**Evolution with Genetic Algorithm**

**Abstract**

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution.

1. **Overall**

In the Real world, living beings changed generation by generation. In computer world, natural selection can be modeling by genetic algorithm. By GA we can find that after adjusting to the environment how mutation and crossover changed the chromosome.

This project aims to know how Chromosome changed and after adjust fitness, what is every Generation's best chromosome and current chromosome. By check function value we can find that the difference between best and current, also we can know the difference between first generation and the other generations.

In our project, we generate the data of 100 groups with each group size of 500 to show verify our algorithm. The structure of the project is showing below with the diagram.

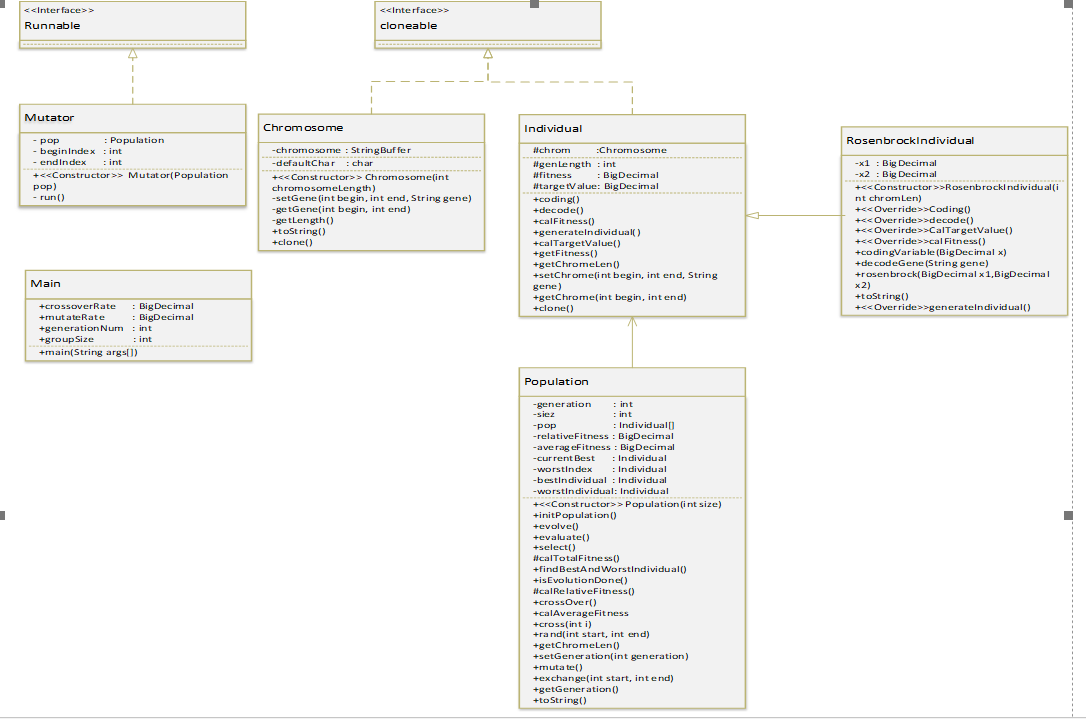


Figure 1.1: UML diagram

BigDecimal was used to replace the double type in our project so that we can scale our result precisely. Although BigDecimal can make the result more accurate, the space and the time it cost is large. To make the project running faster, we run multiple threads while generating the population, specifically in the mutating process. Also because of the cost of the BigDecimal, we rarely create the array list to save the space as more as possible.

1. **Working flow**
   1. Initialization

When starting the project, the JVM will first excecute the main method in the Main class. To start the evolution, we first initiate a group of individuals and store them in the variable pop. The initpopulation method would generate individual step by step to make sure all the individuals are valid as well as contain gene and chrom. Below is the shortcut of the initpopulation function.



Figure 2.1: initPopulation method

The step of generating an individual is coding, calculate the target value of the project and calculate the fitness degree of that particular individual. This step is try to make the program more like the real world evolution.

* 1. Evolution

Selection: Generating child population using roulette wheel method.

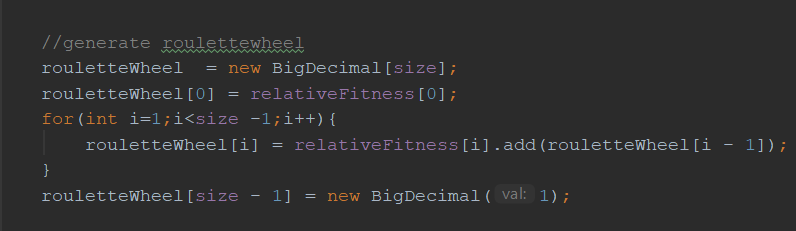


Figure 2.2.1: Generating the roulette wheel

Typically speaking, in nature, the more adapt to the individual the more likely they are to breed. Besides the normal roulette wheel method, we can also maintain the best solution found over time before selection. This operation would be the next step of our project in the future.

Crossover: Swap and exchange single point gene.

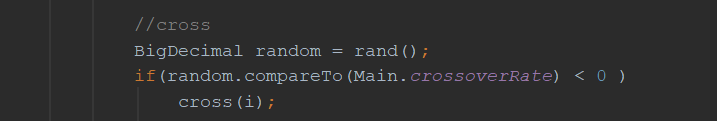


Figure 2.2.2: Cross the single point

**Recombination is mainly implemented by exchanging the binary code of different genes.** Binary coded genetic exchange process is very similar to speak of the association of homologous chromosomes in the high school biology process - random put several of them in the same location coding to exchange, to produce new individual.

Mutation: Mocking real-world mutation process by using 8 threads.

Figure 2.2.3: Mutation

Using mutation operator in groups, and the next generation of group is obtained by the above operation. Changing the individual coding string value of certain genes, so as to form a new individual.

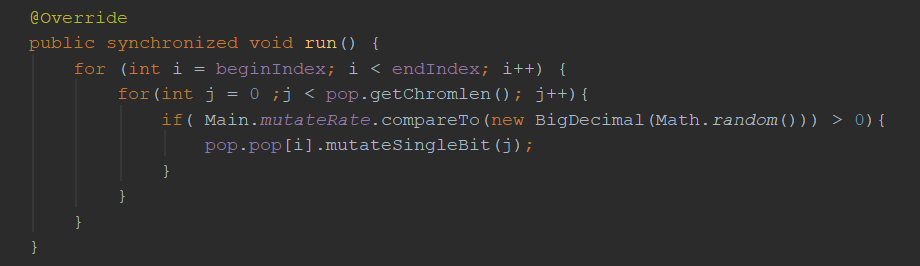


Figure 2.2.4: Synchronized method

To make sure all threads are safe, we choose synchronize as the key word of the run function in the mutator class. However this may lead to a latency of the program so we tried to declare the sharing variable as volatile rather than using synchronized section. The difference of the performance is below.

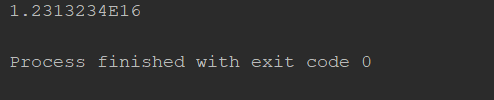


Figure 2.2.5: Average time for mutating with synchronize keyword

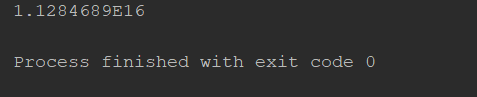


Figure 2.2.6: Average time for mutating without synchronize keyword

Above images are differences with and without synchronized keyword. The time is the average time of 100 times of running the mutate function.

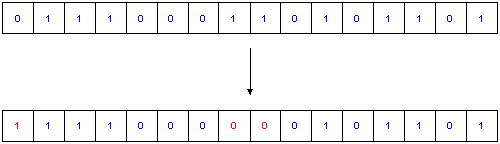


Figure 2.2.7: Before and after mutation

Figure 2.2.7 shows an example of the mutation process to help reader understand the process concisely.

**3. Testing and outcomes**

3.1 Testing.

Testing is extremely required to point out the defects and errors that were made during the development phases. It's essential since it makes sure of the Customer's reliability and their satisfaction in the application. It is a crucial part for the quality of the project.

The testing framework of the project is mainly using junit4.1.3 which is a multifunctional unit test framework in Java. We build different test cases regards different classes and methods. Results of test cases for the whole project is showing below.

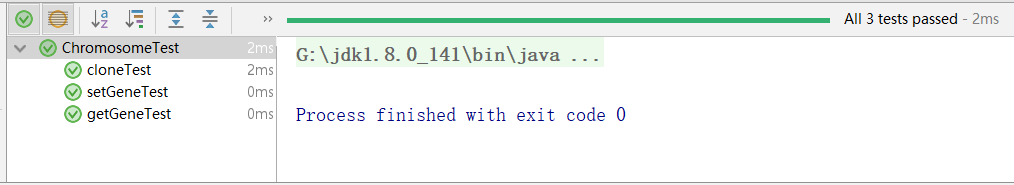


Figure 3.1.1: Chromosome Test

There are three different kinds of methods in the Chromosome Test class each of which is related to a method that is in the Chromosome class. Chromosome Test is to make sure that

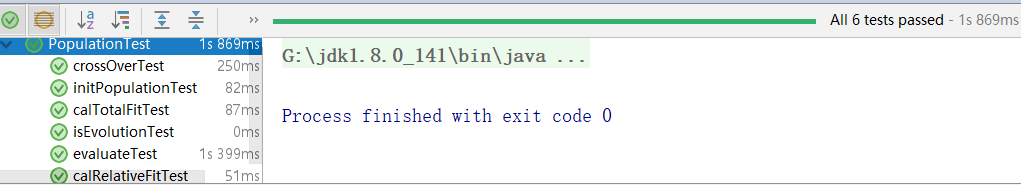
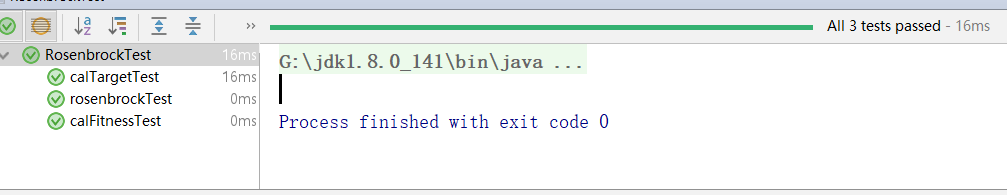


Figure 3.1.2: Population Test

This class contains six kinds of method. initPopulateionTest is to test the initialization of each population in the project which is obviously important for the whole program. It is hard to measure most of the method by using unit test since some of methods are void types. It is impossible to use whether equals function or the true/false function to decide whether the specific method is running correctly. So our solution is to figure out what is the outcome or effect of this method and we test the result to finally verify whether the program is running correctly.

Figure 3.1.3: Rosenbrock Test

This class contains 3 methods with multiple test cases to make sure the Rosenbrock class is working well when executing it.

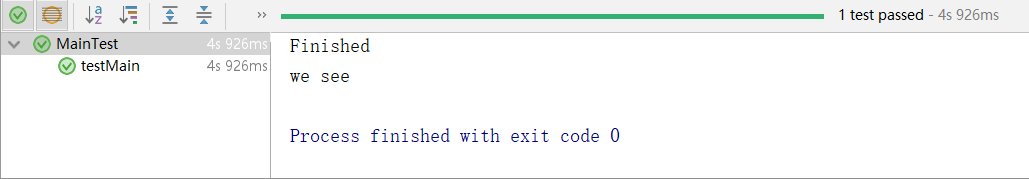


Figure 3.1.4: Main Test

This test is mainly used to make sure that the main method can be executed by JVM correctly as well as the IO stream works well. In case of the missing of the file or missing of the file instance, this process is necessary.

3.2 Results

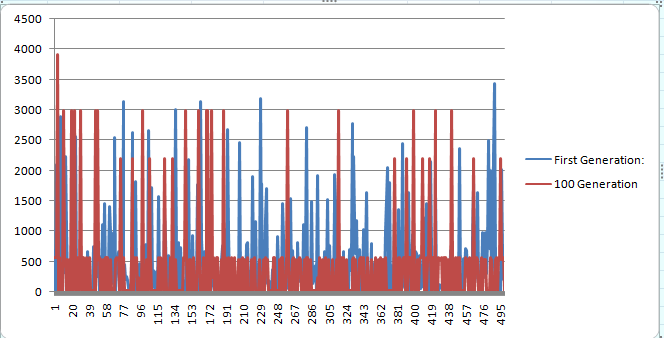


Figure 3.2.1 Outcomes and statistics.

In one generation what the best should be and what the current best is. We can see in next table:

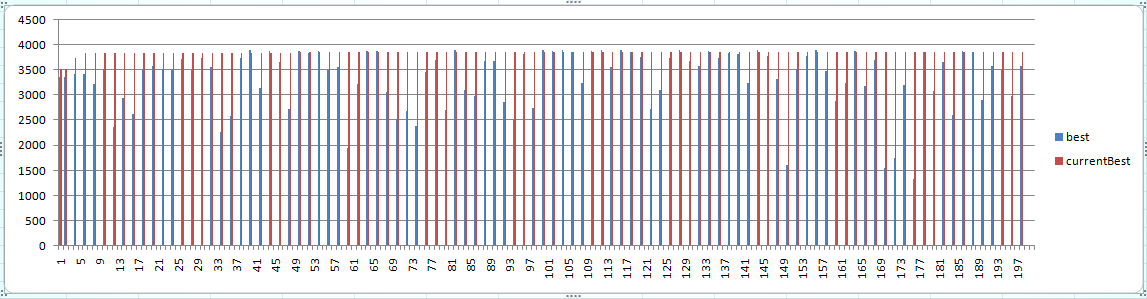


Figure 3.2.2 Performance of the project

Image above illustrated the best and current best individual for every single generation. Our program used the rosenbrock method to get the best value among all possible values which is significant effective. More result details are in the result.txt which is already attached in the zip file.

1. **Conclusion:**

Traditional methods of function optimization based on gradient, for those who have good function for gradient information, the traditional optimization method is optimal. Obviously the rosenbrock method has a better performance as well as the multithread way to generate population is effective. The next step of the project in the future would be optimizing the selection function and the crossover process. The time performance is great now but the space complexity still can be improved.