INFO 6205 Fall 2018 Final Project

Group:303 MengQi Wang ,QiJun Hou

Genetic Algorithm-
Finding the best path of a maze

Introduction

As we all know, genetic algorithms can be used to find best solutions for problems. And we find GAs well suited to maze problems with best path. So we try to design a genetic algorithm to find the shortest path of a maze.

Input:

A maze with a least step number

For example:

2	0	-1	0	-1 = block
0	0	-1	0	0 = road
-1	0	0	0	1 = start point
-1	0	-1	1	2 = exit point

Output:

The steps from start point to end point

For example:

up, left, left, up, up, left

Algorithm details

Genetic code

The genetic code is a list of 0 & 1, each gene has two digits, and the number of genes is the least step number, so the length of the whole genetic code is 2 * least step number.

For the example above:

The least step number is 6, so there will be 6 gene and the length of the genetic code is 12.

Like: 00 10 10 00 00 10

Gene expression

There are four values of each gene and four directions, their expression relationship is:

00	up
01	down
10	left
11	right

For the genetic code example above:

00 10 10 00 00 10 represents: up, left, left, up, up, left

Fitness

endPosition is the position after movements according to the genetic steps

xDistance = |endPositionX - exitPointX|

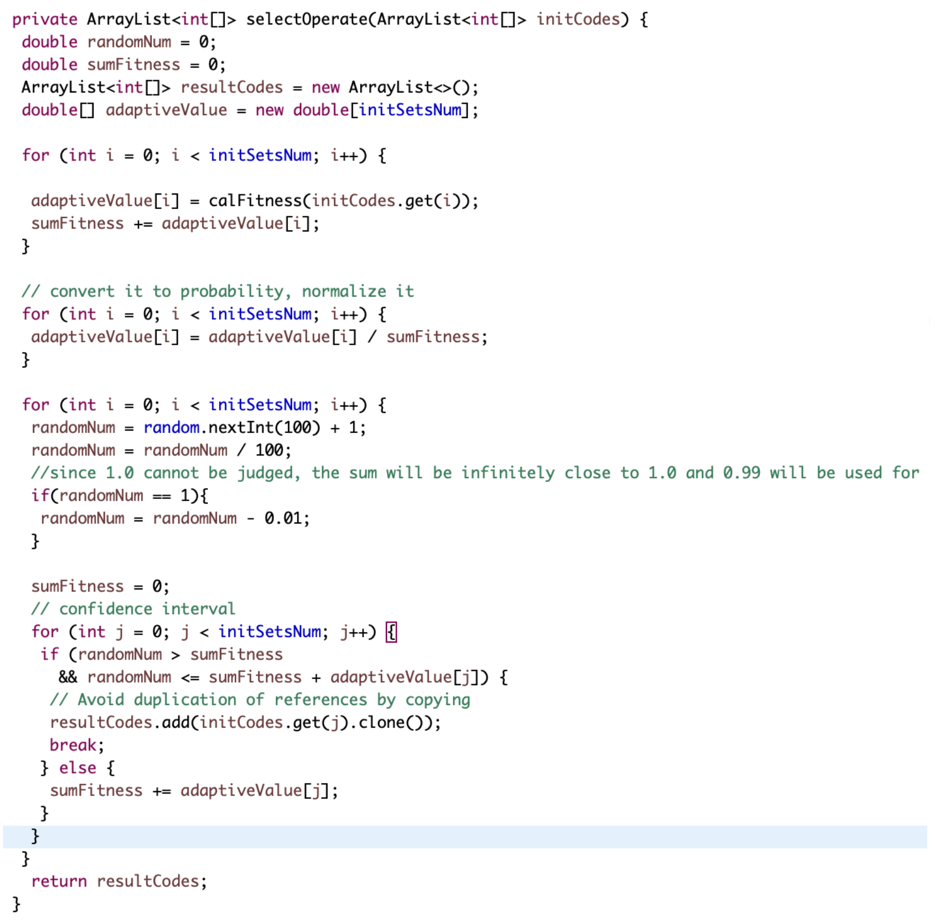
yDistance = |endPositionY - exitPointY|

fitness = 1 / (1 + xDistance + yDistance)

The value of fitness can represent the distance between endPosition and the exit point. 1 is the max value which means ending at the exit point. The closer fitness to 1, the closer end position to exit point.

Selection

Each candidate in one generation will be given a selected probability(adaptiveValue[]). The sum of candidates' selected probability is 1.There is a random number(randomNum) between 0-1 and a counter(sumFitness) starting with 0. We select candidates according to adaptiveValue/randomNum/sumFitness.   code:

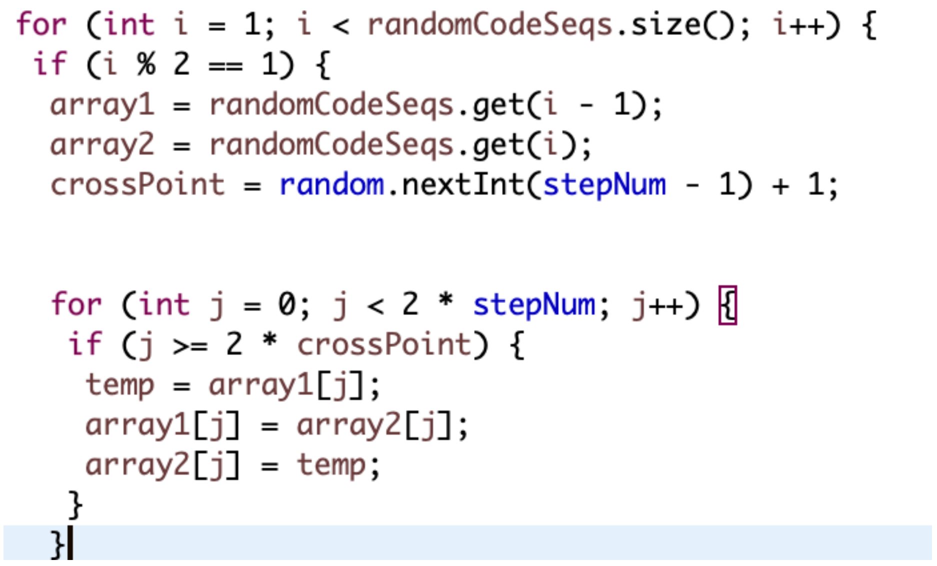


```java
private ArrayList<int[]> selectOperate(ArrayList<int[]> initCodes) {
 double randomNum = 0;
 double sumFitness = 0;
 ArrayList<int[]> resultCodes = new ArrayList<>();
 double[] adaptiveValue = new double[initSetsNum];

 for (int i = 0; i < initSetsNum; i++) {

  adaptiveValue[i] = calFitness(initCodes.get(i));
  sumFitness += adaptiveValue[i];
 }

 // convert it to probability, normalize it
 for (int i = 0; i < initSetsNum; i++) {
  adaptiveValue[i] = adaptiveValue[i] / sumFitness;
 }

 for (int i = 0; i < initSetsNum; i++) {
  randomNum = random.nextInt(100) + 1;
  randomNum = randomNum / 100;
  //since 1.0 cannot be judged, the sum will be infinitely close to 1.0 and 0.99 will be used for
  if(randomNum == 1){
   randomNum = randomNum - 0.01;
  }

  sumFitness = 0;
  // confidence interval
  for (int j = 0; j < initSetsNum; j++) {
   if (randomNum > sumFitness
     && randomNum <= sumFitness + adaptiveValue[j]) {
    // Avoid duplication of references by copying
    resultCodes.add(initCodes.get(j).clone());
    break;
   } else {
    sumFitness += adaptiveValue[j];
   }
  }
 }
  return resultCodes;
}
```

Crossover

There is a random number(randomNum) between 1-least step number. The genes after gene[randomNum * 2] will be swapped.

code:



```
for (int i = 1; i < randomCodeSeqs.size(); i++) {
 if (i % 2 == 1) {
  array1 = randomCodeSeqs.get(i - 1);
  array2 = randomCodeSeqs.get(i);
  crossPoint = random.nextInt(stepNum - 1) + 1;


  for (int j = 0; j < 2 * stepNum; j++) {
   if (j >= 2 * crossPoint) {
    temp = array1[j];
    array1[j] = array2[j];
    array2[j] = temp;
   }
  }
```

Mutation

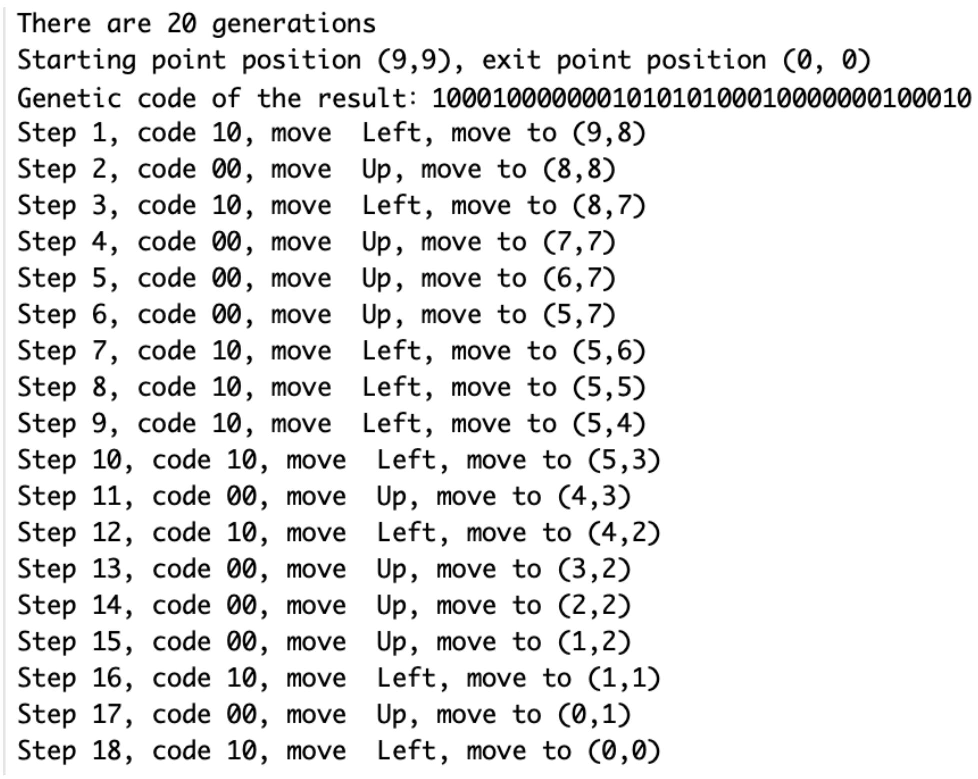
There is a mutation rate(mutationRate), the default value is 1%. For each candidate, generate a random number between 0-1, if it’s lower than mutationRate, start the mutation process.

In the mutation process, generate a random number(variationPoint) between 0 - least step number. Mutate this point.

Program running

Result

On the console window, the output result will show how many steps are there in the maze to find the exit and show the detail from which point to which point.It also gives the instruction that if the program find the shortest test under the limit times test.Besides, the console shows the times of genetic evolution times as well.



```
There are 20 generations
Starting point position (9,9), exit point position (0, 0)
Genetic code of the result: 100010000000101010100010000000100010
Step 1, code 10, move  Left, move to (9,8)
Step 2, code 00, move  Up, move to (8,8)
Step 3, code 10, move  Left, move to (8,7)
Step 4, code 00, move  Up, move to (7,7)
Step 5, code 00, move  Up, move to (6,7)
Step 6, code 00, move  Up, move to (5,7)
Step 7, code 10, move  Left, move to (5,6)
Step 8, code 10, move  Left, move to (5,5)
Step 9, code 10, move  Left, move to (5,4)
Step 10, code 10, move  Left, move to (5,3)
Step 11, code 00, move  Up, move to (4,3)
Step 12, code 10, move  Left, move to (4,2)
Step 13, code 00, move  Up, move to (3,2)
Step 14, code 00, move  Up, move to (2,2)
Step 15, code 00, move  Up, move to (1,2)
Step 16, code 10, move  Left, move to (1,1)
Step 17, code 00, move  Up, move to (0,1)
Step 18, code 10, move  Left, move to (0,0)
```

Screenshot

Maze:

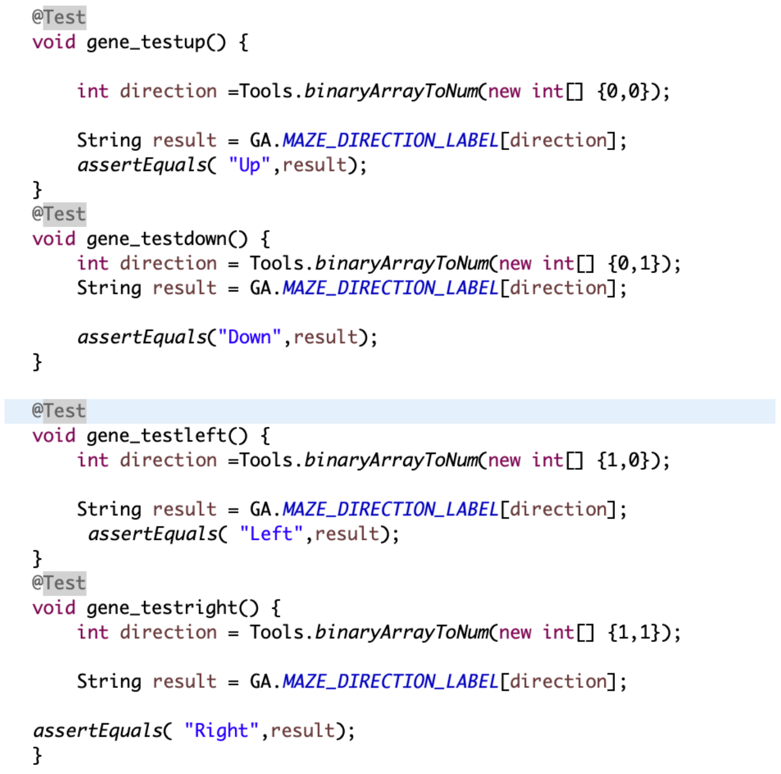
```
2 0 0 0 0 -1 -1 0 0 0
0 0 0 0 0 0 0 0 -1 0
0 -1 0 -1 0 -1 0 0 0 0
0 -1 0 0 0 0 0 0 0 0
0 0 0 0 -1 0 -1 0 0 0
0 0 -1 0 0 0 0 0 0 0
0 0 -1 0 0 0 0 0 0 0
0 -1 0 0 0 0 -1 0 0 0
0 0 0 -1 0 0 0 0 0 0
-1 0 0 0 0 0 -1 0 0 1
```

We made four mazes and output 4 different results.

Unit tests

In this project, we made 8 unit tests to determine each of the function whether works.We wrote 4 groups of gene expression unit test to check if the output binary number equals to the direction we assigned to it.

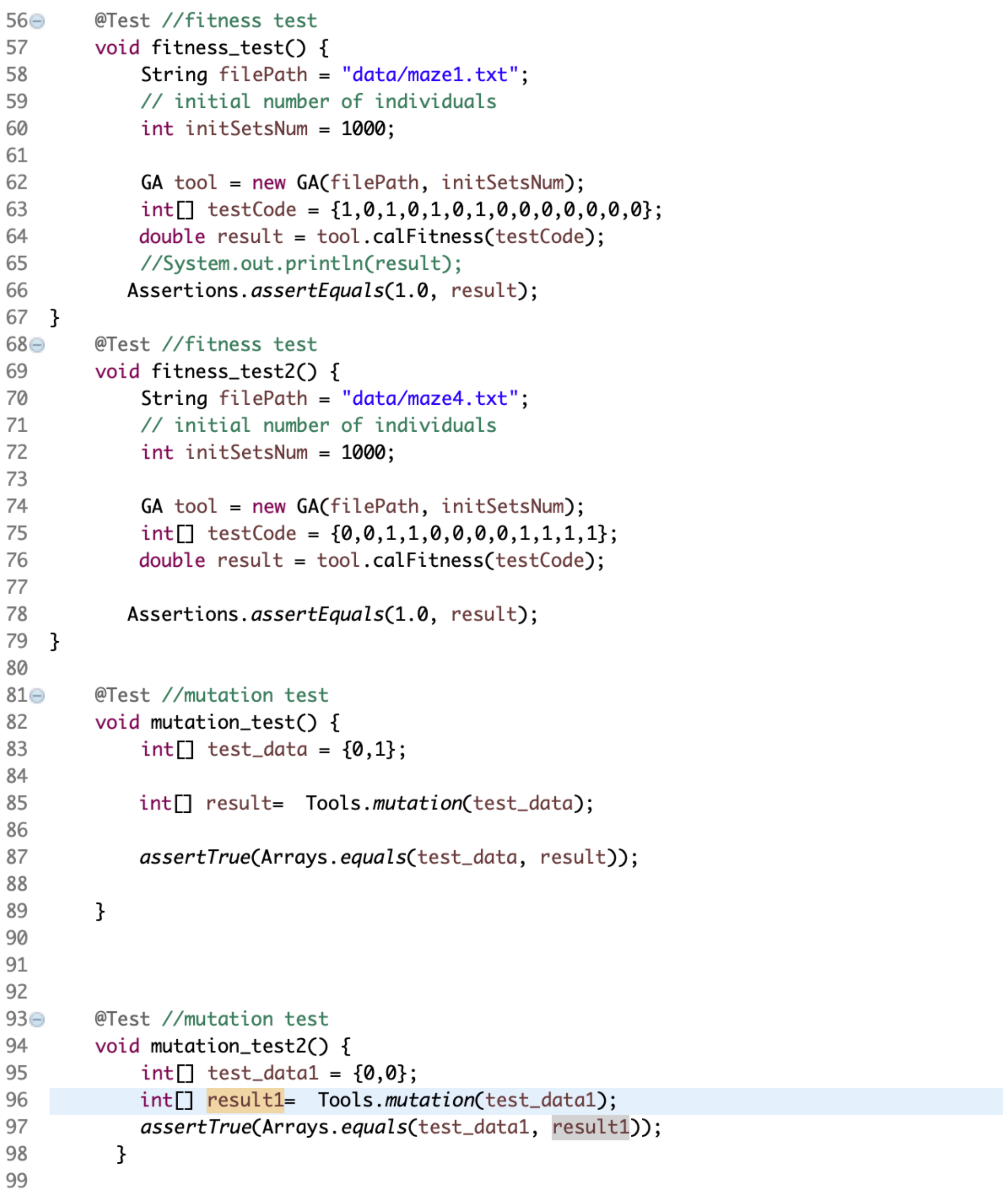
Code:



```
@Test
void gene_testup() {

    int direction =Tools.binaryArrayToNum(new int[] {0,0});

    String result = GA.MAZE_DIRECTION_LABEL[direction];
    assertEquals( "Up",result);
}
@Test
void gene_testdown() {
    int direction = Tools.binaryArrayToNum(new int[] {0,1});
    String result = GA.MAZE_DIRECTION_LABEL[direction];

    assertEquals("Down",result);
}

@Test
void gene_testleft() {
    int direction =Tools.binaryArrayToNum(new int[] {1,0});

    String result = GA.MAZE_DIRECTION_LABEL[direction];
     assertEquals( "Left",result);
}
@Test
void gene_testright() {
    int direction = Tools.binaryArrayToNum(new int[] {1,1});

    String result = GA.MAZE_DIRECTION_LABEL[direction];

assertEquals( "Right",result);
}
```

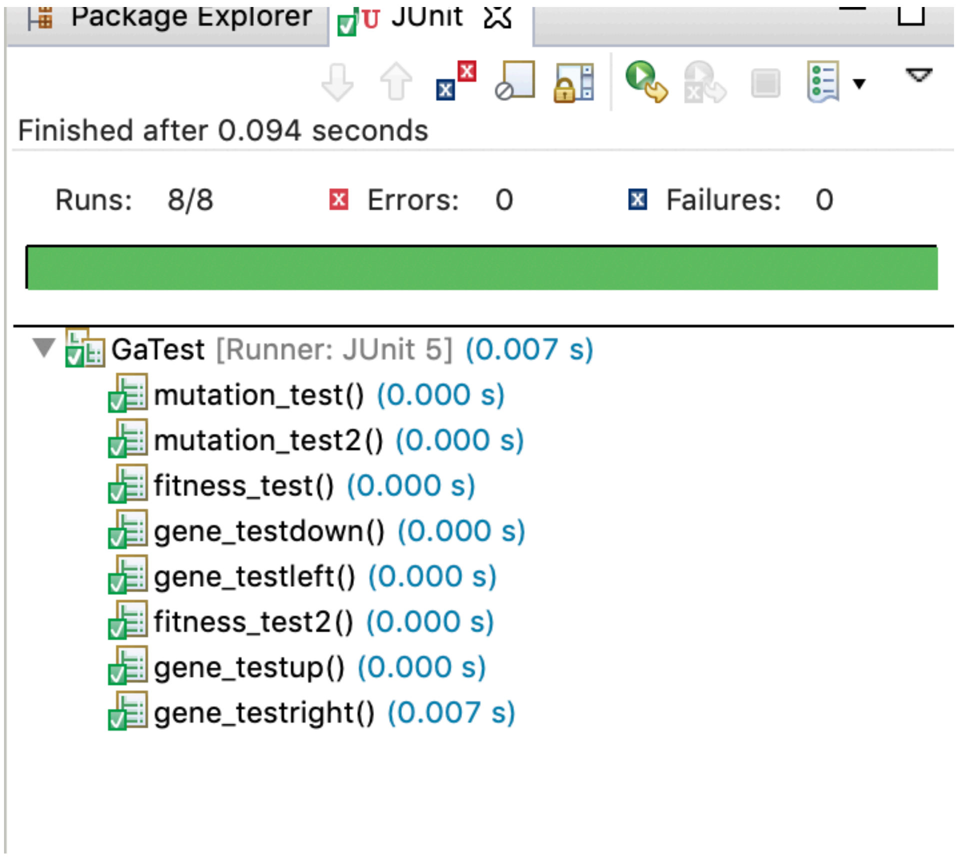
Besides, our group also test the mutation function and the fitness function.By giving several instances to check if the actual result equal to the expect result.Mutation test simply test that if the mutated group number equal to the opposite number of given.The fitness unit test function checks the value of the given array whether equal to the real calculation result.

Code:



```java
    @Test //fitness test
    void fitness_test() {
        String filePath = "data/maze1.txt";
        // initial number of individuals
        int initSetsNum = 1000;

        GA tool = new GA(filePath, initSetsNum);
        int[] testCode = {1,0,1,0,1,0,1,0,0,0,0,0,0,0,0};
        double result = tool.calFitness(testCode);
        //System.out.println(result);
       Assertions.assertEquals(1.0, result);
}
    @Test //fitness test
    void fitness_test2() {
        String filePath = "data/maze4.txt";
        // initial number of individuals
        int initSetsNum = 1000;

        GA tool = new GA(filePath, initSetsNum);
        int[] testCode = {0,0,1,1,0,0,0,0,1,1,1,1};
        double result = tool.calFitness(testCode);

       Assertions.assertEquals(1.0, result);
}

    @Test //mutation test
    void mutation_test() {
        int[] test_data = {0,1};

        int[] result=  Tools.mutation(test_data);

        assertTrue(Arrays.equals(test_data, result));

    }



    @Test //mutation test
    void mutation_test2() {
        int[] test_data1 = {0,0};
        int[] result1=  Tools.mutation(test_data1);
        assertTrue(Arrays.equals(test_data1, result1));
      }

```

screenshot of the Unit Test:



```
Finished after 0.094 seconds

Runs:   8/8        Errors:   0        Failures:   0

GaTest [Runner: JUnit 5] (0.007 s)
    mutation_test() (0.000 s)
    mutation_test2() (0.000 s)
    fitness_test() (0.000 s)
    gene_testdown() (0.000 s)
    gene_testleft() (0.000 s)
    fitness_test2() (0.000 s)
    gene_testup() (0.000 s)
    gene_testright() (0.007 s)
```

Summarize

In this project, we used genetic algorithm to find the shortest path of a maze. The total step number(shortest step number) must be given since our calculation of fitness is based on the distance between end position and exit point. So the existence of the shortest path must be guaranteed first then our algorithm could find it.

After the study of this project, we find that there's still so much to learn about genetic algorithms.