INFO 6205 Final Project Report

Chao Yuan & Zexin Li
Team 227
Fall 2018

Problem Description

In this era of automation, not all work requires humans to complete. Instead, a variety of robots are able to complete these tasks. In particular, the simple and singular work of watering trees can be done using a watering robot. However, when encountering different terrains and not arranging trees one by one, the robot cannot distinguish how watering is most efficient. And if the robot is set to water one by one, the entire design process will become complicated and the entire robot process needs to be redesigned as the trees change throughout the area. Therefore, this test needs to use algorithms that tell us how the robot behaves throughout the watering process.

In the problem of watering robots, we first need to meet the following conditions:

1. Initial population value is 1000.
2. Proportion of organisms that survive and breed is 0.5.
3. Fecundity of mating is 2.
4. The Generation to reproductive maturity is 1.
5. The Maximum number of generations is 10000.

We design a rule for the entire program. Create a 10X10 garden where trees can be planted at every point in the garden. For each experiment we randomly plant 50 trees in this garden and let a watering robot complete the watering of these trees.

The robot has 6 actions: it moves forward, backwards, left, right and stay and watering. Besides, the robot has 5 grids of vision (feeling 5 grids) which includes forward, backwards, left, right and central. Every Grid have 3 conditions :walls, trees or nothing.

The working pattern is based on these movements, which is written in each watering robot’s gene. Through genetic algorithms, the optimal set of gene sequences is continuously derived.

Model Detail Design

Background Design:

In the process of the investigation, we found that the GA algorithm can effectively solve this problem about robot watering. The GA algorithm is a computational model that simulates the natural evolution of Darwin's biological evolution theory and the biological evolution process of genetic mechanism. It is a method to search for optimal solutions by simulating natural evolutionary processes. A genetic algorithm begins with a population that represents a potential set of solutions to a problem, while a population consists of a certain number of individuals encoded by genes. In each generation of this population, individuals are selected according to the fitness of the individual in the problem domain, and crossovers and mutations are performed by means of genetic operators of natural genetics. This process will lead to a population of natural evolution like the descendant population is more adaptable to the environment than the previous generation. The best individual in the last generation population is decoded, which can be used as a problem to approximate the optimal solution.

Environment Design:

Firstly, we design a 10 by 10 garden. In a total of 100 grids, we randomly selected 50 grids to plant the trees. Therefore, we produces two states in each grid: with a tree or no tree. Then, since we stipulate that the robot can't walk out of the garden, we need to expand a circle outside the garden as a wall and inhibit the robot to walk outside. Therefore, each grid uses a total of three states: with a tree, no tree and walls.

Fitness Function Design:

We designed a standard rule to determine whether each robot performs well. Thus, we introduced a scoring table to quantify the performance of the robot.
If the robot hits the wall of the garden, minus 5 points.
Watering at the right place, add 10 points.

Watering the wrong place, minus 1 point.
Watering trees that have been watered, minus 1 point.
the optimal condition is that the robot can get 500 points in the watering process of the 50 trees (no watering in the wrong place once, watering all trees just once and not hitting the wall).

Robot Design:

We mainly designed the robot's action in these grids. There are a total of six actions: go forward, go backward, go left, go right, stay and water tree. In the program, we use 0-5 these int integers to represent these 6 actions.

Gene Design:

The vision problem of the robot has been mentioned before. We stipulate that the watering robot has a perception of the five surrounding grids. These five grids create a micro-environment that is suitable for robots in arbitrary grids. Since each grid has three states: with a tree, no tree and walls, therefore, the surrounding environment in which a robot is located can be quantified into 3^5 cases. These 243 cases cover all possibilities. Each situation is so unique that we can think of these 243 cases as 243 genes on a chromosome. In this program, we use an array of 243 elements to represent the chromosome. The 243 cases are the index of this array element. For each case, the robot will have different actions, which are the values of each element. Therefore, a set of gene sequences is a set of actions for robots. Different genes have different actions in different situations.

Experiment Process

1) Regarding the conditions that the course requirements meet, the Individual in the Initial population represents the 200-step choice for each group of robots. This means that after the robot completes a 200-step watering, it is recorded as an individual. The population is a "population" obtained after robot walks for 1000 times.
2) Then, randomly select 10 groups from population, and select the best fitness from 10 groups. After that, re-select 10 groups from 1000 populations and pick a best fitness, let the two groups of individual crossover or mutation. In this experiment, the probability of crossover is 50%, and the probability of mutation is 0.1%. After this process is

completed, a new individual will be generated and loaded into a new population called second generation. Repeat the above process until the population in this generation meets 1000.

3) The second generation, after storing 1000 individuals, performs the same operation as before to select the third generation. Since the robot's full score is 500 points (mentioned in #3), we set the whole process to stop when the robot's score reaches 400 or the generation exceeds 10,000.

4) In each output, The best represents the best fitness that appears in the 1000 group.

5) In each generation, there will be an Average fitness to compare with best fitness.

6) In the output, we designed to display a detailed value interval for 5 generations. We will present each generation of Average fitness and best fitness in order to refer to the optimization of the entire process.

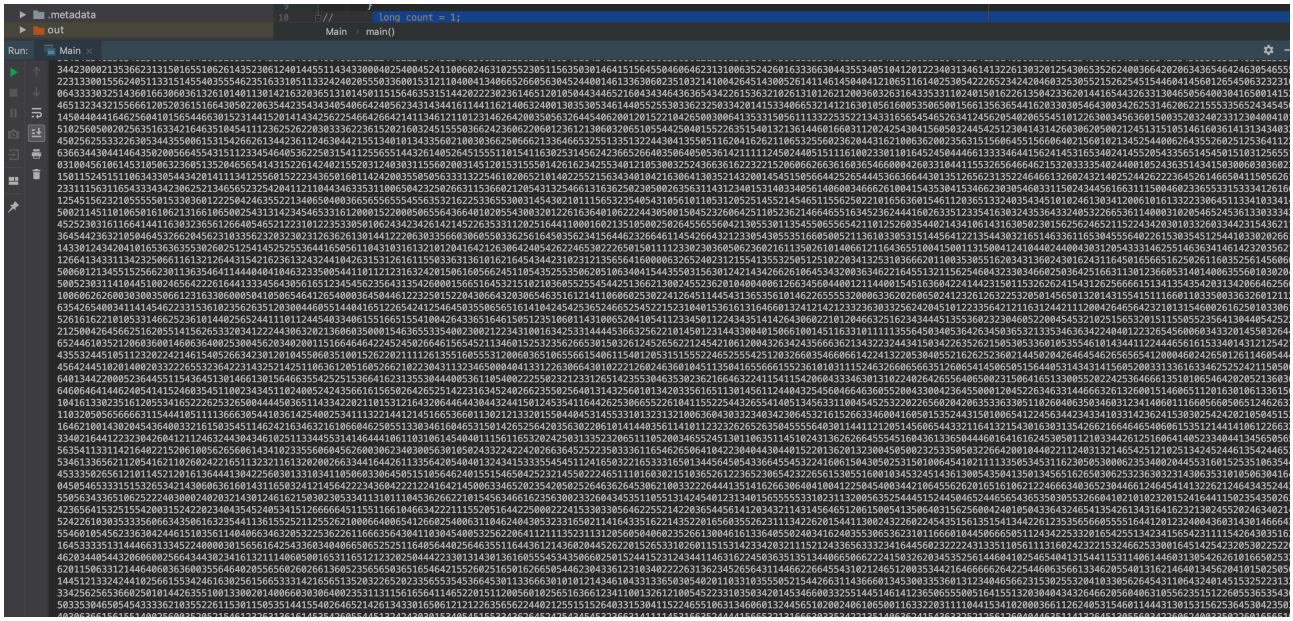
Implementation Details Description

#1 We use the six different actions that the robot makes in each case to represent each gene. The sequence of all 243 cases constitutes a chromosome. There are 6^243 different possible "alleles".

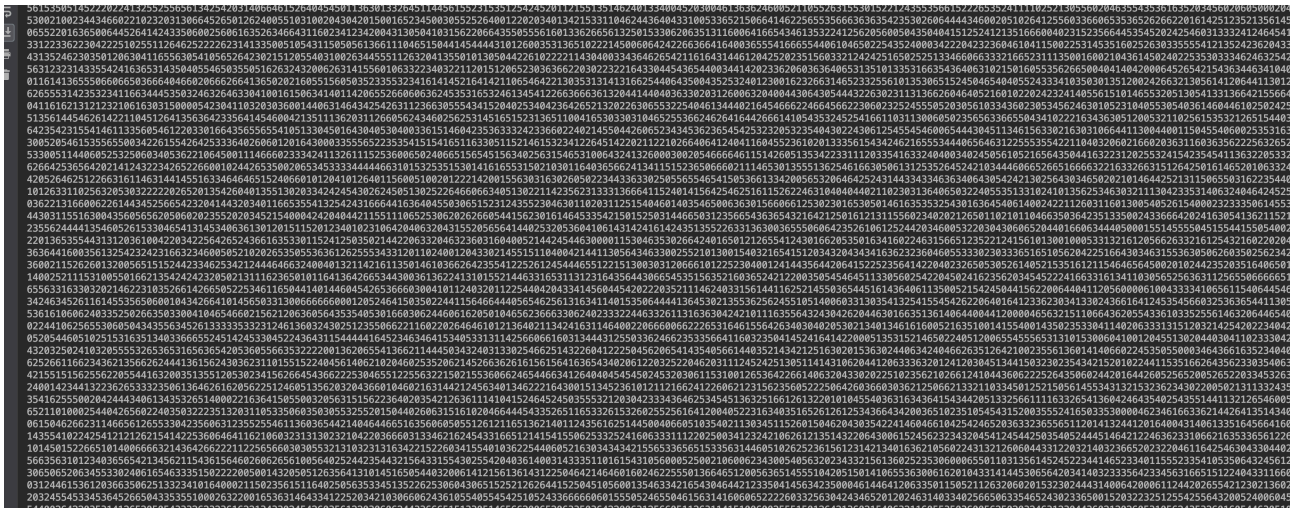
```
public void generateChromos(){
    for (int i = 0; i < length; i++){
        int gene = (int) (Math.random() * action);
        chromos[i] = gene;
    }
}
```

#2 Gene expression：As mentioned above, we use number 0 to number 5, to represent 6 actions. Also, a set of 243 actions is the gene expression for this experiment. 1000 sets arrays form a generation of Gene expression.

The first generation genes:



The last generation genes:

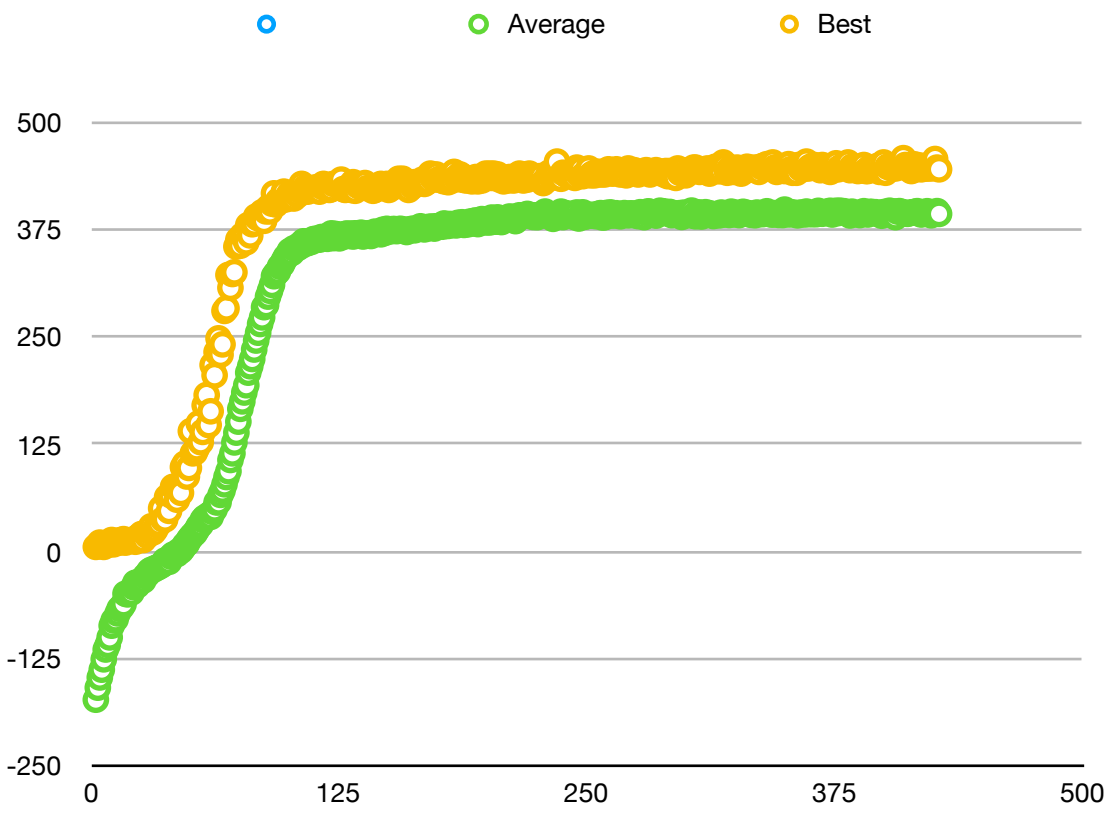


#3The fitness function: As mentioned above, our fitness decision is based on the scoring table. So, the implementation of the fitness function is to score the action in different situations.

```
private static int action(Point point, Map map, int actionNumber){
    int score = 0;
    switch (actionNumber){
        case 0:
            if (map.isIntheMap(point.x, y: point.y-1)){
                score = 0;
                point.y = point.y-1;
            }else{
                score = -5;
            }
            break;
        case 1:
            if (map.isIntheMap( x: point.x-1, point.y)){
                score = 0;
                point.x = point.x -1;
            }else {
                score = -5;
            }
            break;
        case 2:
            if (map.isIntheMap(point.x,  y: point.y + 1)) {
                score = 0;
                point.y = point.y + 1;
            } else {
                score = -5;
            }
            break;
        case 3:
            if (map.isIntheMap( x: point.x + 1, point.y)) {
                score = 0;
                point.x = point.x + 1;
            } else {
                score = -5;
            }
            break;
        case 4:
            if (map.watering(point.x, point.y)) {
                score = 10;
            } else {
                score = -1;
            }
            break;
        case 5:
            score = 0;
            break;
    }
    return score;
}
```

Results & Conclusion

According to the experiment's conditions, we set the population capacity to 1000, the probability of crossover to 0.5, and the probability of mutate to 0.001. We set the termination condition of the experiment to reach 400 for average fitness or 10000 for generation. After running the program to the 428th generation, the result reached 400 points. The figure below shows a scatter plot of 428 sets of average fitness and best fitness data.



As can be seen from the figure, in the initial stage of the program, the genes are continuously optimized while the optimization speed runs fast. It takes about 100 generations to reach 350 points. However, after reaching 350 points, it reached 400

points in about 300 generations. In conclusion, the rate of gene optimization was significantly slowed down.

Conclusion: Genetic algorithm is able to quickly extract favorable conditions from complex problems, and the optimization effect is obvious in the early stage. The genetic algorithm can quickly obtain the basic feasible solution. However, because the excellent genes are gradually saturated, the optimization efficiency drops sharply and it will take a long time to reach the optimal solution.

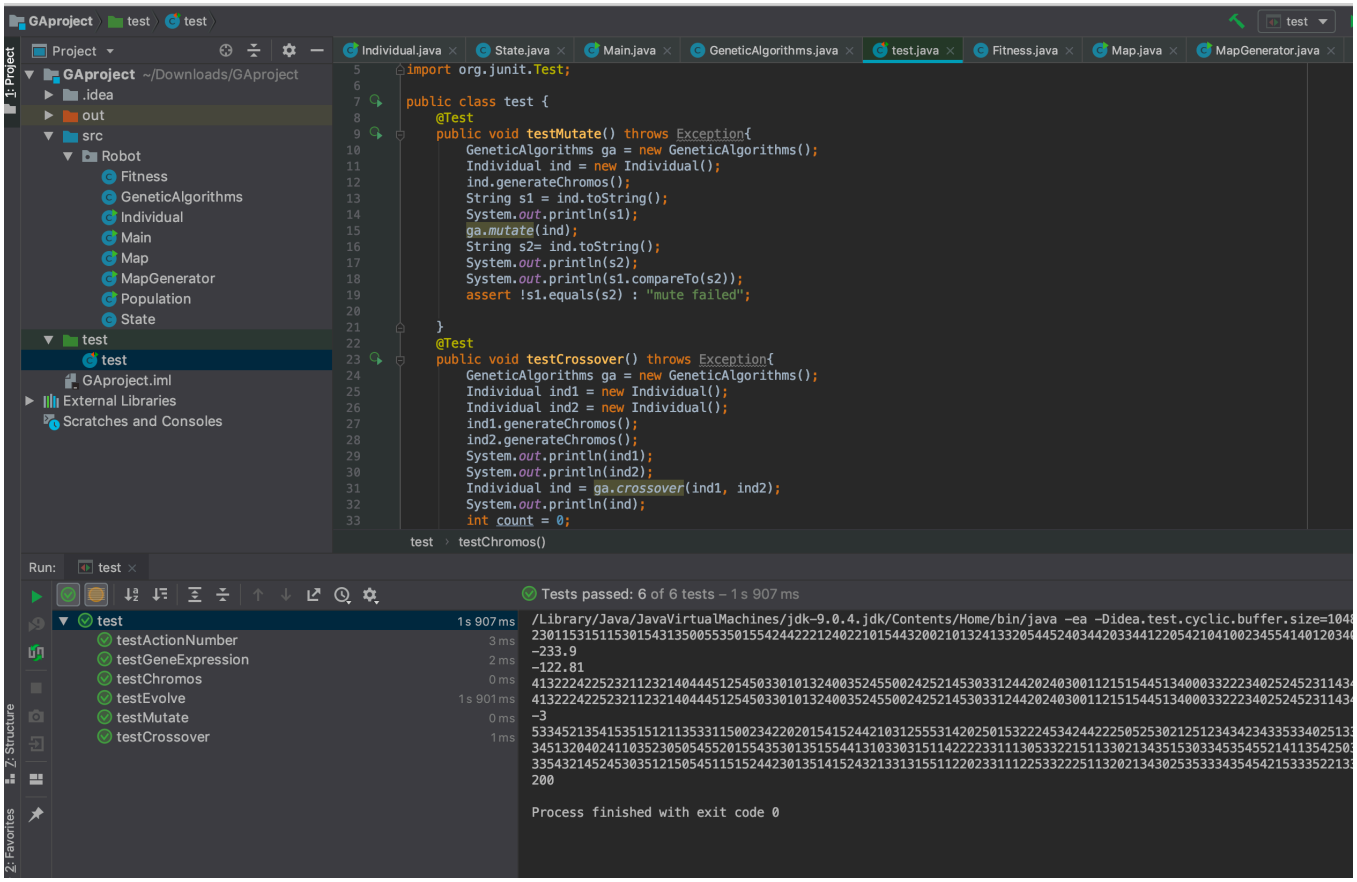
Evidence of Running

Output

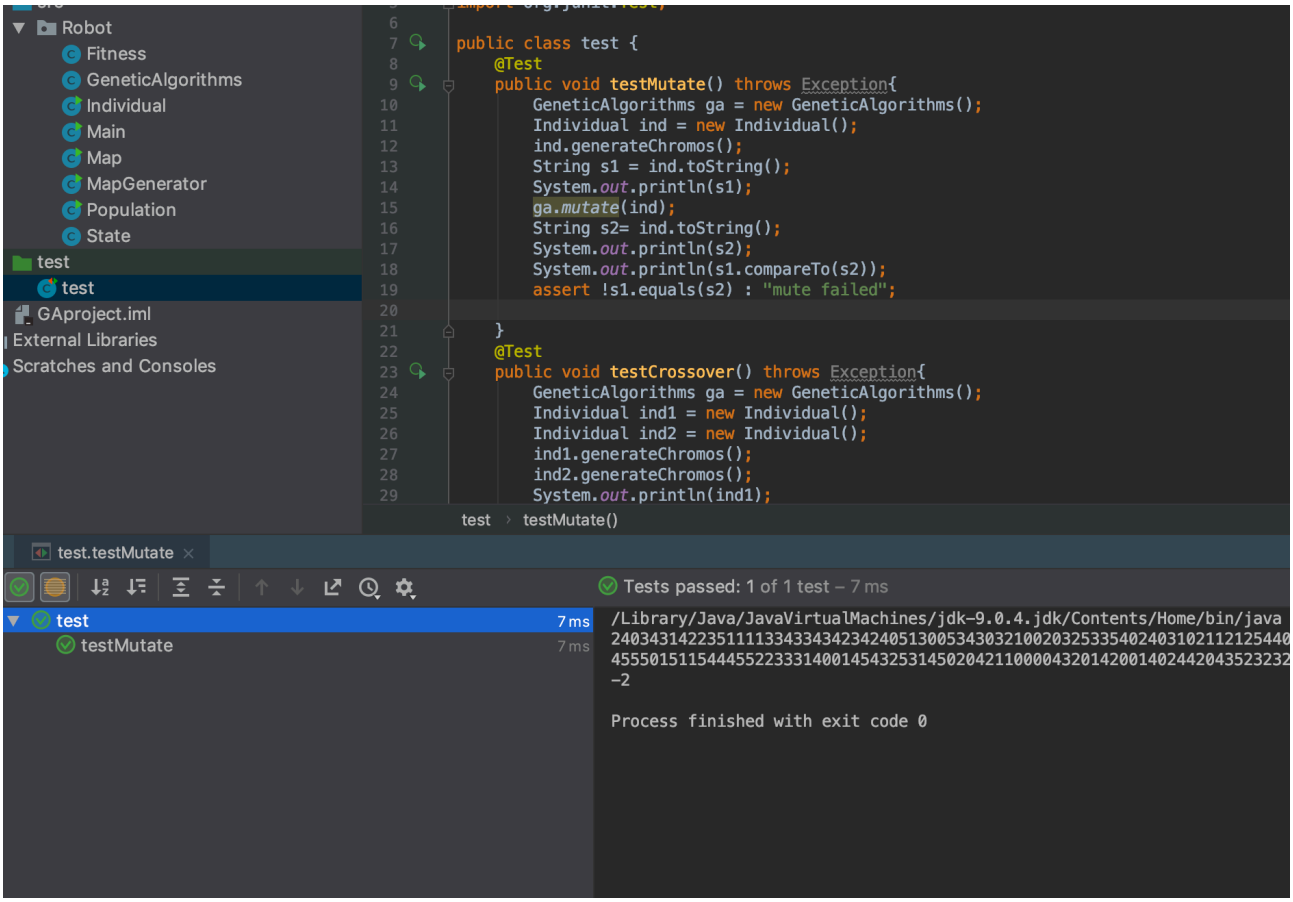
```
Population size: 1000
the Best fitness: 6
the Worst Fitness: -607
Average Fitness: -179.876

----------++++ -177.422 +++++----------
----------++++ -166.499 +++++----------
----------++++ -159.739 +++++----------
----------++++ -142.821 +++++----------
The 5 generations
Population size: 1000
the Best fitness: 7
the Worst Fitness: -494
Average Fitness: -133.165

----------++++ -133.165 +++++----------
----------++++ -128.553 +++++----------
----------++++ -115.947 +++++----------
----------++++ -104.885 +++++----------
----------++++ -96.683 +++++----------
The 10 generations
Population size: 1000
the Best fitness: 9
the Worst Fitness: -530
Average Fitness: -88.081

----------++++ -88.081 +++++----------
----------++++ -85.951 +++++----------
----------++++ -73.136 +++++----------
----------++++ -74.211 +++++----------
----------++++ -67.497 +++++----------
The 15 generations
Population size: 1000
the Best fitness: 7
the Worst Fitness: -358
Average Fitness: -61.332

----------++++ -61.332 +++++----------
----------++++ -56.684 +++++----------
----------++++ -55.949 +++++----------
----------++++ -50.732 +++++----------
----------++++ -44.959 +++++----------
The 20 generations
Population size: 1000
the Best fitness: 10
the Worst Fitness: -392
Average Fitness: -43.023

----------++++ -43.023 +++++----------
----------++++ -40.16 +++++----------
----------++++ -33.513 +++++----------
----------++++ -34.362 +++++----------
----------++++ -25.881 +++++----------
The 25 generations
Population size: 1000
the Best fitness: 17
the Worst Fitness: -298
Average Fitness: -26.748

----------++++ -26.748 +++++----------
```

```
Average Fitness: 397.538

----------++++ 397.538 +++++----------
----------++++ 398.126 +++++----------
----------++++ 397.921 +++++----------
----------++++ 397.49 +++++----------
----------++++ 397.982 +++++----------
The 1150 generations
Population size: 1000
the Best fitness: 437
the Worst Fitness: 241
Average Fitness: 398.052

----------++++ 398.052 +++++----------
----------++++ 396.521 +++++----------
----------++++ 397.454 +++++----------
----------++++ 396.373 +++++----------
----------++++ 397.715 +++++----------
The 1155 generations
Population size: 1000
the Best fitness: 436
the Worst Fitness: -344
Average Fitness: 396.611

----------++++ 396.611 +++++----------
----------++++ 397.627 +++++----------
----------++++ 396.928 +++++----------
----------++++ 398.45 +++++----------
----------++++ 397.772 +++++----------
The 1160 generations
Population size: 1000
the Best fitness: 439
the Worst Fitness: 261
Average Fitness: 398.025

----------++++ 398.025 +++++----------
----------++++ 398.464 +++++----------
----------++++ 397.404 +++++----------
----------++++ 397.194 +++++----------
----------++++ 396.704 +++++----------
The 1165 generations
Population size: 1000
the Best fitness: 434
the Worst Fitness: 174
Average Fitness: 398.11

----------++++ 398.11 +++++----------
----------++++ 397.232 +++++----------
----------++++ 397.868 +++++----------
----------++++ 398.347 +++++----------
----------++++ 398.162 +++++----------
The 1170 generations
Population size: 1000
the Best fitness: 447
the Worst Fitness: 125
Average Fitness: 398.588

----------++++ 398.588 +++++----------
----------++++ 399.812 +++++----------
```

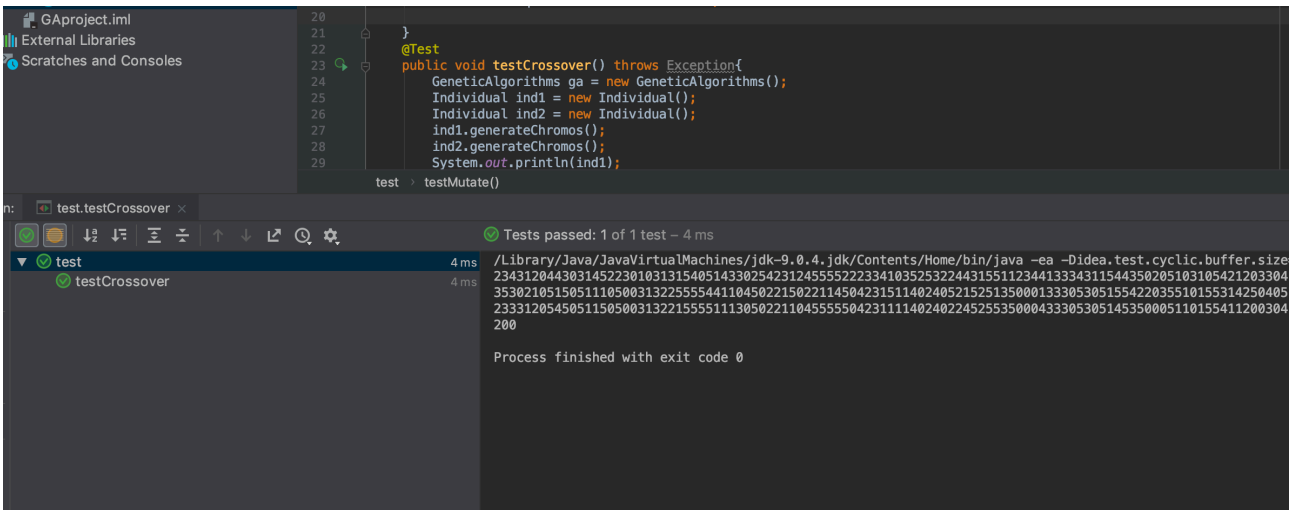
Unit Test


GAproject
test
test
Project
GAproject ~/Downloads/GAproject
.idea
out
src
Robot
Fitness
GeneticAlgorithms
Individual
Main
Map
MapGenerator
Population
State
test
test
GAproject.iml
External Libraries
Scratches and Consoles
Individual.java
State.java
Main.java
GeneticAlgorithms.java
test.java
Fitness.java
Map.java
MapGenerator.java
import org.junit.Test;
public class test {
@Test
public void testMutate() throws Exception{
GeneticAlgorithms ga = new GeneticAlgorithms();
Individual ind = new Individual();
ind.generateChromos();
String s1 = ind.toString();
System.out.println(s1);
ga.mutate(ind);
String s2= ind.toString();
System.out.println(s2);
System.out.println(s1.compareTo(s2));
assert !s1.equals(s2) : "mute failed";
}
@Test
public void testCrossover() throws Exception{
GeneticAlgorithms ga = new GeneticAlgorithms();
Individual ind1 = new Individual();
Individual ind2 = new Individual();
ind1.generateChromos();
ind2.generateChromos();
System.out.println(ind1);
System.out.println(ind2);
Individual ind = ga.crossover(ind1, ind2);
System.out.println(ind);
int count = 0;
test › testChromos()
Run:
test
Tests passed: 6 of 6 tests – 1 s 907 ms
test 1 s 907 ms
testActionNumber 3 ms
testGeneExpression 2 ms
testChromos 0 ms
testEvolve 1 s 901 ms
testMutate 0 ms
testCrossover 1 ms
/Library/Java/JavaVirtualMachines/jdk-9.0.4.jdk/Contents/Home/bin/java -ea -Didea.test.cyclic.buffer.size=1048
-233.9
-122.81
-3
200
Process finished with exit code 0
1: Project
Z: Structure
2: Favorites

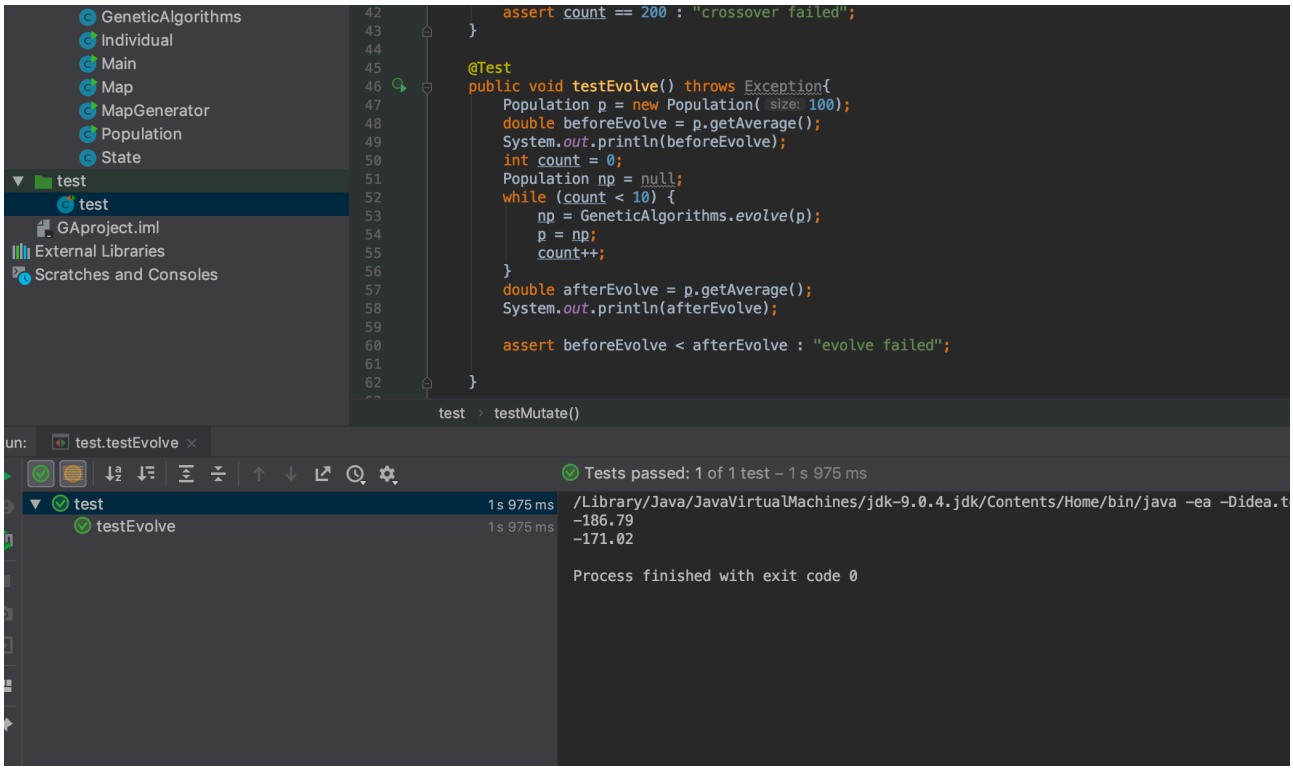
Mutate test:



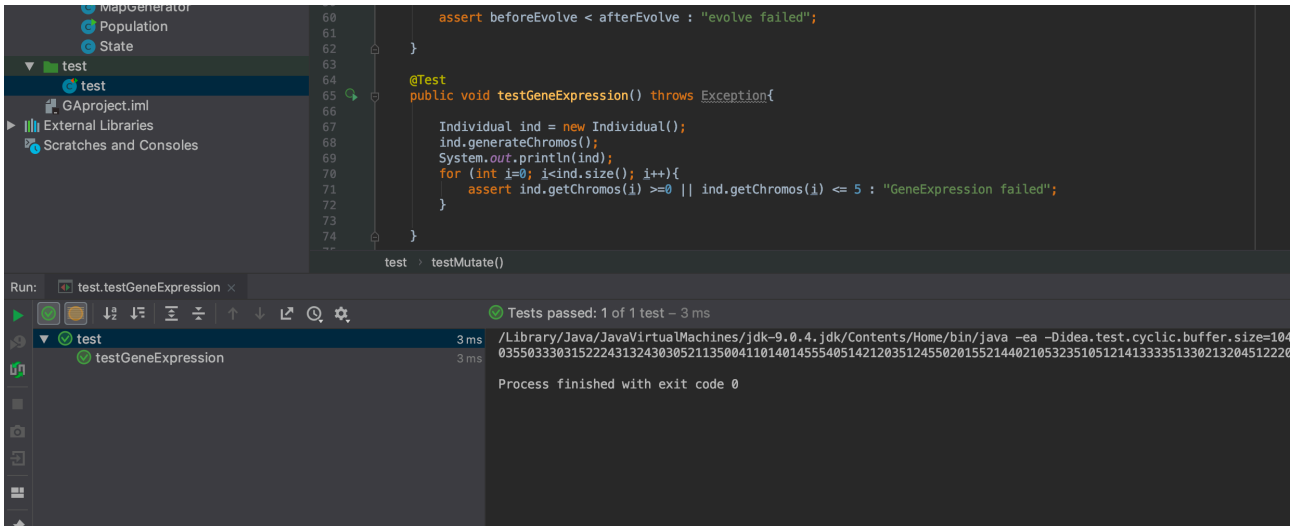
Crossover test:



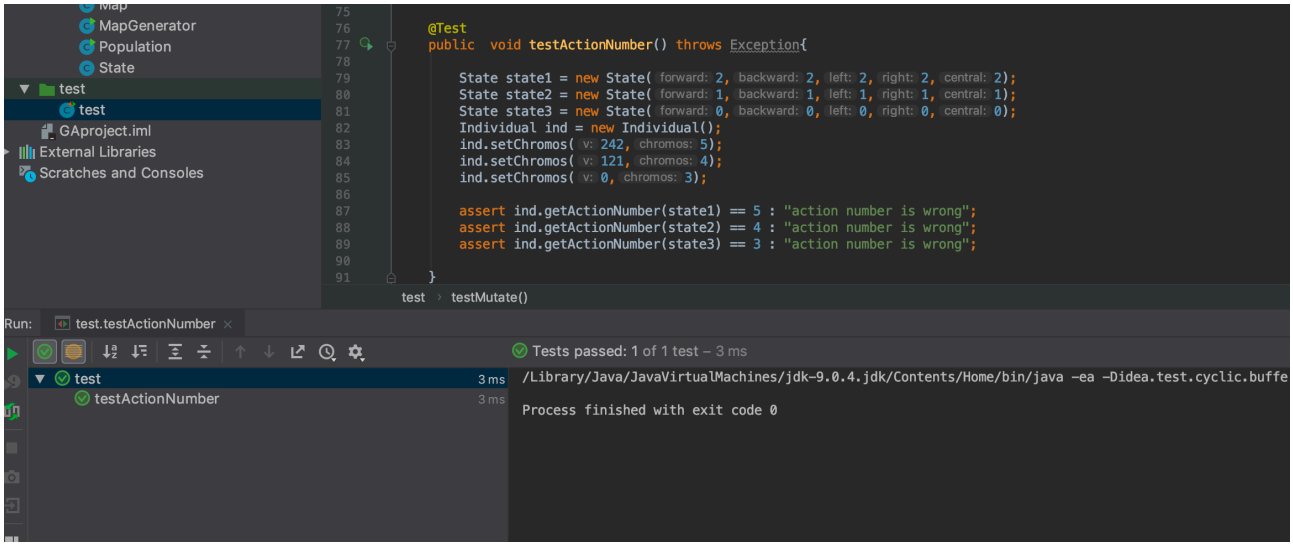
Evolve test:



Gene expression test:



Get action number test:



Get&set chromes test: