
# INFO 6205

# Program Structure and Algorithm

# Final Project

## *Genetic Algorithms for Black-and-White Image*

Team: *221*

Members: *Kaiyuan Zhao*

*Kaichun Wu*

*Xiaobin Gao*

Github: *https://github.com/Xiaobin-Gao/INFO6205_221*

# Contents

## Problem Statement

By using Genetic Algorithms, we generate new black-and-white images by evolving random black pixels on a white background, and after several generations, we select an image that looks like the target image the most.

## Key Words

### Phenotype:

The set of properties of an organism which in some way interact with the environment

### Genotype:

The set of replicable and heritable information.

### Fitness:

Inverse of distance traveled.

### Mutation:

To mutate in order to keep genetic diversity.

### Cross Over:

Like Sexual reproduction, crossover the father and mother to get offsprings.

### Expression:

The “mapping” of genotype to phenotype.

## Design Steps

1. Search all pixels of an input image and save all black dots' location information as the target.
2. For saving memory and time, we compress the picture into a smaller one.
3. To initialize population for the first generation, we randomly generated 1000 points to set as individuals.
4. Then calculate the total distances of every points in an individual to the target.
5. Use multiplication of inverse for the distance to calculate the fitness, then sum all individuals' fitness as all individuals's total fitness in a generation and then calculate the weight of every individual's fitness relative to the total fitness value.
6. We make 50% possibility for the survival.
7. Use the fitness weighted value to select two individuals and then select the individual with better fitness as father, as well as for the mother. One pair of father and mother will generation 2 children.
8. Crossover two genes and we have a probability for them to mutate. The method for crossover is randomly choosing the byte, which means gene, in father or mother and pass it to the child. Use multi-thread for faster speed.
9. Generate the next generation and their genes. The next generation will have a 1% possibly to reduce one individual for the protecting of diversity.
10. If the average weight of the fitness is very close to the best weight of the fitness, we make a big mutation for each individual.
11. Repeat all these steps for 5000 times.
12. Get the result and draw the image every 100 generations, log for each generation.

## Implementation Details

### 1. Implementation of Gene Expression

Usually we use two variables x and y to represent a point in the 2D plane. So we can create an object for point that have two member variables, x and y. This is just like the phenotype in the nature. By doing so, we can use the point to better build the image.

To uniquely identify a point, we can think just how to identity a num. In JAVA, we use 32 bits to represent an Integer. So, if x and y in a point are all int type, we can use an array of 4 bytes to represent x and another one to represent y. That is to say, an array of 8 bytes can represent a point. This is the way we use for gene expression. The array of bytes is just like genotype and the Object Point is just like phenotype.

### 2. Implementation of Fitness

Since we have only two kinds of points, we can use the black point to identity a picture if the background is white. To consider two images as the same, all the black points in an image should be the same as the other. The simplest way to determine whether a point is closer to another is judge the distance between them. When the distance becomes 0, the two points are the same. So to calculate the fitness, just calculate all points distance to corresponding points in the target. The smaller the distance is, the better the image is. Therefore,  the fitness should be reciprocal of the distance.

### 3. Implementation of Survive

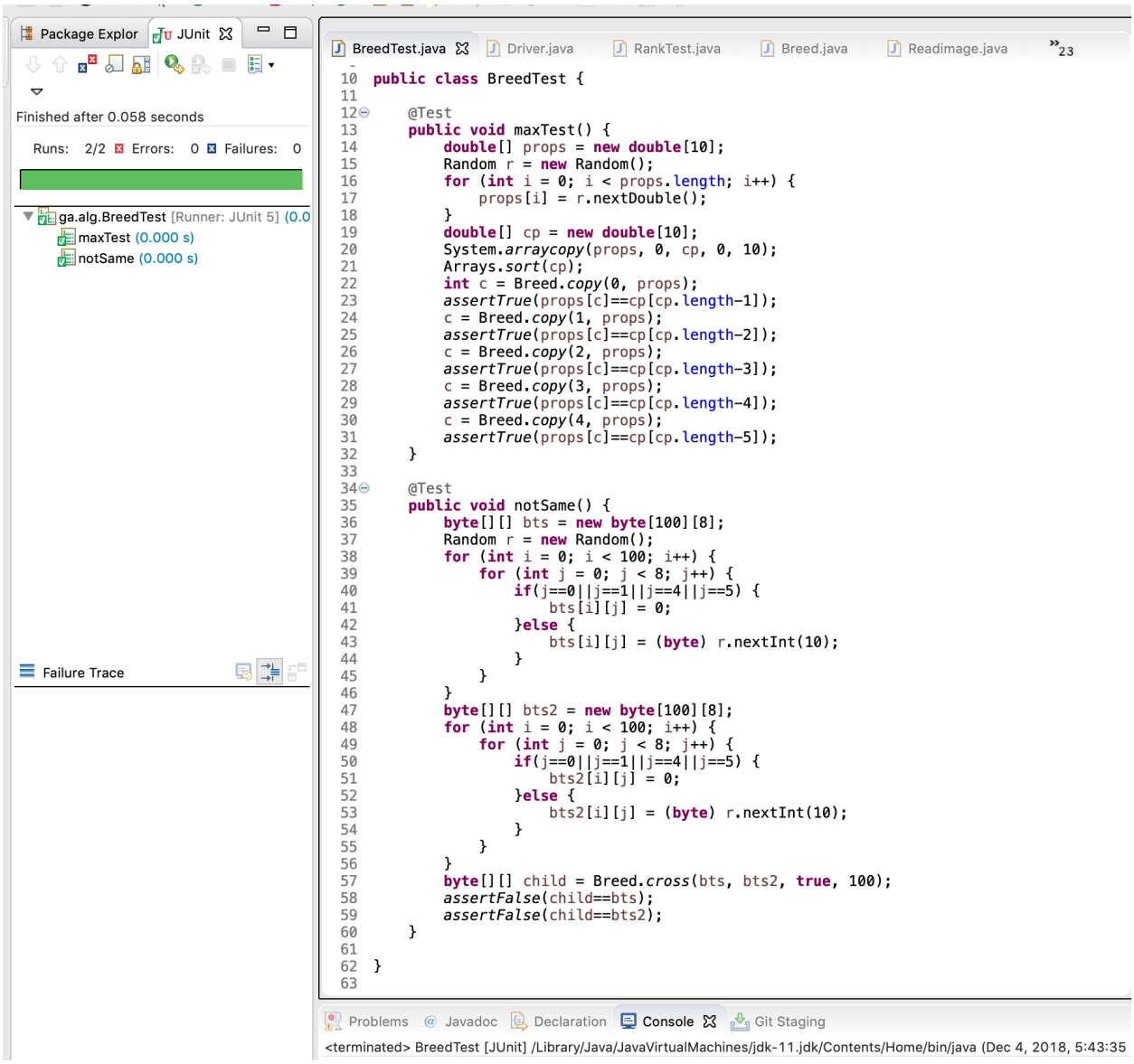
Whether an individual wii survive and successfully breed is determined by the weight of its fitness to the total fitness of the population. Only half population in a generation with bigger fitness-weight value will survive to the next generation.

## Supporting Materials

## 1.  Output

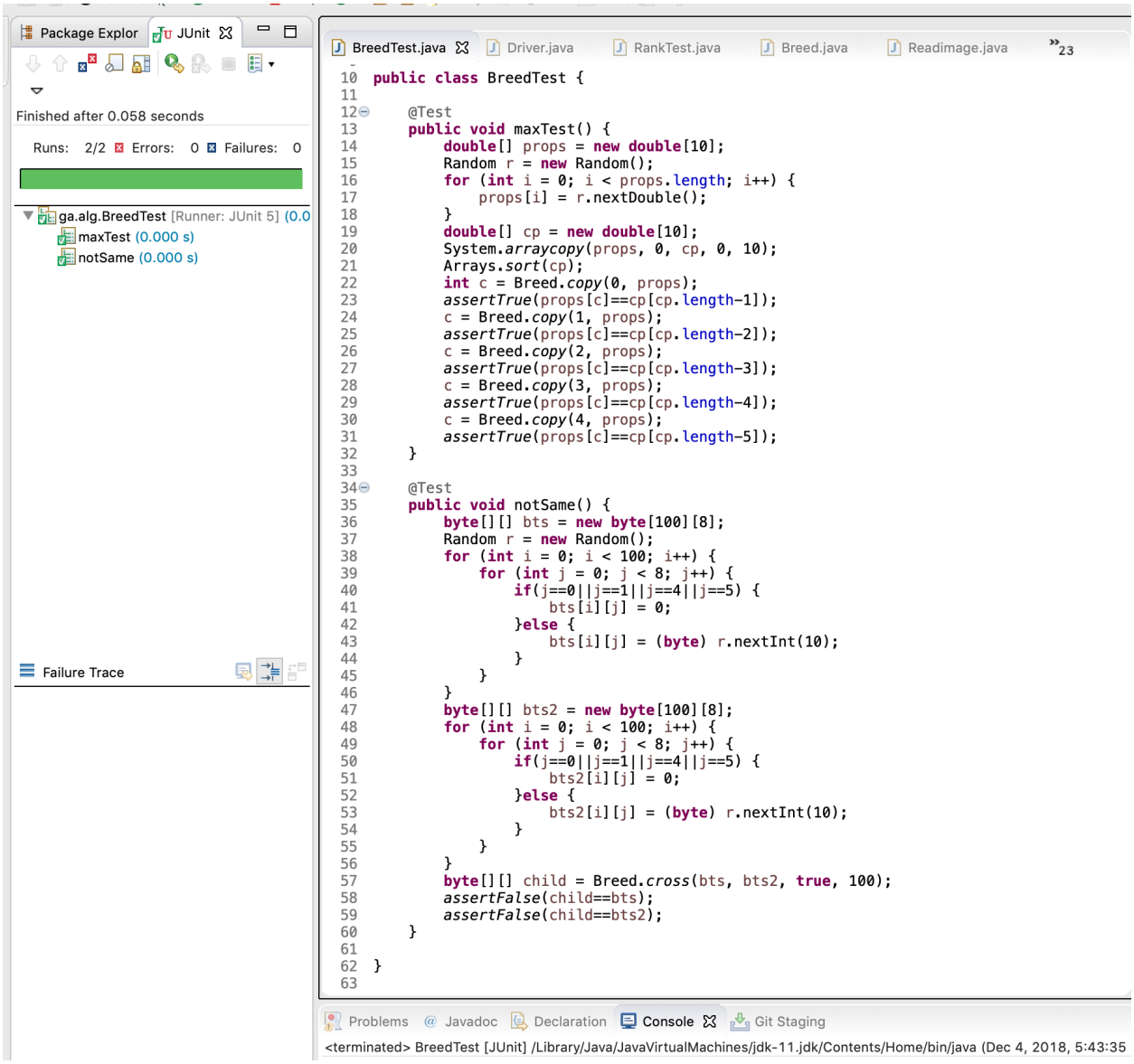
We set the generation is 5000 times and we set output an image once 100 times, here are the initial generation(Figure 1), result(Figure 2) and target image(Figure 3):

Figure 1 Figure 2 Figure 3


Package Explor
JUnit
Finished after 0.058 seconds
Runs: 2/2 Errors: 0 Failures: 0
ga.alg.BreedTest [Runner: JUnit 5] (0.0
maxTest (0.000 s)
notSame (0.000 s)
Failure Trace
BreedTest.java
Driver.java
RankTest.java
Breed.java
Readimage.java
23
public class BreedTest {
@Test
public void maxTest() {
double[] props = new double[10];
Random r = new Random();
for (int i = 0; i < props.length; i++) {
props[i] = r.nextDouble();
}
double[] cp = new double[10];
System.arraycopy(props, 0, cp, 0, 10);
Arrays.sort(cp);
int c = Breed.copy(0, props);
assertTrue(props[c]==cp[cp.length-1]);
c = Breed.copy(1, props);
assertTrue(props[c]==cp[cp.length-2]);
c = Breed.copy(2, props);
assertTrue(props[c]==cp[cp.length-3]);
c = Breed.copy(3, props);
assertTrue(props[c]==cp[cp.length-4]);
c = Breed.copy(4, props);
assertTrue(props[c]==cp[cp.length-5]);
}
@Test
public void notSame() {
byte[][] bts = new byte[100][8];
Random r = new Random();
for (int i = 0; i < 100; i++) {
for (int j = 0; j < 8; j++) {
if(j==0||j==1||j==4||j==5) {
bts[i][j] = 0;
}else {
bts[i][j] = (byte) r.nextInt(10);
}
}
}
byte[][] bts2 = new byte[100][8];
for (int i = 0; i < 100; i++) {
for (int j = 0; j < 8; j++) {
if(j==0||j==1||j==4||j==5) {
bts2[i][j] = 0;
}else {
bts2[i][j] = (byte) r.nextInt(10);
}
}
}
byte[][] child = Breed.cross(bts, bts2, true, 100);
assertFalse(child==bts);
assertFalse(child==bts2);
}
}
Problems
Javadoc
Declaration
Console
Git Staging
<terminated> BreedTest [JUnit] /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java (Dec 4, 2018, 5:43:35

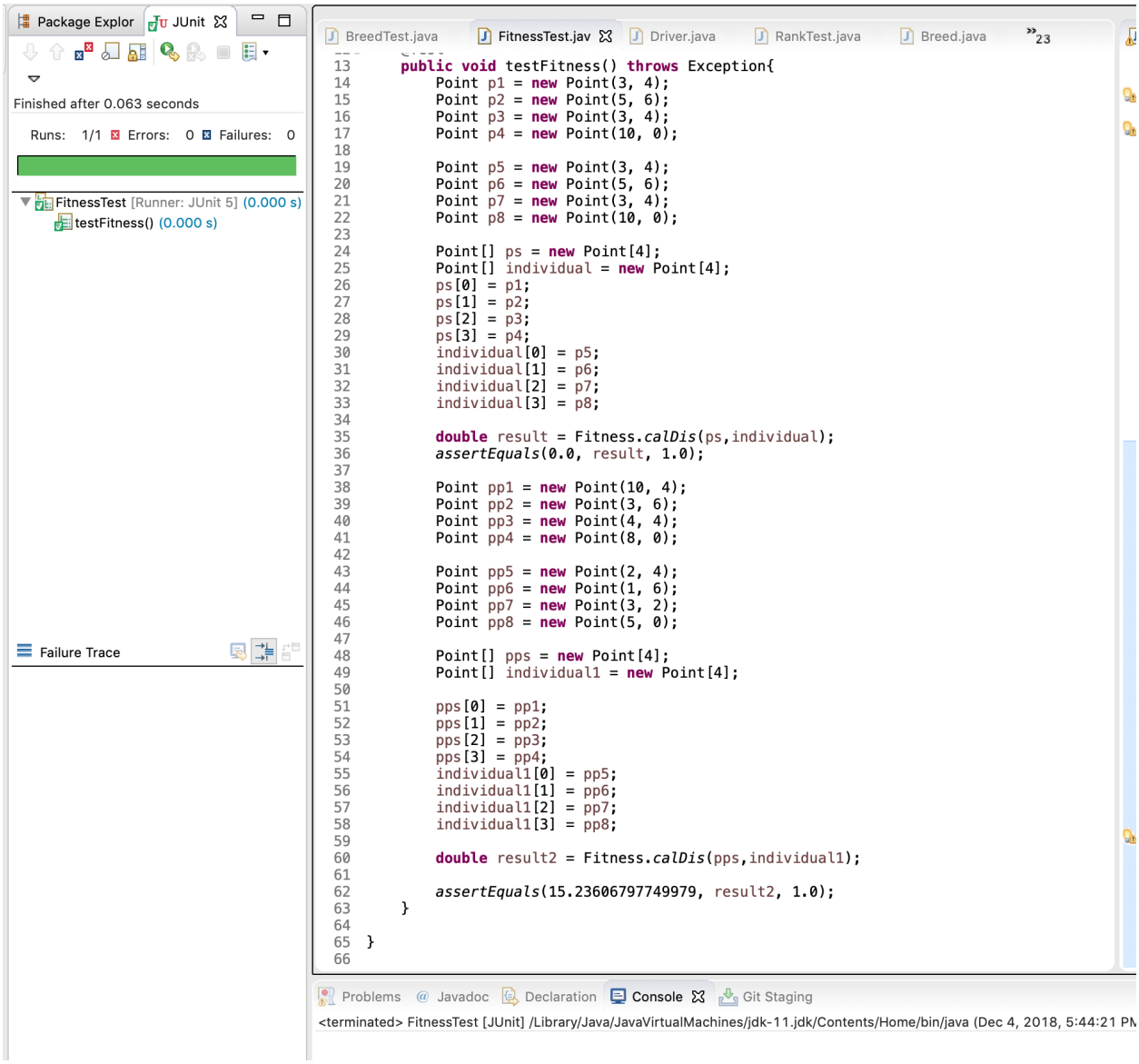
# 2. Unit Tests

Breed Test:

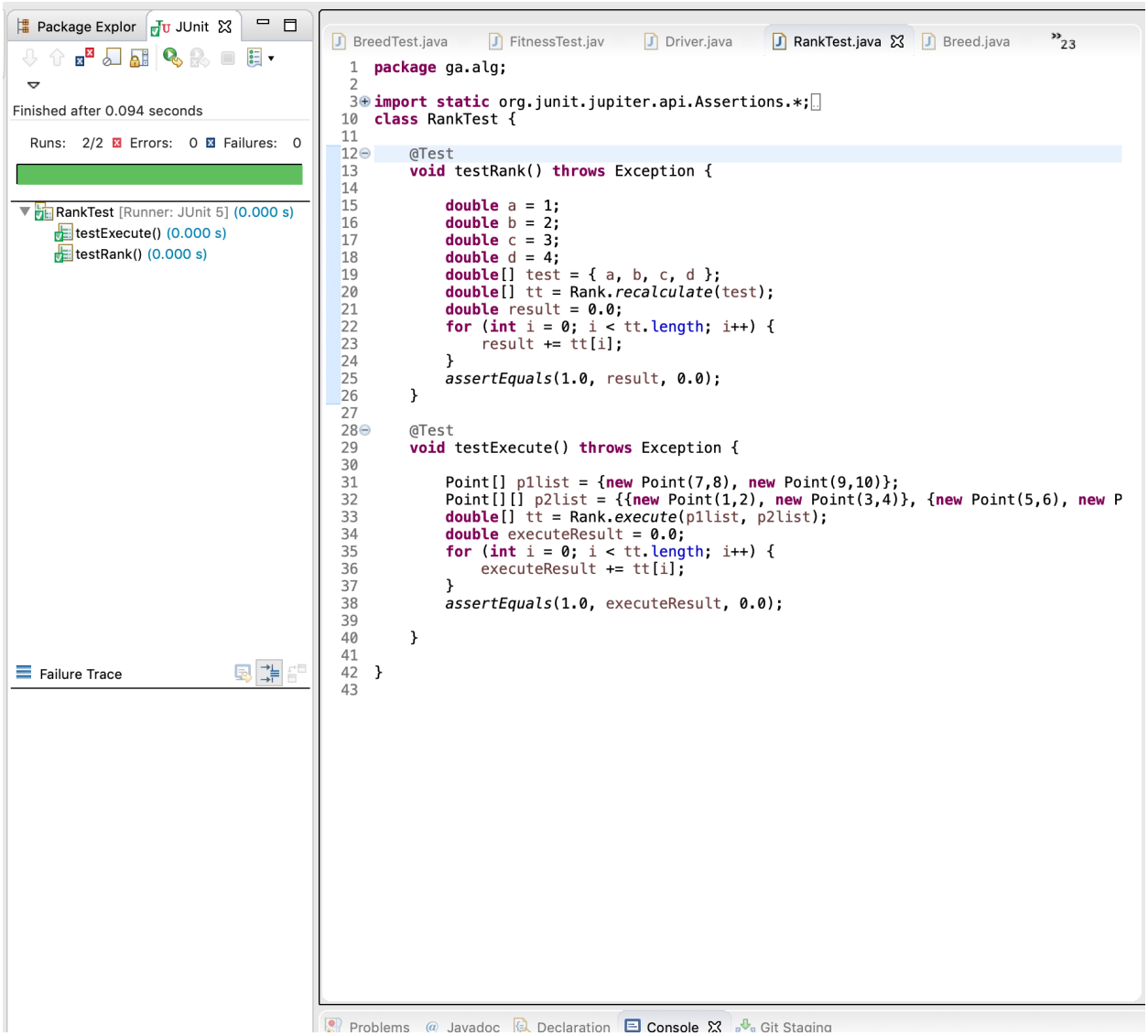



```java
public class BreedTest {

    @Test
    public void maxTest() {
        double[] props = new double[10];
        Random r = new Random();
        for (int i = 0; i < props.length; i++) {
            props[i] = r.nextDouble();
        }
        double[] cp = new double[10];
        System.arraycopy(props, 0, cp, 0, 10);
        Arrays.sort(cp);
        int c = Breed.copy(0, props);
        assertTrue(props[c]==cp[cp.length-1]);
        c = Breed.copy(1, props);
        assertTrue(props[c]==cp[cp.length-2]);
        c = Breed.copy(2, props);
        assertTrue(props[c]==cp[cp.length-3]);
        c = Breed.copy(3, props);
        assertTrue(props[c]==cp[cp.length-4]);
        c = Breed.copy(4, props);
        assertTrue(props[c]==cp[cp.length-5]);
    }

    @Test
    public void notSame() {
        byte[][] bts = new byte[100][8];
        Random r = new Random();
        for (int i = 0; i < 100; i++) {
            for (int j = 0; j < 8; j++) {
                if(j==0||j==1||j==4||j==5) {
                    bts[i][j] = 0;
                }else {
                    bts[i][j] = (byte) r.nextInt(10);
                }
            }
        }
        byte[][] bts2 = new byte[100][8];
        for (int i = 0; i < 100; i++) {
            for (int j = 0; j < 8; j++) {
                if(j==0||j==1||j==4||j==5) {
                    bts2[i][j] = 0;
                }else {
                    bts2[i][j] = (byte) r.nextInt(10);
                }
            }
        }
        byte[][] child = Breed.cross(bts, bts2, true, 100);
        assertFalse(child==bts);
        assertFalse(child==bts2);
    }

}
```

Fitness Test:

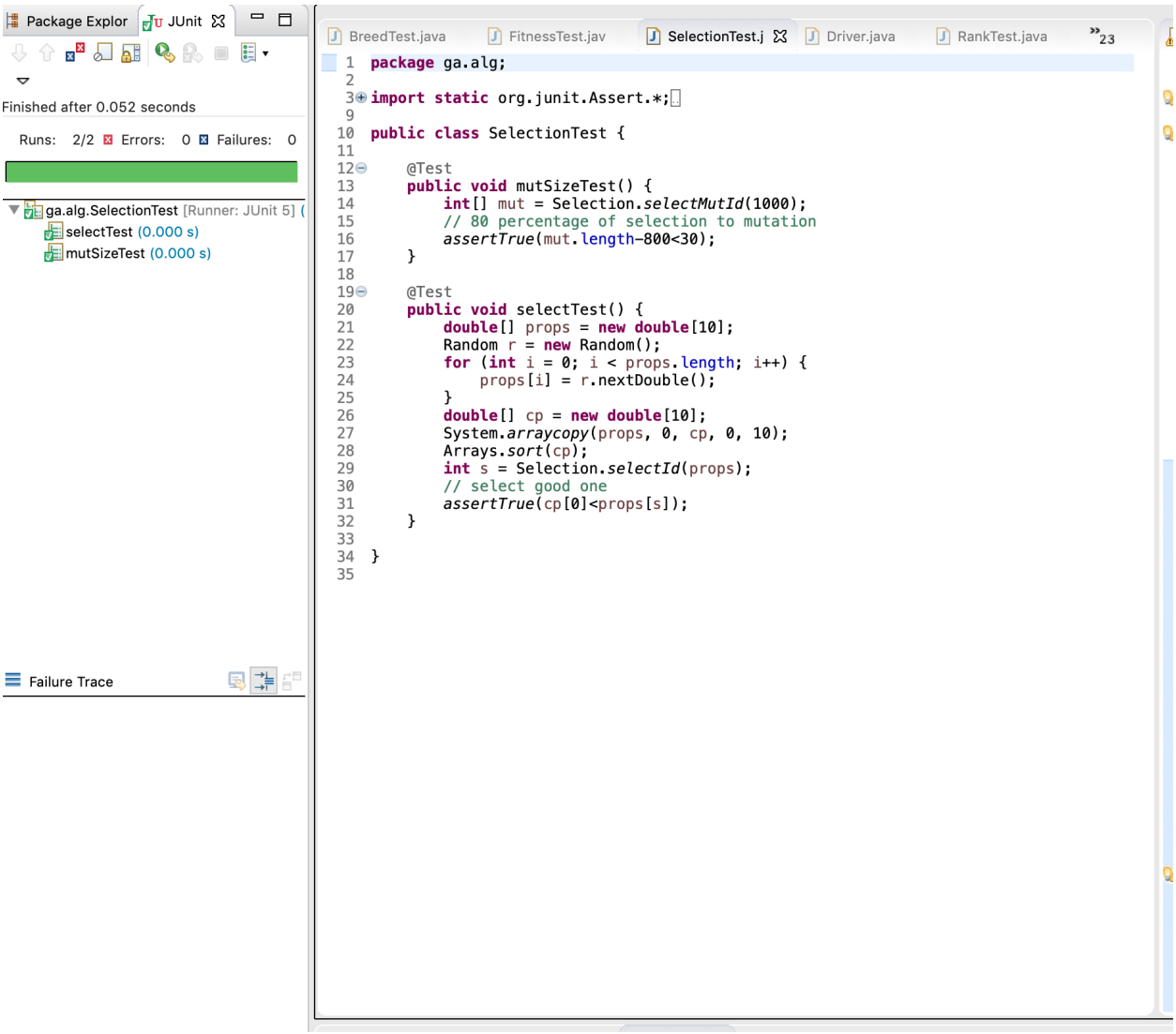


Rank Test:

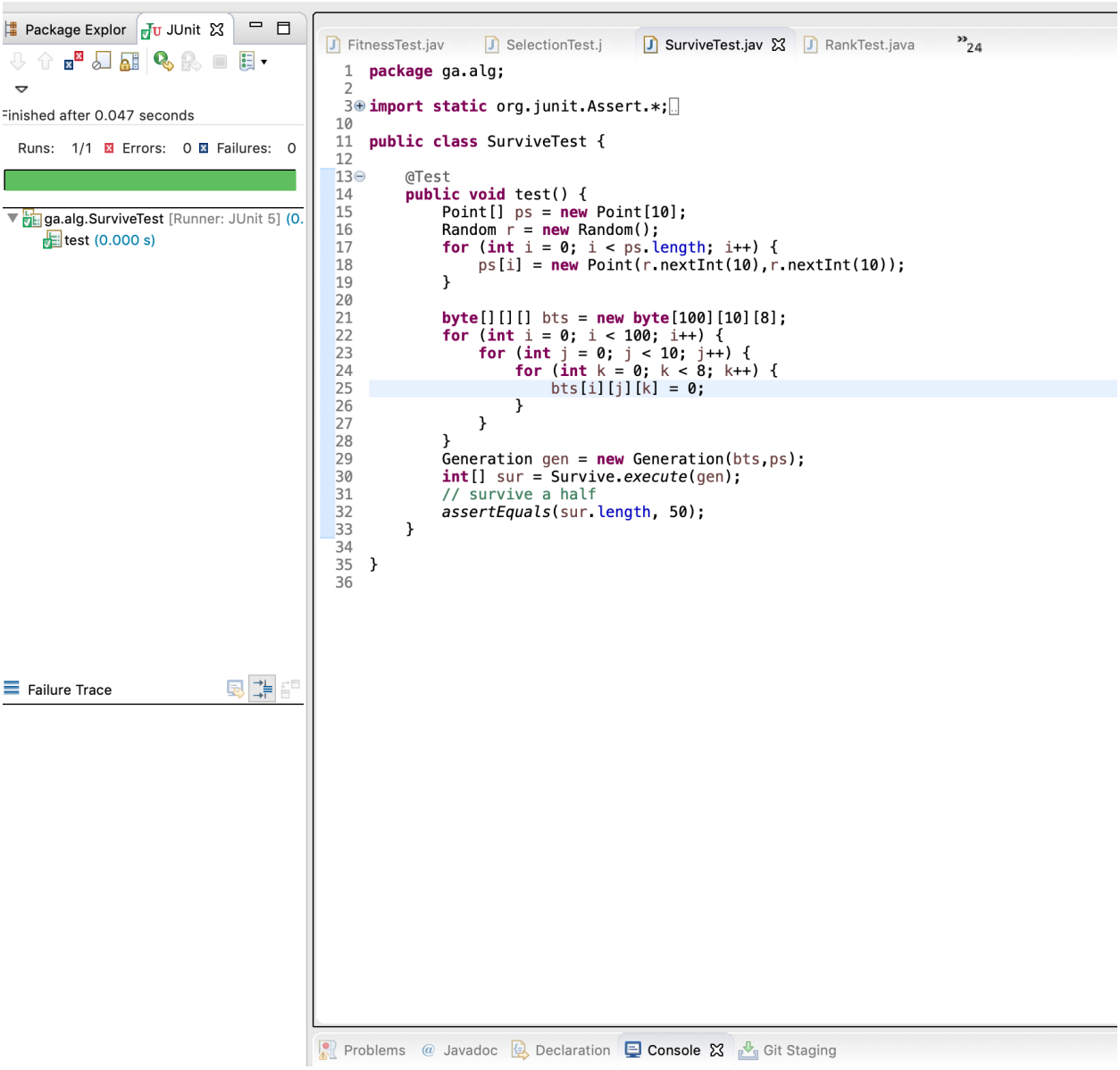



```java
package ga.alg;

import static org.junit.jupiter.api.Assertions.*;
class RankTest {

    @Test
    void testRank() throws Exception {

        double a = 1;
        double b = 2;
        double c = 3;
        double d = 4;
        double[] test = { a, b, c, d };
        double[] tt = Rank.recalculate(test);
        double result = 0.0;
        for (int i = 0; i < tt.length; i++) {
            result += tt[i];
        }
        assertEquals(1.0, result, 0.0);
    }

    @Test
    void testExecute() throws Exception {

        Point[] p1list = {new Point(7,8), new Point(9,10)};
        Point[][] p2list = {{new Point(1,2), new Point(3,4)}, {new Point(5,6), new P
        double[] tt = Rank.execute(p1list, p2list);
        double executeResult = 0.0;
        for (int i = 0; i < tt.length; i++) {
            executeResult += tt[i];
        }
        assertEquals(1.0, executeResult, 0.0);

    }

}
```

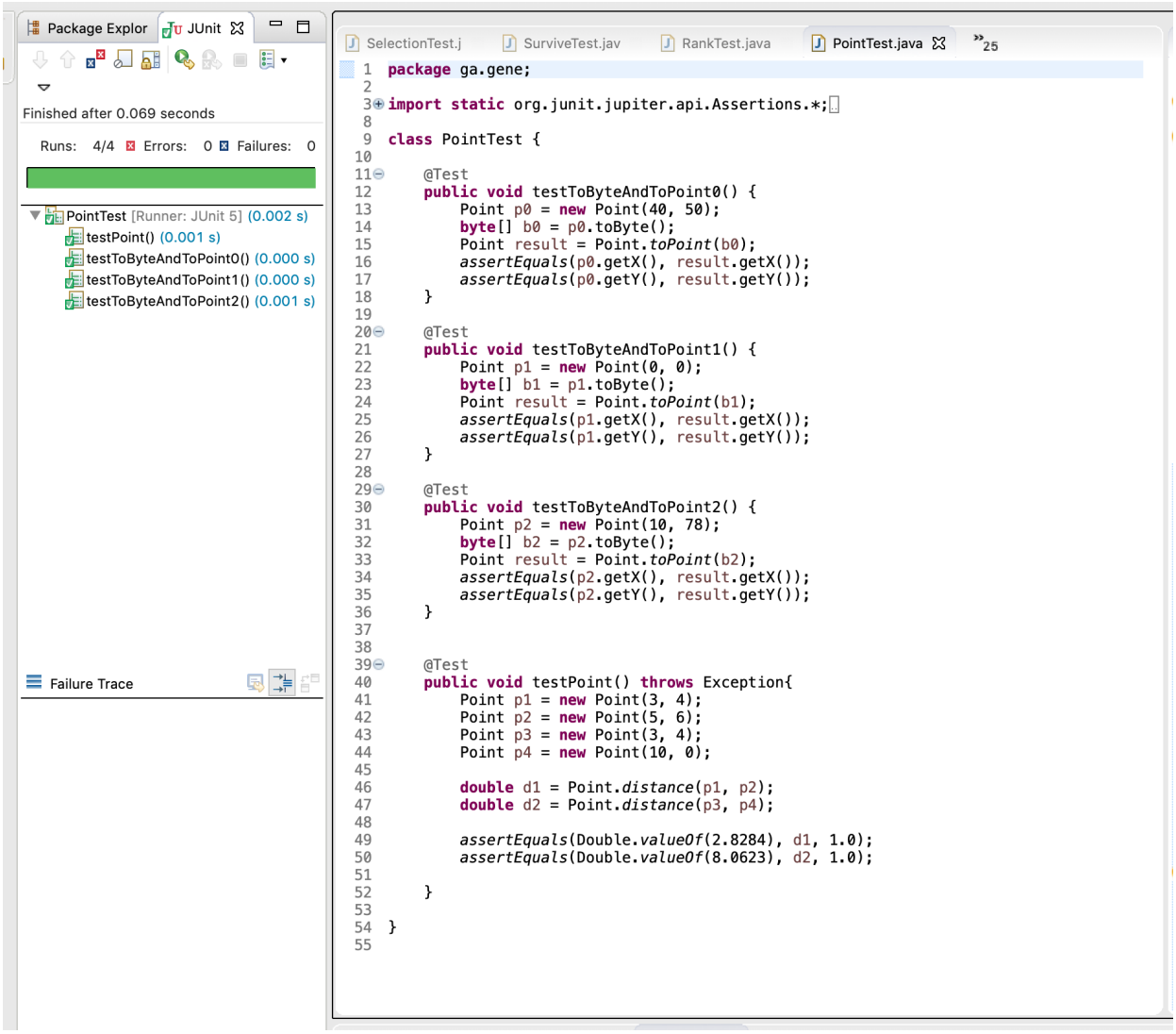
Selection Test:




Survive Test::


Package Explor
JUnit
Finished after 0.047 seconds
Runs: 1/1 Errors: 0 Failures: 0
ga.alg.SurviveTest [Runner: JUnit 5] (0.
test (0.000 s)
Failure Trace
FitnessTest.jav
SelectionTest.j
SurviveTest.jav
RankTest.java
24
1 package ga.alg;
2
3 import static org.junit.Assert.*;
10
11 public class SurviveTest {
12
13 @Test
14 public void test() {
15 Point[] ps = new Point[10];
16 Random r = new Random();
17 for (int i = 0; i < ps.length; i++) {
18 ps[i] = new Point(r.nextInt(10),r.nextInt(10));
19 }
20
21 byte[][][] bts = new byte[100][10][8];
22 for (int i = 0; i < 100; i++) {
23 for (int j = 0; j < 10; j++) {
24 for (int k = 0; k < 8; k++) {
25 bts[i][j][k] = 0;
26 }
27 }
28 }
29 Generation gen = new Generation(bts,ps);
30 int[] sur = Survive.execute(gen);
31 // survive a half
32 assertEquals(sur.length, 50);
33 }
34
35 }
36
Problems
Javadoc
Declaration
Console
Git Staging

Point Test:




## 3. Generations VS Distances

Here are the graph for the generation vs distance. X-axis is the generation, Y-axis is the distance, When generations get more, we can see the Y-axis is closer to 0,  we can see more generations less distance we can get.

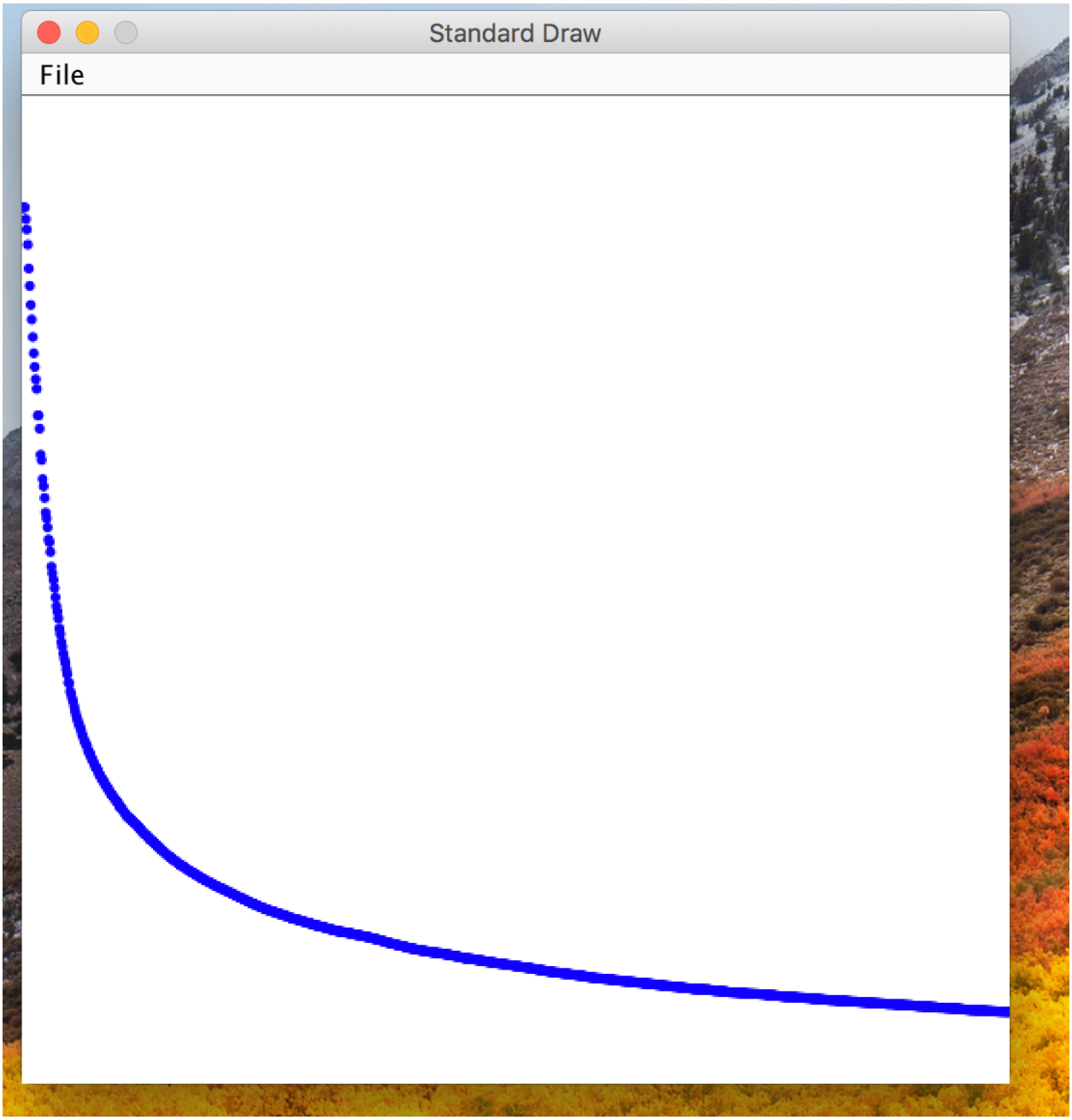



Figure 1000 Generation VS Distance

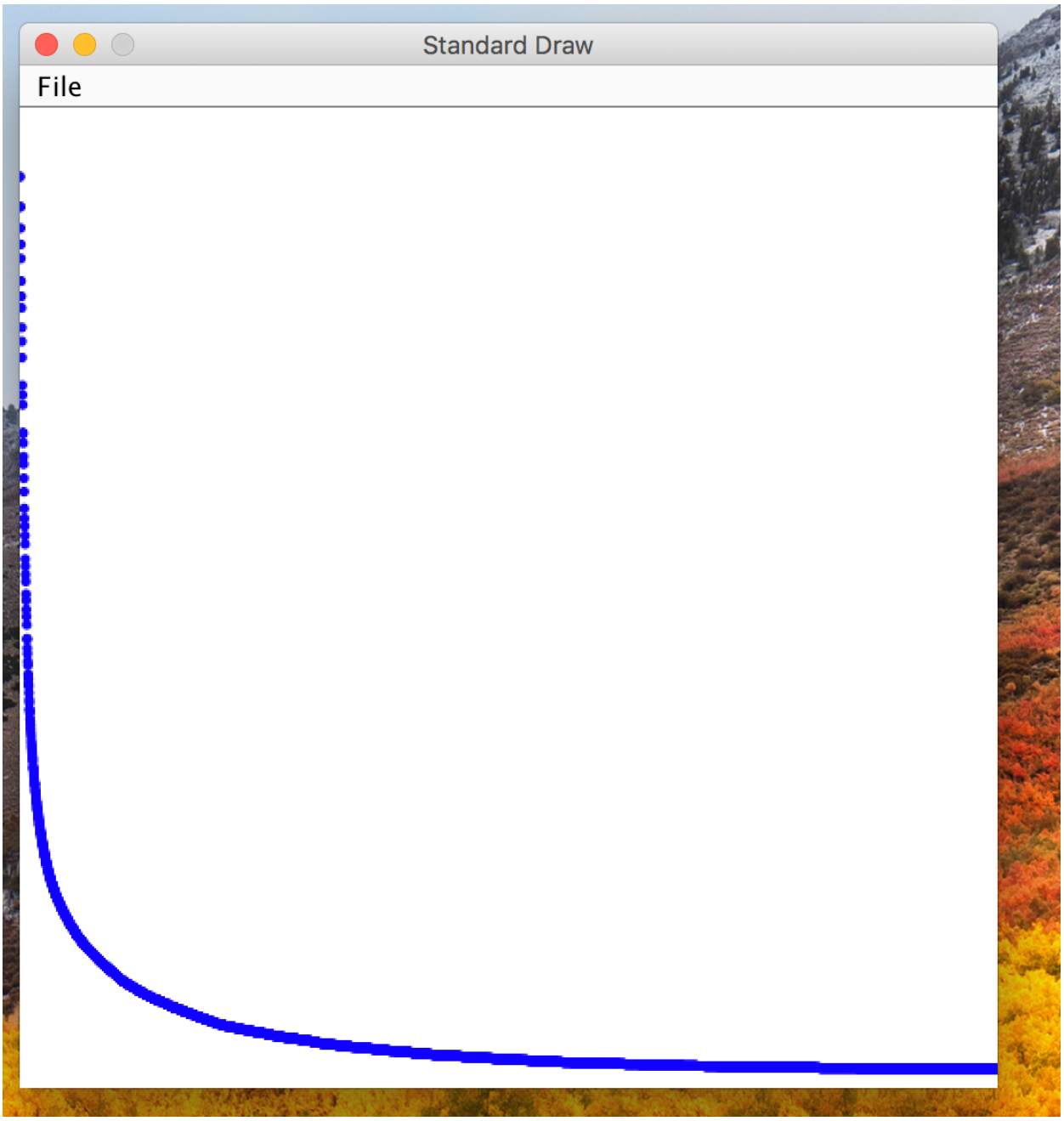



Figure 5000 Generation VS Distance

## 4. Generations VS Fitness

The more generations we get, the larger fitness value we get.

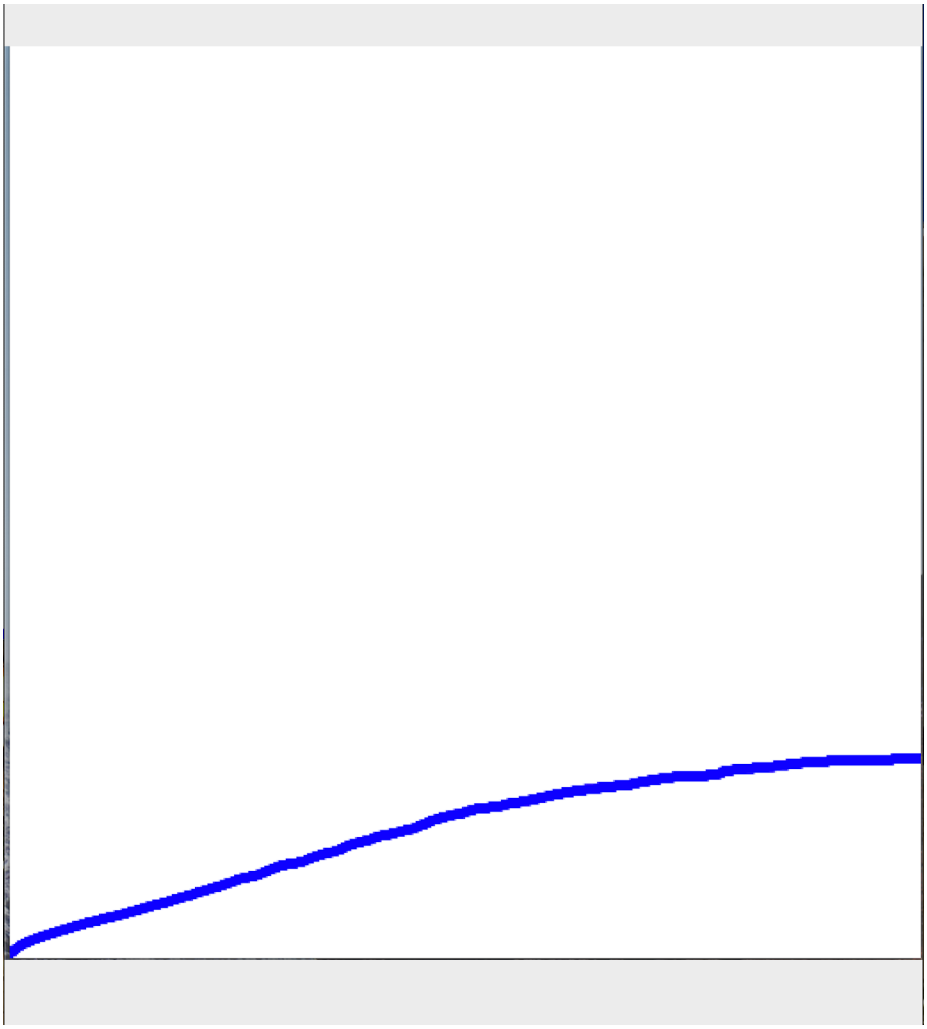


Figure 5000 generation Fitness

## Conclusion

During the process of the evolution, we can see from the diagram that the total distances between the target and the best individual of one generation is becoming smaller. In other words, the fitness for individual is growing. In the first 500 generations, the growing is faster, after that, the speeding of growing is becoming slower. In this time, if we still go as before, the change may be small and there will need longer time for the child of the generation to get closer to the target. So we need a more powerful mutation. We use big mutation. That means if there are less difference between the best and the average, we change more genes in one individual to make the diversity bigger.

To make the program run faster, we use the multi-thread in the crossover process.

We turn the point to a picture each 100 generations to show if the algorithm runs correctly and if the image we drew is more likely to the target picture.

After 5000 generations, we can see the result is very close to the target.