Genetic Algorithm: The Travelling Salesman Problem

Team Number: 336
Mohao Li 001838229
Jiahua Li 001886557

1. Problem

We try to create a genetic algorithm to solve the travelling salesman problem. Hypothetically, given a set of cities that have different co-ordinates (x, y), the constant departure and the destination cities is same city but not in the cities set, we try to find the shortest path for a route that begins from the departure city, visits each city once in the cities set and returns to the departure city.

In our problem, we create 50 cities co-ordinates as passed cities and 1 city as the departure and destination city, which means that the whole tour contains 52 cities.

And all cities data as following,

All passed cities:

```
City city0 = new City(102, 253);
cityList.add(city0);
City city1 = new City(198, 112);
cityList.add(city1);
City city2 = new City(19, 78);
cityList.add(city2);
City city3 = new City(202, 122);
cityList.add(city3);
City city4 = new City(134, 49);
cityList.add(city4);
City city5 = new City(60, 113);
cityList.add(city5);
City city6 = new City(178, 202);
cityList.add(city6);
City city7 = new City(140, 71);
cityList.add(city7);
City city8 = new City(274, 182);
cityList.add(city8);
City city9 = new City(96, 232);
cityList.add(city9);
City city10 = new City(106, 244);
cityList.add(city10);
City city11 = new City(58, 286);
cityList.add(city11);
City city12 = new City(76, 35);
cityList.add(city12);
City city13 = new City(206, 110);
cityList.add(city13);
City city14 = new City(281, 195);
cityList.add(city14);
City city15 = new City(53, 43);
cityList.add(city15);
City city16 = new City(41, 91);
cityList.add(city16);
City city17 = new City(185, 80);
cityList.add(city17);
City city18 = new City(48, 240);
cityList.add(city18);
City city19 = new City(244, 277);
cityList.add(city19);
City city20 = new City(231, 245);
cityList.add(city20);
City city21 = new City(139, 259);
cityList.add(city21);
City city22 = new City(77, 75);
cityList.add(city22);
City city23 = new City(189, 260);
cityList.add(city23);
City city24 = new City(231, 128);
cityList.add(city24);
City city25 = new City(104, 65);
cityList.add(city25);
City city26 = new City(209, 247);
cityList.add(city26);
City city27 = new City(28, 284);
cityList.add(city27);
City city28 = new City(245, 180);
cityList.add(city28);
City city29 = new City(148, 53);
cityList.add(city29);
City city30 = new City(5, 133);
cityList.add(city30);
City city31 = new City(105, 49);
cityList.add(city31);
City city32 = new City(158, 111);
cityList.add(city32);
City city33 = new City(189, 218);
cityList.add(city33);
City city34 = new City(83, 106);
cityList.add(city34);
City city35 = new City(19, 42);
cityList.add(city35);
City city36 = new City(188, 82);
cityList.add(city36);
City city37 = new City(160, 198);
cityList.add(city37);
```

```
City city38 = new City(290, 6);
cityList.add(city38);
City city39 = new City(10, 265);
cityList.add(city39);
City city40 = new City(66, 161);
cityList.add(city40);
City city41 = new City(194, 128);
cityList.add(city41);
City city42 = new City(47, 166);
cityList.add(city42);
City city43 = new City(120, 173);
cityList.add(city43);
City city44 = new City(100, 116);
cityList.add(city44);
City city45 = new City(161, 107);
cityList.add(city45);
City city46 = new City(19, 29);
cityList.add(city46);
City city47 = new City(13, 65);
cityList.add(city47);
City city48 = new City(52, 115);
cityList.add(city48);
City city49 = new City(254, 63);
cityList.add(city49);
```

The departure and destination city: (90, 150)

2. Implement design

Genetic code: we design a constant length chromosome via ArrayList, and each city stands for one type of genetic code. When we run the mutation or the crossover method, the place genetic codes in the ArrayList would be changed or switch.

Gene expression: if we get a specific chromosome, we could calculate the distance between cities in the chromosome to get the total distance of chromosome by co-ordinates of cities.

Fitness function: we can know that each chromosome has a total distance and calculate the score by $f(d) = 1/d$, where $d = the\ total\ distance\ of\ a\ chromosome$, and this score means that the greater score is, the higher fitness the individual has.

Mutation function: we can get two different random indexes except the first one and the last one in ArrayList, and then, switch two bases of these indexes in the chromosome. For example,
Parent: | **1** | **2** | **3** | **4** | **5** |
Swapped indexes: 2, 4
Offspring: | **1** | **2** | **5** | **4** | **3** |

Crossover function: we can get two different random indexes except the first one and the last one in ArrayList as the start-point and crossover-point, and then, witch these two parts between the start-point and crossover-point in two parent chromosomes.

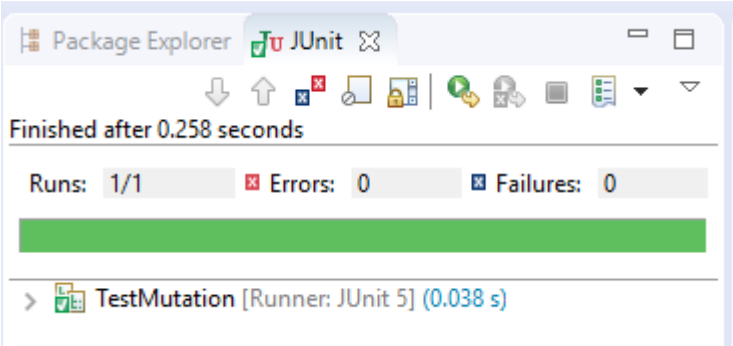
Apparently, some bases would be repeated in chromosomes. We need to delete repeated bases and replace them with bases not in chromosomes.
For example,
Parent 1: | **1** | **2** | **3** | **4** | **5** |
Parent 1: | **4** | **5** | **2** | **1** | **3** |
start-point: 2
crossover-point: 4
Step 1:
Offspring 1: | **1** | **5** | **2** | **1** | **5** |
Offspring 2: | **4** | **2** | **3** | **4** | **3** |
Step 2:
Offspring 1: | **3** | **5** | **2** | **1** | **4** |
Offspring 2: | **5** | **2** | **3** | **4** | **1** |

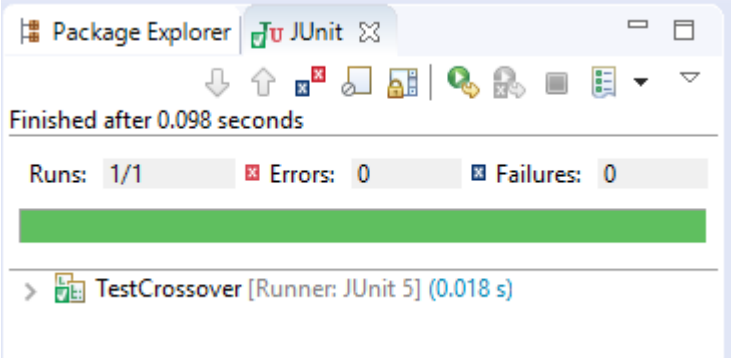
Evolution: all individuals is stored in a population(ArrayList) and is ordered by their fitness. In each generation, only individuals that have good fitness can reproduce, and another part of individuals will die and be replaced with offspring from good fitting individuals. We would like to select two individuals from the good fitting part, let them crossover and each offspring would have chance to mutate. In the end of each generation, the size of population would be constant.

Logging function: in each generation, we can get the most fitting individual, its distance, fitness score and the number of generation it existed. The evolutionary process is ended until the highest fitness score hasn't changed in 20 generations.

3. JUnit Test

We design two junit tests to test mutation function and crossover function.

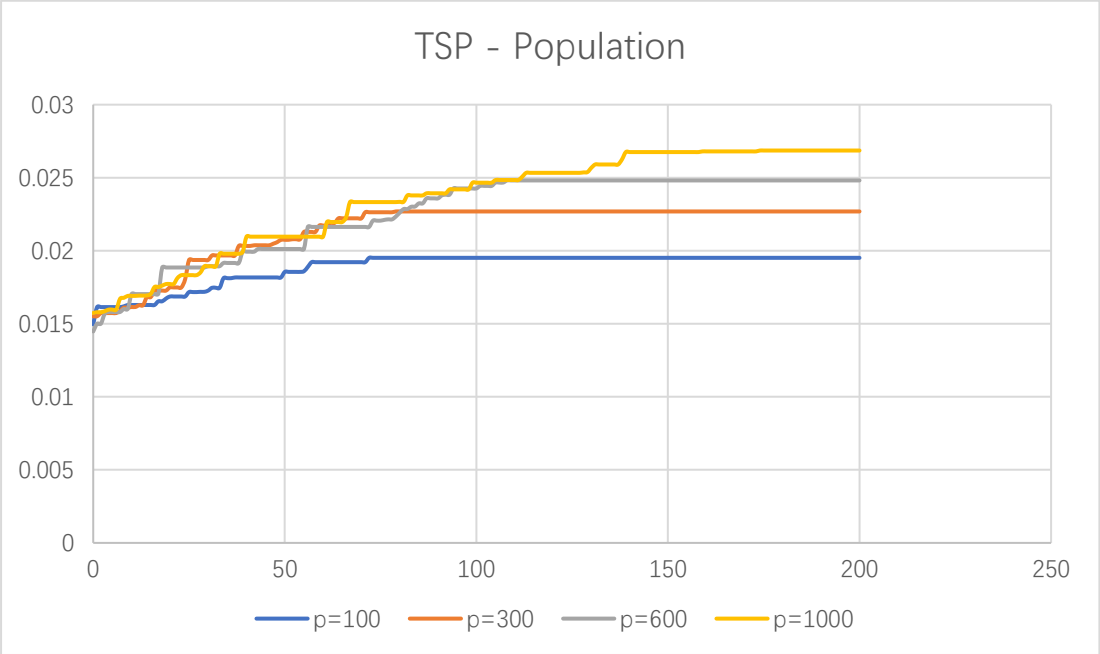




As screenshots show above, our two methods pass tests successfully, which means that they operate properly.

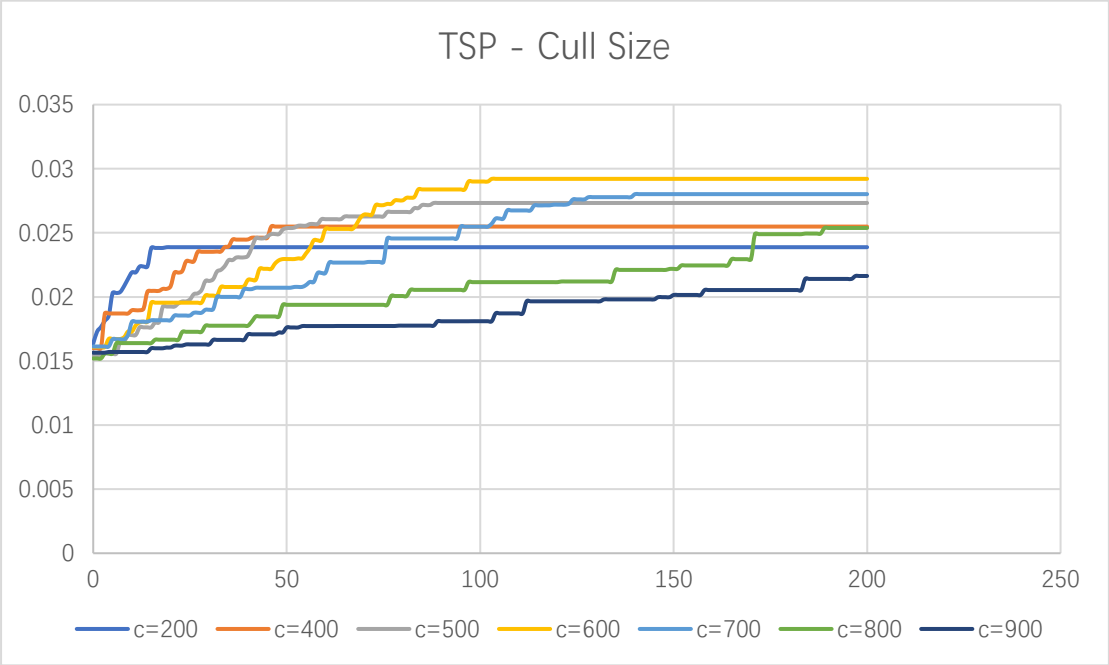
4. Analysis & Conclusion

First, we try to figure out the relationship between the population size and the fitness score. We make the cull size, mutate rate constant, meanwhile we also let the population size as 100, 300, 600 and 1000 per time. The result is showed as following:



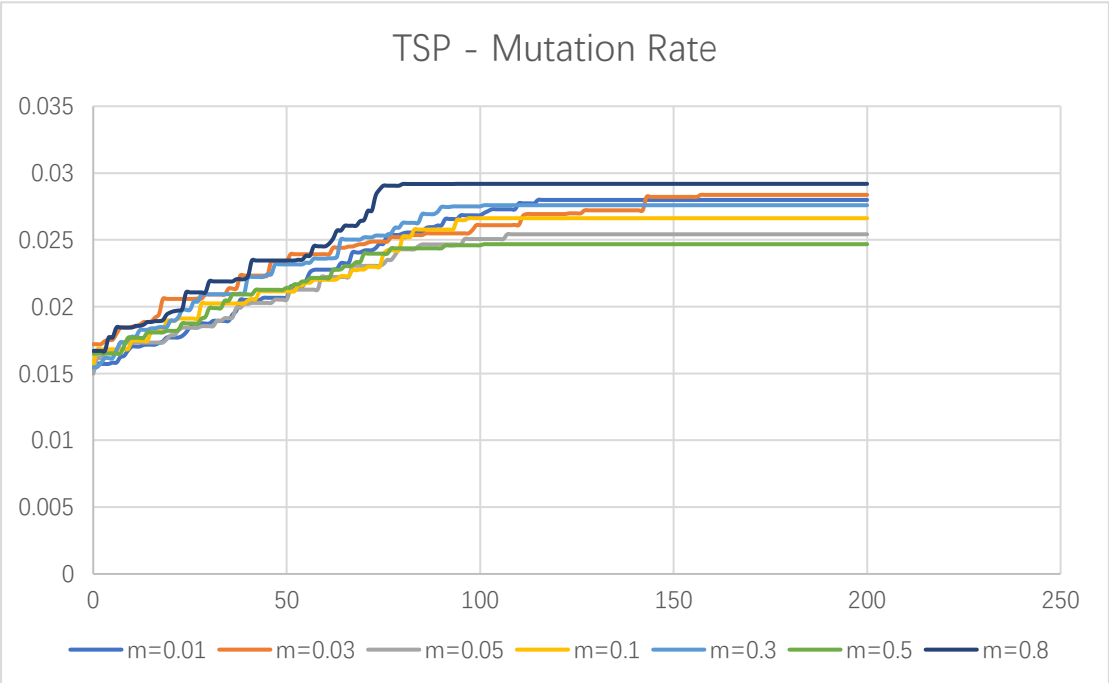
Based on the diagram, we can know that while the population size increases, the fitness score also increases, but the convergence speed of our algorithms also decreases, so we assume that the greater population would be good for our algorithms.

Second, we test the relationship between the cull size and the fitness score. We make the population size, mutate rate constant, meanwhile we also let the cull size as 200, 400, 500, 600, 700, 800 and 900 per time. The result is showed as following:



Based on the diagram, we can see that while the cull size increases, the convergence speed of our algorithms also decreases, but the stable fitness score increases first and decreases late. However, when the cull size is 600, the fitness score is the greatest. Because we set the population size as 1000, we could assume that, when the cull size is 2/3 population size, which means that 2/3 population would die and 1/3 population would keep surviving, we have a higher possibility to get the higher fitness score.

Third, we test the relationship between the mutation rate and the fitness score. We make the population size, cull size constant, meanwhile we also let the mutation rate as 0.01, 0.03, 0.05, 0.1, 0.3, 0.5 and 0.8 per time. The result is showed as following:



The mutation rate just can vary from 0 to 1. Based on the diagram, we can know that 0.03 is the most suitable mutation rate, because when the rate is equal to 0.03, the convergence speed of our algorithms is the lowest and the stable fitness score is the relatively high, which means in this case we have the greatest possibility to get good fitness score.

Based on the analysis before, we assume that if we want to have a better fitness score probably and lower convergence speed of our algorithms, the population size should be greater, the cull size should be 2/3 of population size and the mutation rate should be 0.03.

We set the best combination of population size, cull size and mutation rate as variable in the begin method, then got the best solution we thought as following:

```
Tour order is:
(90, 150) (47, 166)(48, 240)(10, 265)(106, 244)(139, 259)(102, 253)(83,
106)(52, 115)(41, 91)(5, 133)(134, 49)(290, 6)(206, 110)(185, 80)(140,
71)(148, 53)(104, 65)(76, 35)(53, 43)(60, 113)(13, 65)(19, 42)(19, 29)(19,
78)(77, 75)(105, 49)(100, 116)(188, 82)(254, 63)(231, 128)(245, 180)(194,
128)(274, 182)(244, 277)(209, 247)(189, 260)(189, 218)(66, 161)(120,
173)(161, 107)(158, 111)(198, 112)(202, 122)(281, 195)(231, 245)(178,
202)(160, 198)(96, 232)(58, 286)(28, 284)(90, 150)
```

The best solution appeared in the 138 of generation, the distance is 3074 miles and the fitness score is 0.0325. And the first generation's fitness score is about 0.0169. Obviously, after the genetic algorithms, the fitness score increases about 48% and this is the best solution we knew right now.