

---

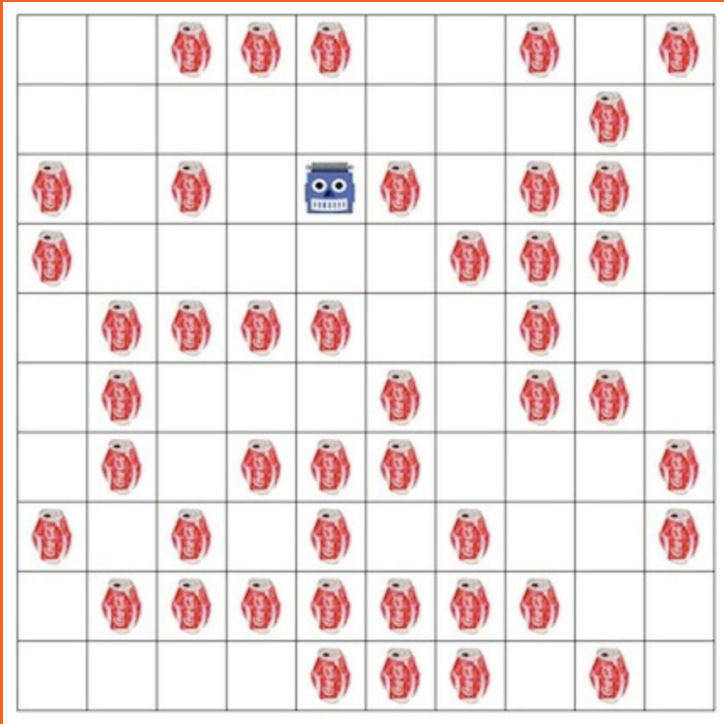
# GA Project -- Robot Robi

Team: 217  
Haoyu Yin  
Jianxi Li  
Mingguan Liu

---

# Introduction

Robi is a self-directed robot and its task is to pick up the garbage in a  $M \times M$  map. In this game Robi have a limited steps of movements. In the limited movements Robi need to get a grade as high as possible.



---

# Map

The map will be a  $M \times N$  grids map.(in this case  $M=10$ ,  $N=10$ )  
There will be 50 soda cans in the grids map randomly spread.  
The status of each grid in the map can be either empty(0) or have a soda can(1) or wall(2).

Every time the robot Robi make a move, will get a value of the target grid(0,1,2). If Robi is making a move to hit the wall(the coordinate of target grid is outside of the map) then it will get a value 2 that indicate Robi hit on a wall.

---

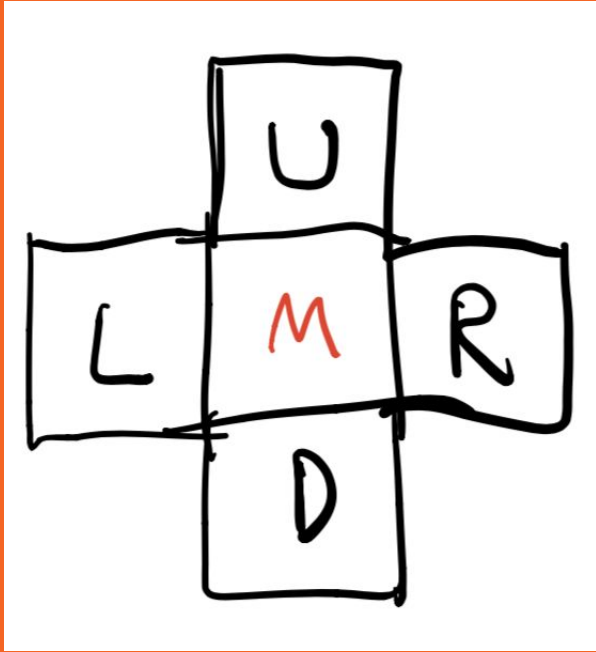
0	0	0	0	0	0	0
0	1	0	1	0	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	
0	0	0	0	0		1



1	1					
				1		
1						
						1

---

# Vision of Robi



Unfortunately, Robi is only able to observe the 5 grid around him. The UP, DOWN, LEFT, RIGHT and the grid the robot is staying.

---

---

# Status of Robi

From the previous slides, we could know that Robi can observe the five grid around him. And according the status, we have made a function to reflect its status in a 5-digit Integer. Each digit represent a specific grid. And latter it will

---

---

# Gene of Robi

In this project, we have the gene of robi is designed to be stored in an Integer array with length of 243. There are 3 possible status of each grid of the 5 near Robi. So the all possible status is  $3^5 = 243$ .

Each index of the gene represent one status and the movement of Robi when he found that he is in this situation.

---

---

# Behaviors of Robi

In our project, Robi have 7 kind of behavior: move left, move right, move up, move down, random move, pick up trash, do nothing.

And the behavior will cause different result of the grade, here are the rule of the grading:

pick up a garbage successfully +10;

pick up nothing -1;

knock the wall -5;

---



---

# Population-parameter

Size(Initial (“seed”) population): 200

Cross probability: 0.82

Mutation probability: 0.078

Max mutation count: 10

Dead rate: 0.25

---

---

# Population-initial population

When initial a population, we need to create some objects according to the size of population.

In our project, we create 200 robot, which has a random gene. For each robot, they need to clean 1000 different map. Then, they can get a average grade.

---

---

# Population

After each robot has a average grade. We are going to generate a new generation.

kill part of individuals -> select parents -> generate new gene  
-> gene mutation -> generate children and replace current generation

---

---

---

Kill part of individuals

In order to accelerate the evolution process, we decide to sort the robots in population and kill the worst  $\frac{1}{4}$  robot.

This may be a little crueller than real nature, but what we want to do is get a better solution. And this is a good way to select better parents.

---

---

# Select parents

1. Generate fitness probabilities——Roulette game

Generate an array of probabilities according to their grade. If there are some grades less than zero, we need to add the absolute value of the minimum grade to each robot's grade.

What we do next is get the sum of grades of each robot. Then use the grade over the total grade to get the fitness of each robot.

---

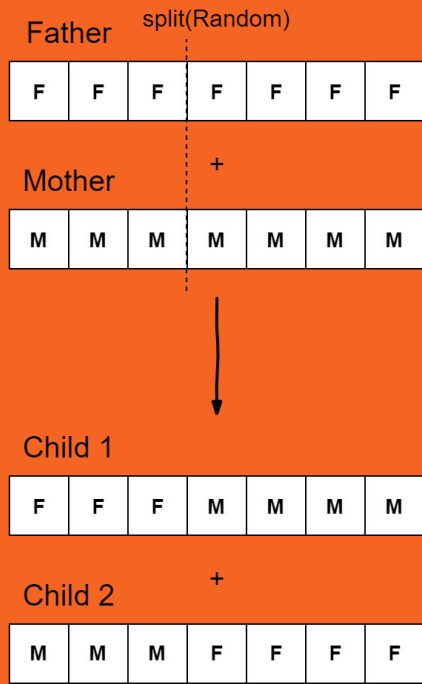
---

# Select parents

2. Choose parents using the probabilities

After generate the fitness probabilities, use them to choose two integer, which represent the index of the parents.

---



---

# Gene cross

After select parents, we can start reproduction.

We set a probability of cross, which means those two parents may not generate children, but still live in the new generation.

When doing gene cross, we random split the father gene and mother gene. And generate two children, first uses first part of father and second part of mother, and second uses second part of father and first part of mother.

---

---

# Gene mutation

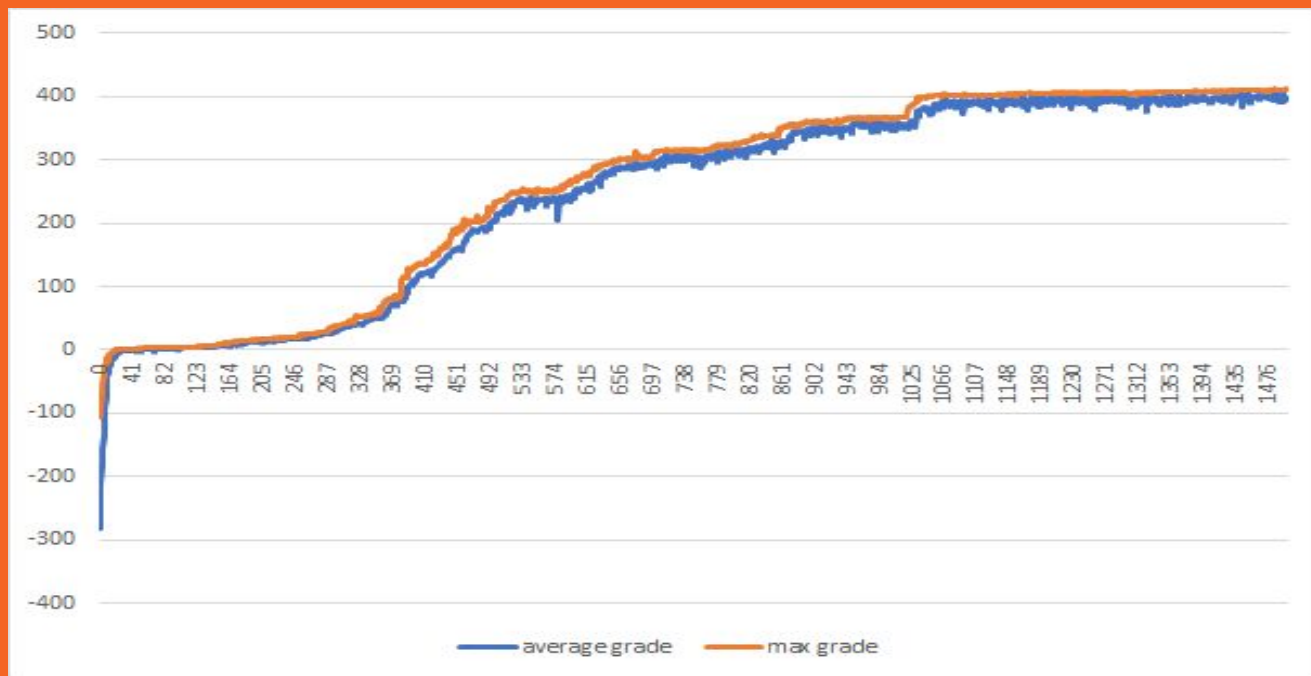
After generate two children gene from parents, the genes may mutate.

The mutation rate is very low. In our project, it is 7.8%. When mutation, limited number of chromosomes may change randomly. It is good to keep the diversity of the population.

---



# Result



# Test Case -- Map Test

Finished after 0.073 seconds

Runs: 2/2    Errors: 0    Failures: 0

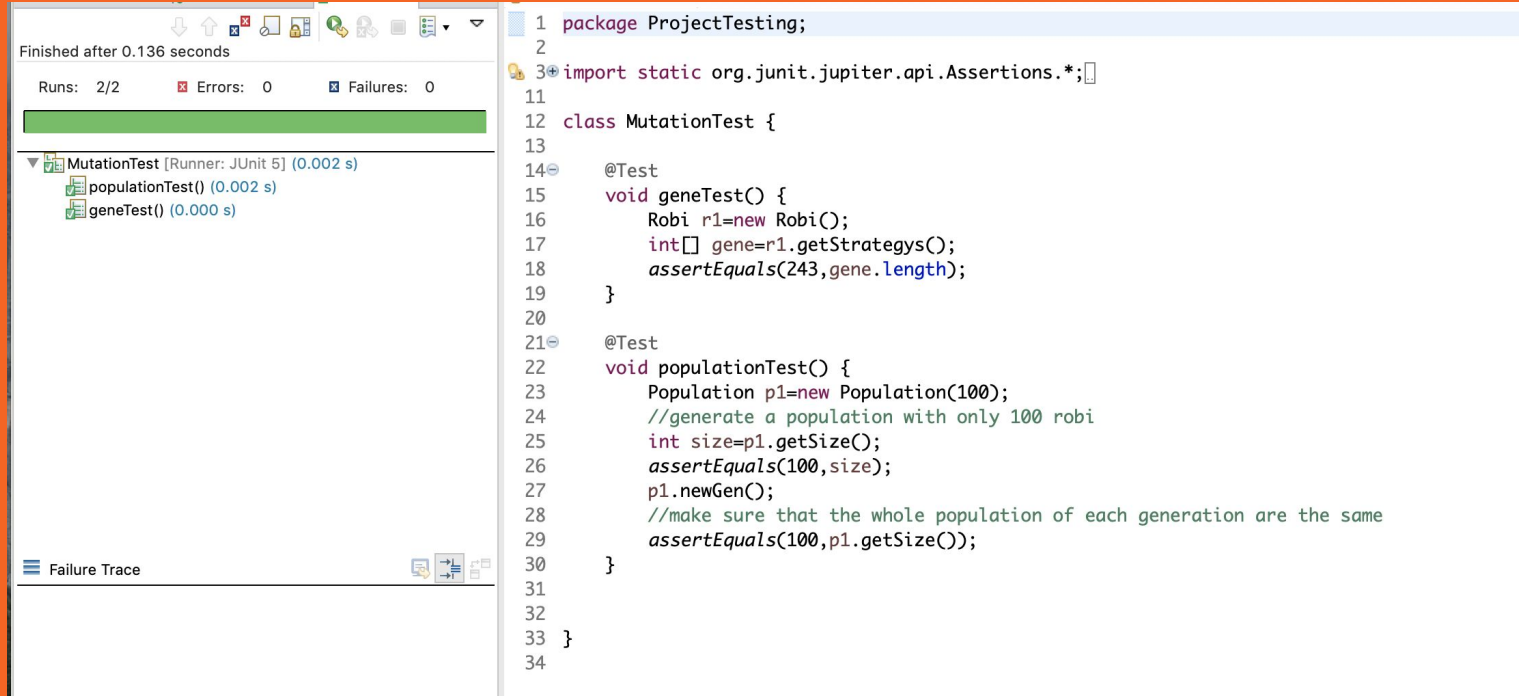
MapTest [Runner: JUnit 5] (0.000 s)

- testlength() (0.000 s)
- testGarbage() (0.000 s)

Failure Trace

```
1 package ProjectTesting;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class MapTest {
6
7     @Test
8     void testlength() {
9         Stage map = new Stage();
10        // will generate a 10*10 map and with 50 garbage
11        int[][] maparray = map.getStage();
12        assertEquals(10, maparray.length);
13        assertEquals(10, maparray[0].length);
14    }
15
16     @Test
17     void testGarbage() {
18         Stage map = new Stage();
19        // will generate a 10*10 map and with 50 garbage
20        int[][] maparray = map.getStage();
21        int count = 0;
22        for (int i = 0; i < maparray.length; i++) {
23            for (int j = 0; j < maparray[i].length; j++) {
24                if (maparray[i][j] == 1) {
25                    count++;
26                }
27            }
28        }
29        assertEquals(50, count);
30    }
31 }
```

# TestCase -- Reproduce



The screenshot displays an IDE interface with two main panels. The left panel shows the test execution results, and the right panel shows the source code of the test class.

**Test Execution Results (Left Panel):**

- Finished after 0.136 seconds
- Runs: 2/2
- Errors: 0
- Failures: 0
- Test Class: `MutationTest` [Runner: JUnit 5] (0.002 s)
- Test Methods:
  - `populationTest()` (0.002 s)
  - `geneTest()` (0.000 s)
- Failure Trace: (Empty)

**Source Code (Right Panel):**

```
1 package ProjectTesting;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7
8
9
10
11
12 class MutationTest {
13
14     @Test
15     void geneTest() {
16         Robi r1=new Robi();
17         int[] gene=r1.getStrategys();
18         assertEquals(243,gene.length);
19     }
20
21     @Test
22     void populationTest() {
23         Population p1=new Population(100);
24         //generate a population with only 100 robi
25         int size=p1.getSize();
26         assertEquals(100,size);
27         p1.newGen();
28         //make sure that the whole population of each generation are the same
29         assertEquals(100,p1.getSize());
30     }
31
32
33 }
34
```