Optimal Path for Traveler

---- Application of Genetic Algorithm

Qincheng Luo 001443834

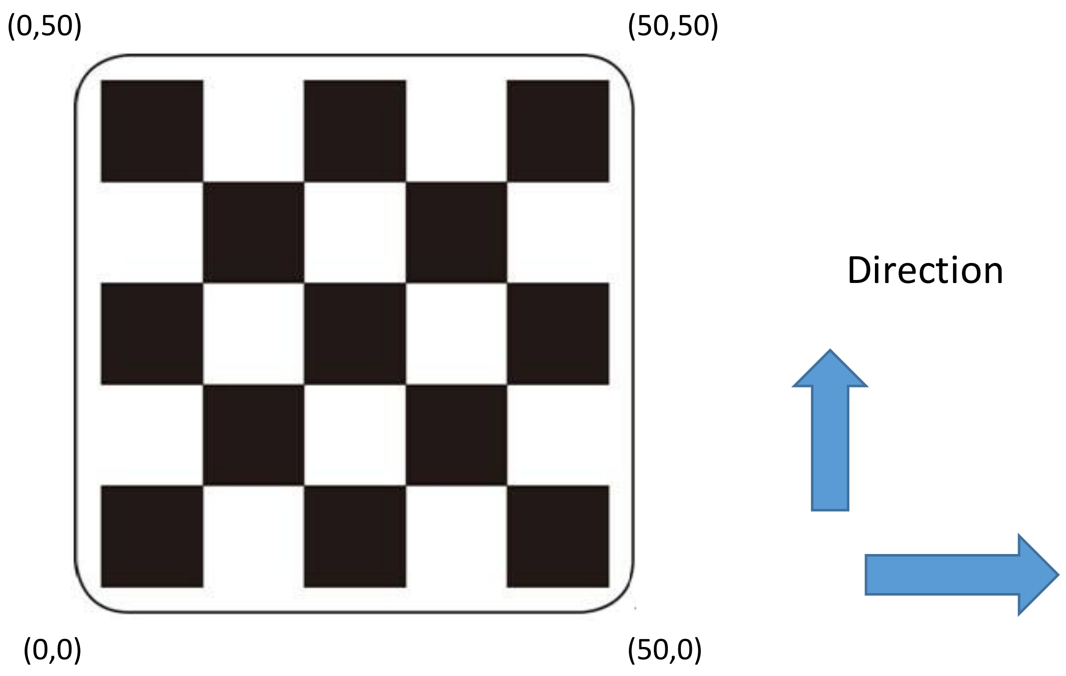
Guanrui Wang 001498692

Sida Chen 001409406

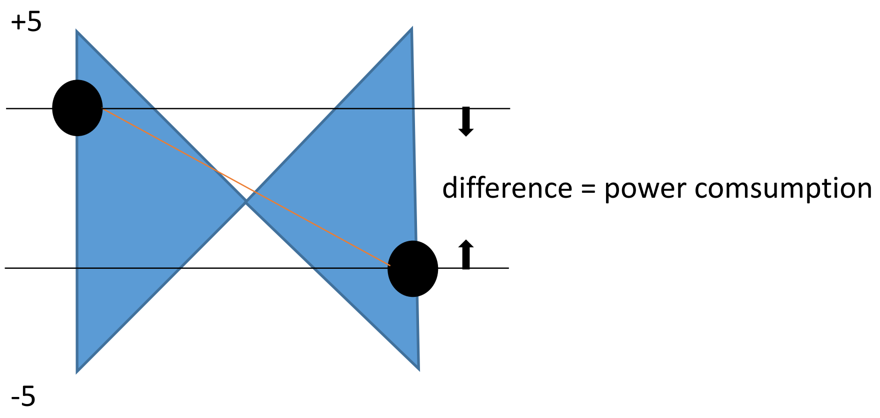
1. Introduction

Genetic Algorithm (GA) is a computational model that simulates the natural evolution of Darwin's biological evolution theory and the biological evolution process of genetics. It is a method to search for optimal solutions by simulating natural evolutionary processes. The genetic algorithm targets all individuals in a group and uses randomization techniques to guide efficient searching of a coded parameter space.

In our project, suppose there is a two-dimensional array space. Our goal is to reach the point in the upper right corner (50, 50) from the lower left corner (0,0). And then find the most optimal path with least power consumption.



Each point corresponds to a random slope. Thus, one point to another point has a random slope difference of (-5) to (+5). Turning over the slope will have a certain amount of physical exertion, and the absolute value of the slope difference is equal to the individual's total power consumption.



2. Model Design

2.1 Environment design

Initialize a 50 * 50 environment and initialize a population of 10 random individuals.

2.2 Gene design

Because the individual can only choose to go up or to the right, we can record “1” to genome when we move towards to the right; record “0” to genome when we move towards to up. In order to reach (50, 50) from (0, 0), each must advance a total of 50 squares to the top, and a total of 50 grids to the right. In this way, we can clearly see its path and treat the path as a genetic sequence through special recording methods.

2.3 Fitness design

As we said above, each point corresponds to a random slope value. From one point to another, there will be a slope difference, which corresponds to the individual's physical consumption per point of movement. In this way, we can easily calculate the total power consumption of each individual, at the same time, we can also calculate the total power consumption of the individual in this population.

Fitness reflects the adaptability of a species to its living environment. Fitness in our final project is the sum of power consumption of a population. Less power consumption means better fitness. Our goal is to find the least power consumption and best fitness.

2.4 Crossover design

Evolution refers to the gradual adaptation of the population to its living environment and the continuous improvement of its quality. In this process, there are two particularly important stages, one of which is the crossover. Before the crossover, we design to choose fittest individual with least power consumption and second fittest individual. Because they are best two in population. Firstly, we generate two random numbers. According to the sequence, we pick up this segment of gene from best individual and keep it. The generation should have the same start point and end point with best parents. The other gene is to ensure the least power consumption at the basic of the same points.

2.5 Mutation design

The five pairs of genes (5 pairs of 0-1 exchanges) of the new individuals have a 30% chance of mutation, and the new individuals will replace the ones with the highest and worst physical expenditure in the original population. This will produce the next generation.

As for the reason why we need to to exchange gene “0” with gene “1” is that we need to ensure that individual can reach final point (50,50). No matter what mutation or crossover, we are always keeping the number of gene “0” and gene “1” is completely same and the number strictly equals to 50. This is the normal operation of practical application.

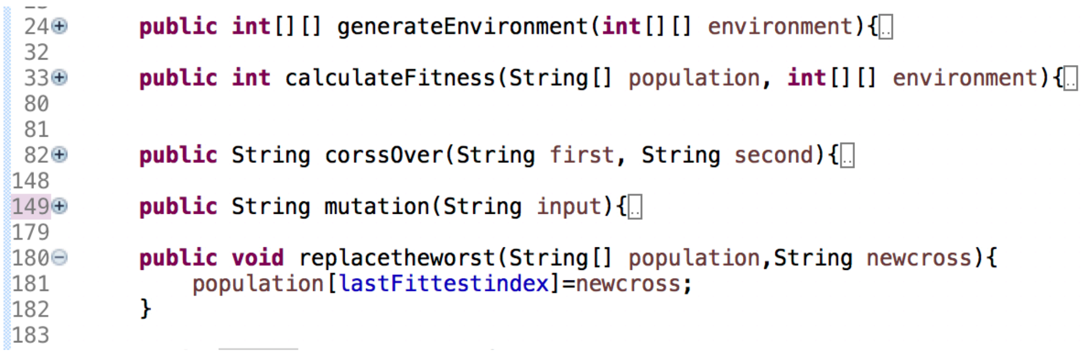
2.6 Multi-threading design

In our project, we have actively applied multithreading. Multithreading is used to improve the efficiency of program operation. We add total 5 threads in our project, in other word, we establish 5 populations with 10 random individuals. Each thread runs individually. Once any one of the total threads find the fittest gene, this program stops and the other thread stops as well. Because in single-threaded programming, we sometimes encounter a practical problem that after too many generations of evolution, we still can't find the best genes. So our program has to stop. We need to run our program again. Then multi-threading can effectively improve the efficiency of the program.

3. Function Implementation

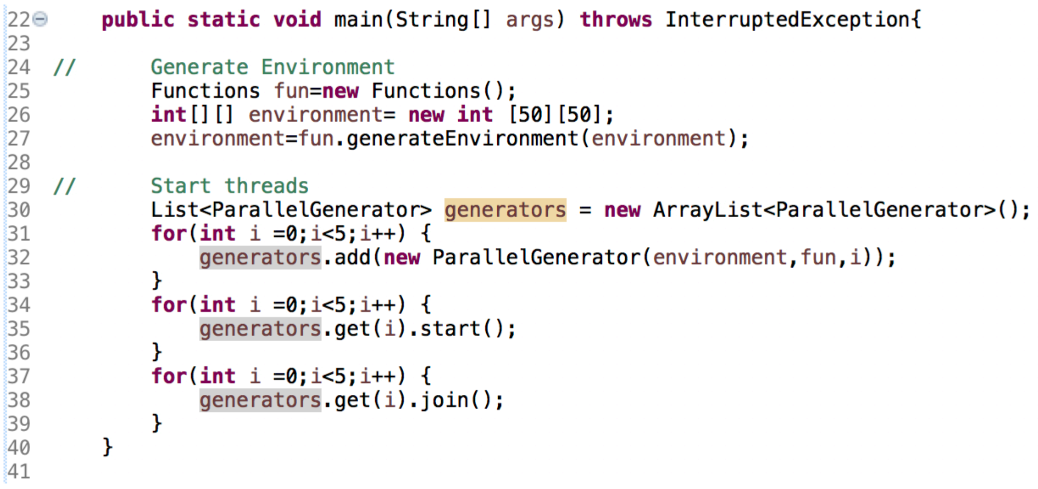
Fiction lists:

3.1 generateEnvironment
3.2 calculateFitness
3.3 crossover
3.4 mutation
3.5 replacetheworst



```java
    public int[][] generateEnvironment(int[][] environment){..

    public int calculateFitness(String[] population, int[][] environment){..


    public String corssOver(String first, String second){..

    public String mutation(String input){..

    public void replacetheworst(String[] population,String newcross){
        population[lastFittestindex]=newcross;
    }

```

3.6 Main
3.7 Multi-threads

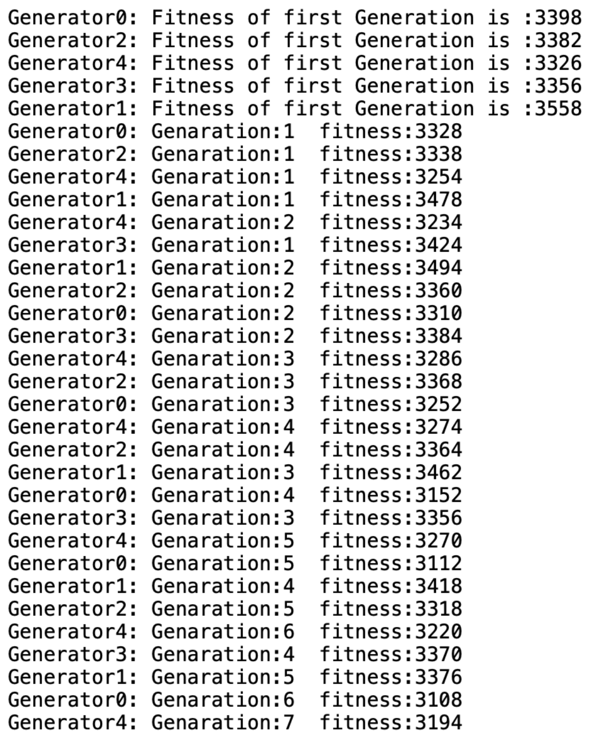


```java
    public static void main(String[] args) throws InterruptedException{

//      Generate Environment
        Functions fun=new Functions();
        int[][] environment= new int [50][50];
        environment=fun.generateEnvironment(environment);

//      Start threads
        List<ParallelGenerator> generators = new ArrayList<ParallelGenerator>();
        for(int i =0;i<5;i++) {
            generators.add(new ParallelGenerator(environment,fun,i));
        }
        for(int i =0;i<5;i++) {
            generators.get(i).start();
        }
        for(int i =0;i<5;i++) {
            generators.get(i).join();
        }
    }

```

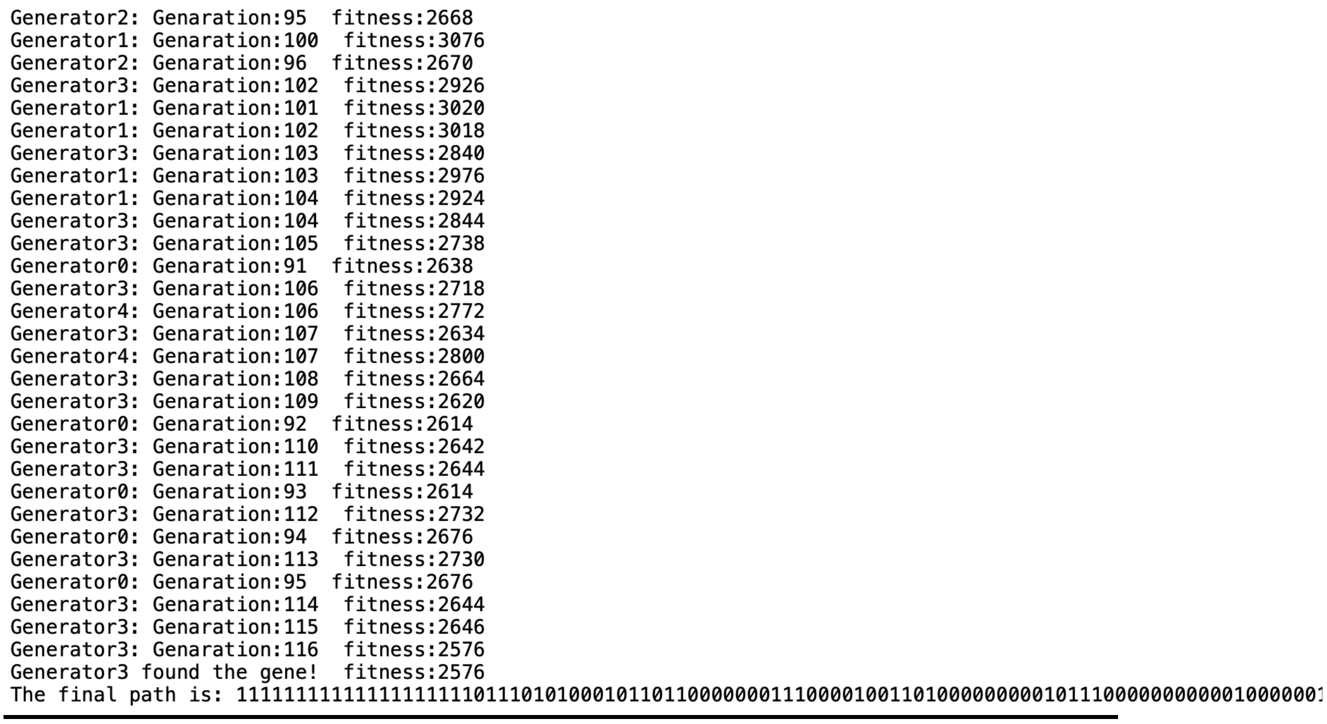
4. Results

4.1 Successfully find the gene

screen-shots:



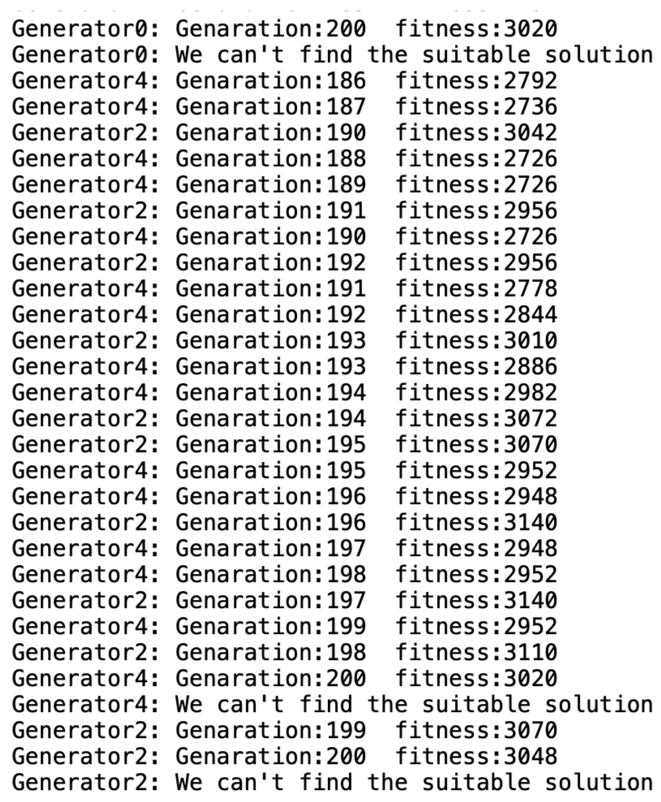
```
Generator0: Fitness of first Generation is :3398
Generator2: Fitness of first Generation is :3382
Generator4: Fitness of first Generation is :3326
Generator3: Fitness of first Generation is :3356
Generator1: Fitness of first Generation is :3558
Generator0: Genaration:1  fitness:3328
Generator2: Genaration:1  fitness:3338
Generator4: Genaration:1  fitness:3254
Generator1: Genaration:1  fitness:3478
Generator4: Genaration:2  fitness:3234
Generator3: Genaration:1  fitness:3424
Generator1: Genaration:2  fitness:3494
Generator2: Genaration:2  fitness:3360
Generator0: Genaration:2  fitness:3310
Generator3: Genaration:2  fitness:3384
Generator4: Genaration:3  fitness:3286
Generator2: Genaration:3  fitness:3368
Generator0: Genaration:3  fitness:3252
Generator4: Genaration:4  fitness:3274
Generator2: Genaration:4  fitness:3364
Generator1: Genaration:3  fitness:3462
Generator0: Genaration:4  fitness:3152
Generator3: Genaration:3  fitness:3356
Generator4: Genaration:5  fitness:3270
Generator0: Genaration:5  fitness:3112
Generator1: Genaration:4  fitness:3418
Generator2: Genaration:5  fitness:3318
Generator4: Genaration:6  fitness:3220
Generator3: Genaration:4  fitness:3370
Generator1: Genaration:5  fitness:3376
Generator0: Genaration:6  fitness:3108
Generator4: Genaration:7  fitness:3194
```



```
Generator2: Genaration:95  fitness:2668
Generator1: Genaration:100  fitness:3076
Generator2: Genaration:96  fitness:2670
Generator3: Genaration:102  fitness:2926
Generator1: Genaration:101  fitness:3020
Generator1: Genaration:102  fitness:3018
Generator3: Genaration:103  fitness:2840
Generator1: Genaration:103  fitness:2976
Generator1: Genaration:104  fitness:2924
Generator3: Genaration:104  fitness:2844
Generator3: Genaration:105  fitness:2738
Generator0: Genaration:91  fitness:2638
Generator3: Genaration:106  fitness:2718
Generator4: Genaration:106  fitness:2772
Generator3: Genaration:107  fitness:2634
Generator4: Genaration:107  fitness:2800
Generator3: Genaration:108  fitness:2664
Generator3: Genaration:109  fitness:2620
Generator0: Genaration:92  fitness:2614
Generator3: Genaration:110  fitness:2642
Generator3: Genaration:111  fitness:2644
Generator0: Genaration:93  fitness:2614
Generator3: Genaration:112  fitness:2732
Generator0: Genaration:94  fitness:2676
Generator3: Genaration:113  fitness:2730
Generator0: Genaration:95  fitness:2676
Generator3: Genaration:114  fitness:2644
Generator3: Genaration:115  fitness:2646
Generator3: Genaration:116  fitness:2576
Generator3 found the gene!  fitness:2576
The final path is: 111111111111111111011101010001011011000000001110000100110100000000010111000000000000010000001
```

4.2 Unsuccessfully find the gene

screen-shots:



```
Generator0: Genaration:200  fitness:3020
Generator0: We can't find the suitable solution
Generator4: Genaration:186  fitness:2792
Generator4: Genaration:187  fitness:2736
Generator2: Genaration:190  fitness:3042
Generator4: Genaration:188  fitness:2726
Generator4: Genaration:189  fitness:2726
Generator2: Genaration:191  fitness:2956
Generator4: Genaration:190  fitness:2726
Generator2: Genaration:192  fitness:2956
Generator4: Genaration:191  fitness:2778
Generator4: Genaration:192  fitness:2844
Generator2: Genaration:193  fitness:3010
Generator4: Genaration:193  fitness:2886
Generator4: Genaration:194  fitness:2982
Generator2: Genaration:194  fitness:3072
Generator2: Genaration:195  fitness:3070
Generator4: Genaration:195  fitness:2952
Generator4: Genaration:196  fitness:2948
Generator2: Genaration:196  fitness:3140
Generator4: Genaration:197  fitness:2948
Generator4: Genaration:198  fitness:2952
Generator2: Genaration:197  fitness:3140
Generator4: Genaration:199  fitness:2952
Generator2: Genaration:198  fitness:3110
Generator4: Genaration:200  fitness:3020
Generator4: We can't find the suitable solution
Generator2: Genaration:199  fitness:3070
Generator2: Genaration:200  fitness:3048
Generator2: We can't find the suitable solution
```

5. Unit Test

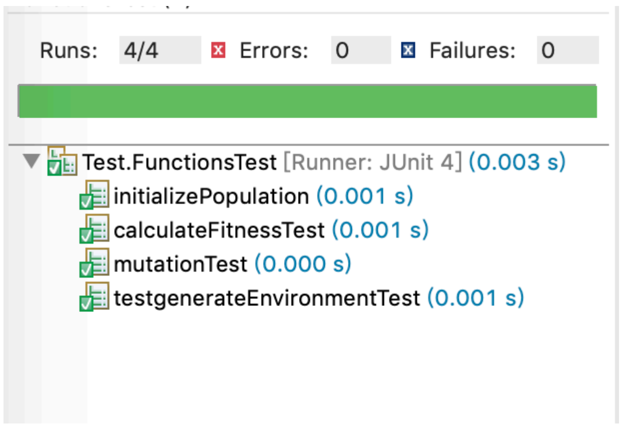
5.1 initializePopulation

5.2 calculateFitnessTest

5.3 mutationTest

5.4 testgenerateEnvironment

screen-shots:



```
Runs: 4/4    Errors: 0    Failures: 0

Test.FunctionsTest [Runner: JUnit 4] (0.003 s)
    initializePopulation (0.001 s)
    calculateFitnessTest (0.001 s)
    mutationTest (0.000 s)
    testgenerateEnvironmentTest (0.001 s)
```

6. Conclusion

In our project, we can easily find the the average fitness value in the first generation is around 3500. In the evolution of early generations, the fitness value is unstable. The improvement of fitness value is not remarkable. After continuous operation and testing, the final generation number is maintained around 80. And the final fitness value is approximately between 2400-2700. By the limitation of our project, we set the condition of fitness as 2600. The condition 2600 is tested by us continuously when the generation number is around 200. When the final fitness value is less than 2600, and we stop generating. And this generation is our result. Of course, this results is not always stable because regardless of initial values, slope, selection, mutation, etc., these data are random and uncontrollable.

However, there is an important conclusion that with the continuous evolution of the population, the population gradually adapts to the living environment, and the quality is constantly improved. Generally speaking, we can always find the best solution to solve practical problems. Our project is also fully in line with Darwin's natural selection and genetic mechanisms of biological evolution. Such a theoretical point of view also has a very important enlightening role in various scenarios in real life.