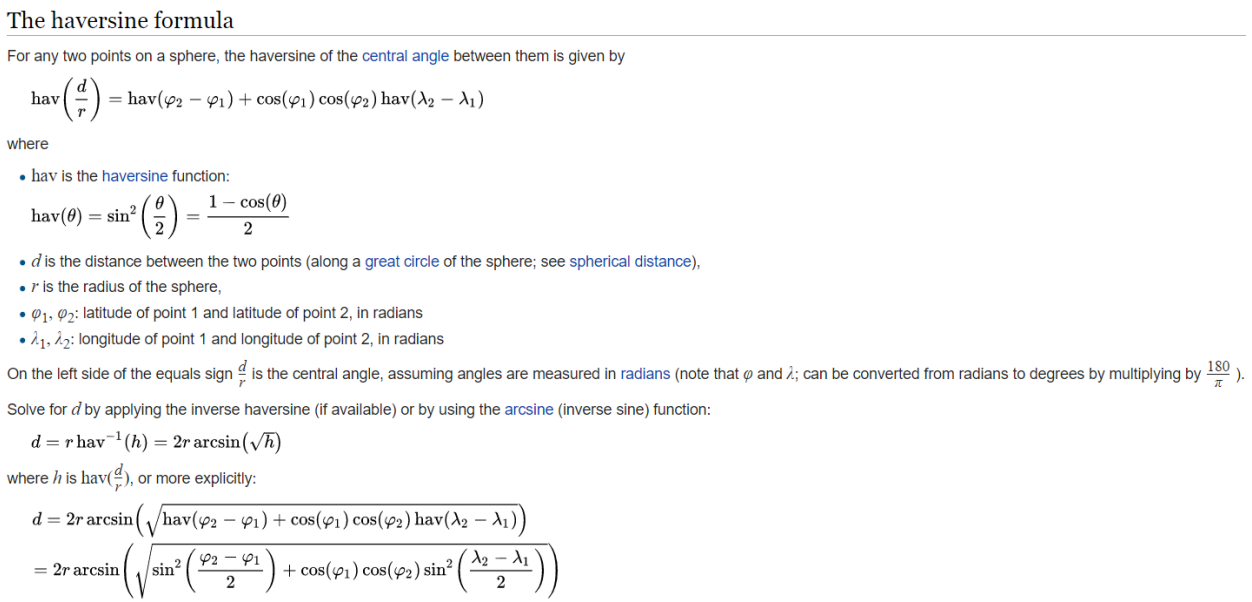
GenoType and PhenoType

- GenoType is the list of cities

- PhenoType is the best 2 cities with maximum fitness score from each of the population

Fitness Function

- In TravelPath class we are calculating the distance and fitness score by using the TotalDistance and FitnessScore methods. DistanceBetweenCities is calculating distance between two cities by using haversine formula



The haversine formula

For any two points on a sphere, the haversine of the central angle between them is given by

$$\operatorname{hav}\left(\frac{d}{r}\right) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

where

- hav is the haversine function:

$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

- d is the distance between the two points (along a great circle of the sphere; see spherical distance),
- r is the radius of the sphere,
- φ_1, φ_2: latitude of point 1 and latitude of point 2, in radians
- λ_1, λ_2: longitude of point 1 and longitude of point 2, in radians

On the left side of the equals sign $\frac{d}{r}$ is the central angle, assuming angles are measured in radians (note that φ and λ; can be converted from radians to degrees by multiplying by $\frac{180}{\pi}$).

Solve for d by applying the inverse haversine (if available) or by using the arcsine (inverse sine) function:

$$d = r\operatorname{hav}^{-1}(h) = 2r\arcsin\left(\sqrt{h}\right)$$

where h is $\operatorname{hav}(\frac{d}{r})$, or more explicitly:

$$d = 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)}\right)$$
$$= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Fitness Function

- We calculate the fitnessScore in FitgetFitnessnessScore method based on the total distance calculated in calculateTotalDistance method. FitnessScore is obtained by using the follow formula.
- fitness = (1/calculateTotalDistance())*10000;
- calculateTotalDistance method is used to calculate Distance between a pair of cities using latitude and longtitude of the cities.

Evolution

- The population is seeded with 5 cities with genes that have been shuffled randomly. For each generation, all individuals sexually reproduce, doubling the population, and each child also experiences mutation methods. I then calculate the fitness function for each individual and write those values to an array. The remaining individuals then reproduce, doubling the population again.
- For each generation, after the culling is complete the number of survivors is logged along with the highest fitness score.

```
IntStream.range(0, ELITE_ROUTES)
        .forEach(x -> crossoverPopulation.getRoutes().set(x, population.getRoutes().get(x)));
IntStream.range(ELITE_ROUTES, crossoverPopulation.getRoutes().size()).forEach(x -> {
    Route route1 = selectPopulation(population).getRoutes().get(0);
    Route route2 = selectPopulation(population).getRoutes().get(0);
    crossoverPopulation.getRoutes().set(x, crossoverRoute(route1, route2));
```

Implemented Elitism to retain best individual and Intstream function to achieve parallel processing

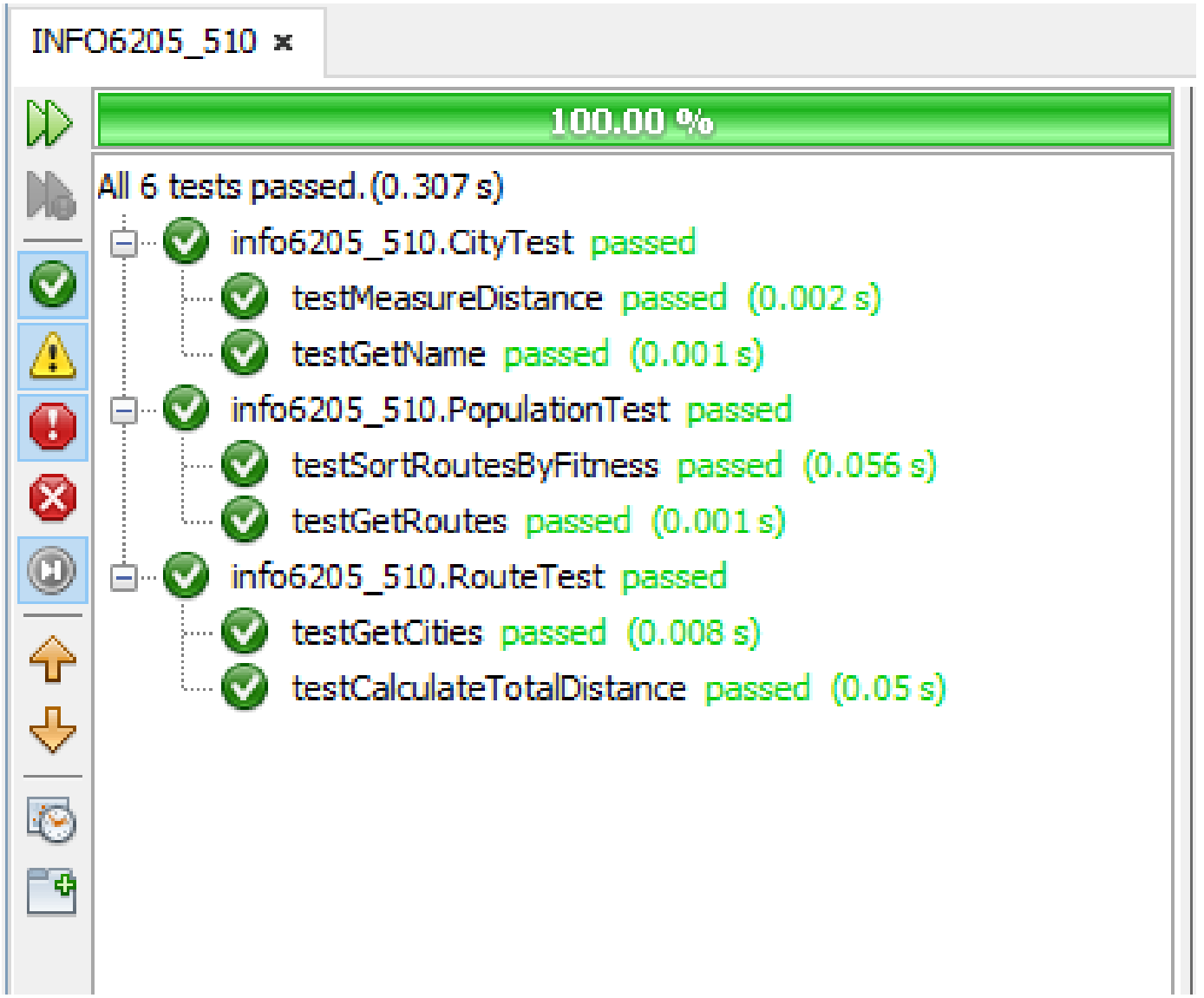
Output Results (Gen#0 d -> 33358.58 and Best Route found in Gen#29 d -> 15693.24)

```
> Generation #0
                                           Route                                              | Fitness | Distance (in miles)
-------------------------------------------------------------------------------------------------------------------------------
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58
[Sydney, Austin, Tokyo, Dallas, New York, Cape Town, Chicago, Houston, San Francisco, Seattle, Los Angeles, Denver, Boston] | 0.2998 | 33358.58


> Generation #29
                                           Route                                              | Fitness | Distance (in miles)
-------------------------------------------------------------------------------------------------------------------------------
[Denver, New York, Seattle, San Francisco, Tokyo, Dallas, Sydney, Cape Town, Houston, Austin, Los Angeles, Boston, Chicago] | 0.6372 | 15693.24
[Denver, New York, Seattle, Dallas, Tokyo, San Francisco, Chicago, Boston, Sydney, Cape Town, Houston, Austin, Los Angeles] | 0.5917 | 16901.02
[Los Angeles, New York, Seattle, San Francisco, Tokyo, Dallas, Sydney, Cape Town, Denver, Austin, Boston, Houston, Chicago] | 0.4872 | 20525.57
[Denver, New York, Seattle, San Francisco, Los Angeles, Sydney, Dallas, Cape Town, Houston, Austin, Tokyo, Boston, Chicago] | 0.3529 | 28339.82
[Denver, Boston, Tokyo, San Francisco, Seattle, Dallas, Sydney, Austin, Houston, Cape Town, Los Angeles, New York, Chicago] | 0.3452 | 28967.90
[Chicago, New York, Austin, Dallas, Tokyo, Sydney, Seattle, San Francisco, Cape Town, Houston, Los Angeles, Boston, Denver] | 0.3241 | 30851.43
[Houston, New York, Seattle, San Francisco, Tokyo, Dallas, Sydney, Denver, Cape Town, Austin, Los Angeles, Boston, Chicago] | 0.3192 | 31329.38
[San Francisco, Houston, Chicago, Austin, Tokyo, Dallas, Seattle, Denver, Boston, Cape Town, New York, Sydney, Los Angeles] | 0.2967 | 33699.50

Best Route so far:[Denver, New York, Seattle, San Francisco, Tokyo, Dallas, Sydney, Cape Town, Houston, Austin, Los Angeles, Boston, Chicago]
w/ a distance of:15693.24miles
BUILD SUCCESSFUL (total time: 0 seconds)
```

Unit Test Results



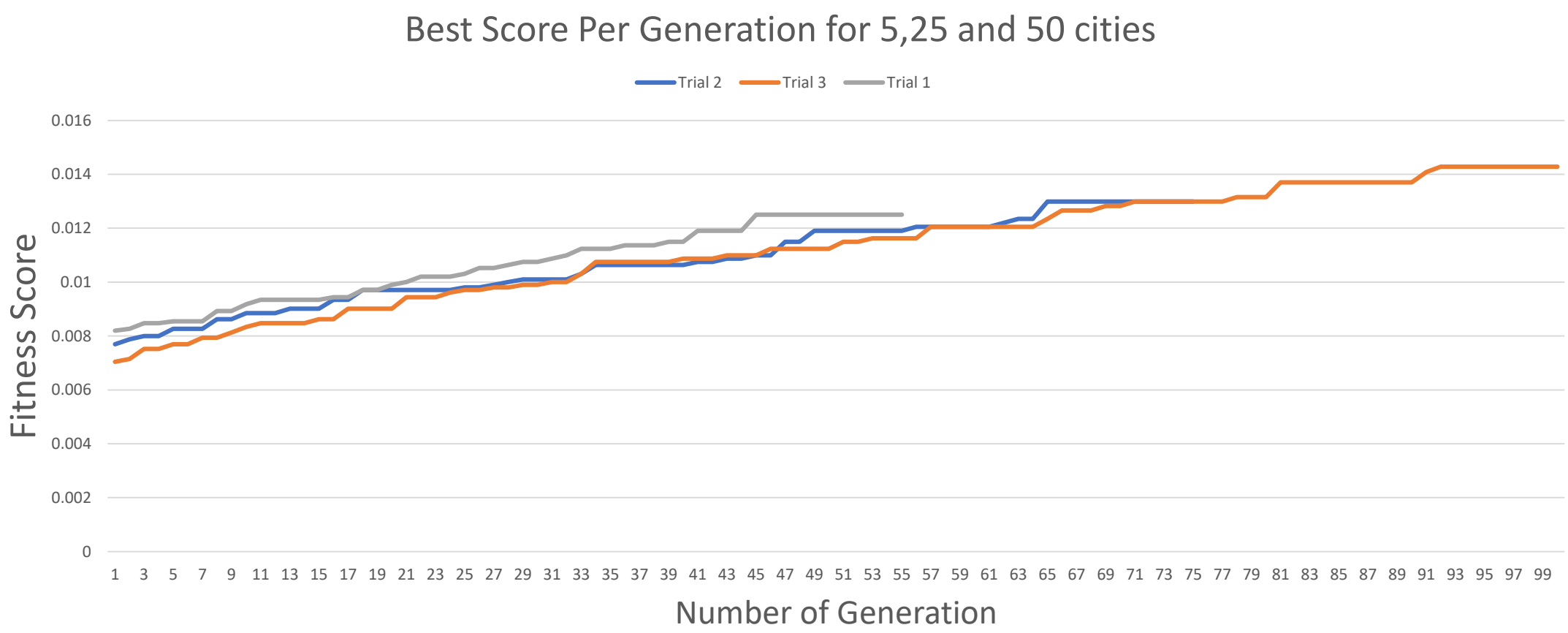
Observations

- Ran the algorithm for different number of cities like 5,25 and 50
- Output from this runs is in log4j
- One trial took a very long time, hence I need to terminate it.
- For the graph with 5 cities, the algorithm produced the same cycle as of
- [New York, Boston, Chicago, Los Angeles, Denver]
- It took 10 generations to reach to the optimal solution
- Since it is a small data of 5 cities, the maximum possible output will be 4!.
- Hence this is the actual solution to the problem for the graph generated.

Analysis

Trial 1	5	25	50
Generations	10	47	49
Final Score	0.1667	0.0286	0.0125
Time	0s	2s	25s
Trial 2			
Generations	19	terminated	69
Final Score	0.1667		0. 0130
Time	0s		5s
Trial 3			
Generations	23	11	95
Final Score	0.1667	0.0196	0.0143
Time	0s	0s	35 m 13 s
Trial 4 - Corrected			
Generations	9	21	31
Final Score	0.1	0.0196	0.0104
Time	1s	0s	5s

Graph


Best Score Per Generation for 5,25 and 50 cities
Trial 2
Trial 3
Trial 1
Fitness Score
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
Number of Generation

Conclusion

- The size of population have effect on runtime and solution.
- More no. of cities too longer to run, as a result of the population size was culled.
- Due to more no. of cities ie genes in the chromosomes there were more opportunities for mutation of crossover.
- There was a positive relationship between population size, no. of generation and fitness score.
- The mutation rate of 0.35 resulted in convergence of a better solution but not optimal solution.
- So, genetic algorithm appear to find good solutions for the travelling salesman problem. However, it depends on the way the problem is encoded and which crossover and mutation rate taken.

Thank you