Genetic Algorithm——Knapsack Problem

**Problem statement:**

A classic problem of genetic algorithm is Knapsack problem.

The knapsack problem is a problem in combinatorial optimization. For example, we are given a set of items, each item has a weight and a value, we have to determine the number of each item and to include in collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

If you are given the following statics and a bag with 10 pounds capacity, you can figure out the optimized combination in a short time. “Right, It’s 1 plus 5.”

|  |  |  |  |
| --- | --- | --- | --- |
| Weight(Pounds) | 1 | 10 | 5 |
| Value(Dollars) | 100 | 20 | 10 |

But what if you are given more complicated statics with a billion different combinations? I’m afraid you can tell the answer immediately. However, by using genetics algorithm we can solve this situation easily.

**Implementation:**

**Genetic code**: I encapsulated the gene (a Boolean array) and fitness (a double primitive) value in “Chromosome” class, and use an array of chromosome to represent for populations.

In the gene array, it’s an boolean array, so there are only two values: true and false, true represents for the object is selected in the chromosome class, in the opposite, false represents for not selected. Its value is generated by a random method.

Fitness is a double value which is the fitness value of the gene.

As to the data structure for population, I encapsulated the gene and fitness value in “Chromosome” class at first, and add chromosome into arraylist. But I found arraylist waste too much time on checking if resize is needed. However, my population number is fixed, so I need an array.

**Gene expression:** if the boolean value of a position is true, the object same to this position is selected and putted into the knapsack, otherwise not.

**Fitness function:** I implemented the fitness function by the logic of knapsack problem, by using a for loop to check if an object is selected, if the value is true, then the function adds the weight and value of the object into different variables, when the weight exceeded the capacity, you return 0 as its fitness which means it will be regenerated later. If the weight didn’t exceed the capacity, the function returns the value.

**Crossover Function:** I use random function to select parents from the parent population and loop from the first gene position to the last one, use an if condition to check if they should do the crossover.

Position: 0 | 1 | 2 | 3 | 4 | 5 | 6

Parent1: 1 | 0 | 1 | 1 | 1 | 0 | 0

Parent2: 1 | 1 | 0 | 1 | 0 | 0 | 1

If the random double smaller than the crossover rate (defined in my constructor and is 0.8 which is best suitable according to the material on the Internet), then I will do the crossover. For example, position 1,2,3,4 and 6 matches the condition.

Position: 0 | 1 | 2 | 3 | 4 | 5 | 6

Child1: 1 | 1 | 0 | 1 | 0 | 0 | 1

Child2: 1 | 0 | 1 | 1 | 1 | 0 | 0

**Mutation Function:** Similar to the crossover function, firstly I loop from the first individual to the last one, and use if to check if they should mutate, if yes, then another loop to check which specific gene should mutate.

**Evolve Function:** Initialize the population first, then loop until find the best solution or reached the max generation limitation. In the loop, there is procedures like calculate fitness, record best solution, replace the worst solution with the best one, select parent population, crossover and mutation

**Improvements:**

After I finished my first draft of source code, the output is not very satisfying, with the best value around 27000, it only outputted 20000 instead, which is not good enough. So, I took the following steps and made very huge progresses:

Firstly, I create a replace function, which replace the worst individual (not satisfy the capacity limitation) with the best individual, the reason why I did not just create a new random individual is that by doing this could improve the chance of find best solution and the speed of convergence.

Secondly, I made some improvements to the select function. A sort process is added into the select function, and then I add the top 1 percent individuals to the parent population (we use this to generate new generation). After this, I use roulette wheel selection to select the rest individuals.

Thirdly, I rewrote the crossover function. My old crossover function is not perfect for the knapsack problem, and it was a drag on my algorithm. So, I searched the Internet and found a better way. Instead of crossover in an interval, my new function choose position randomly and switch it with another random chose individual in the parent population. This way is especially suitable for knapsack problem.

Fourthly, I made some arguments adjustments. This is very important and could help improve the algorithm performance. I set an extremely small capacity value at first, and my laptop keeps looping because almost every solution exceeded the capacity due to the small capacity, so it has to reproduce a new solution which satisfy “if condition”. After several adjustments I found the best arguments for my algorithm.

**Conclusions:**

Firstly, I ran my algorithm with only five objects ten times, and every time it gives me the same output which is the best solution:190. The capacity is 100, and the gene of the best solution is 10110 in the following circumstance.

(value:{50,30,60,80,20} weight:{35,40,40,20,15})

Then, I ran my algorithm with bigger statistics, and the outputs is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Object Number | 20 | 50 | 100 |
| Best Fitness/Value | 1043 | 3087 | 26922 |
| Weight | 497 | 933 | 6743 |
| Generation | 11 | 84 | 56 |
| Capacity | 850 | 1000 | 6750 |
| Population Number | 500 | 500 | 500 |
| The theoretical best fitness | 1043 | 3096 | 27000 |

First row: the number of objects, it equals to the length of my gene code.

Second row: the fitness of the best solution.

Third row: how much does the best solution weight.

Forth row: the generation that get the best fitness. (not the total generation that has been evolved)

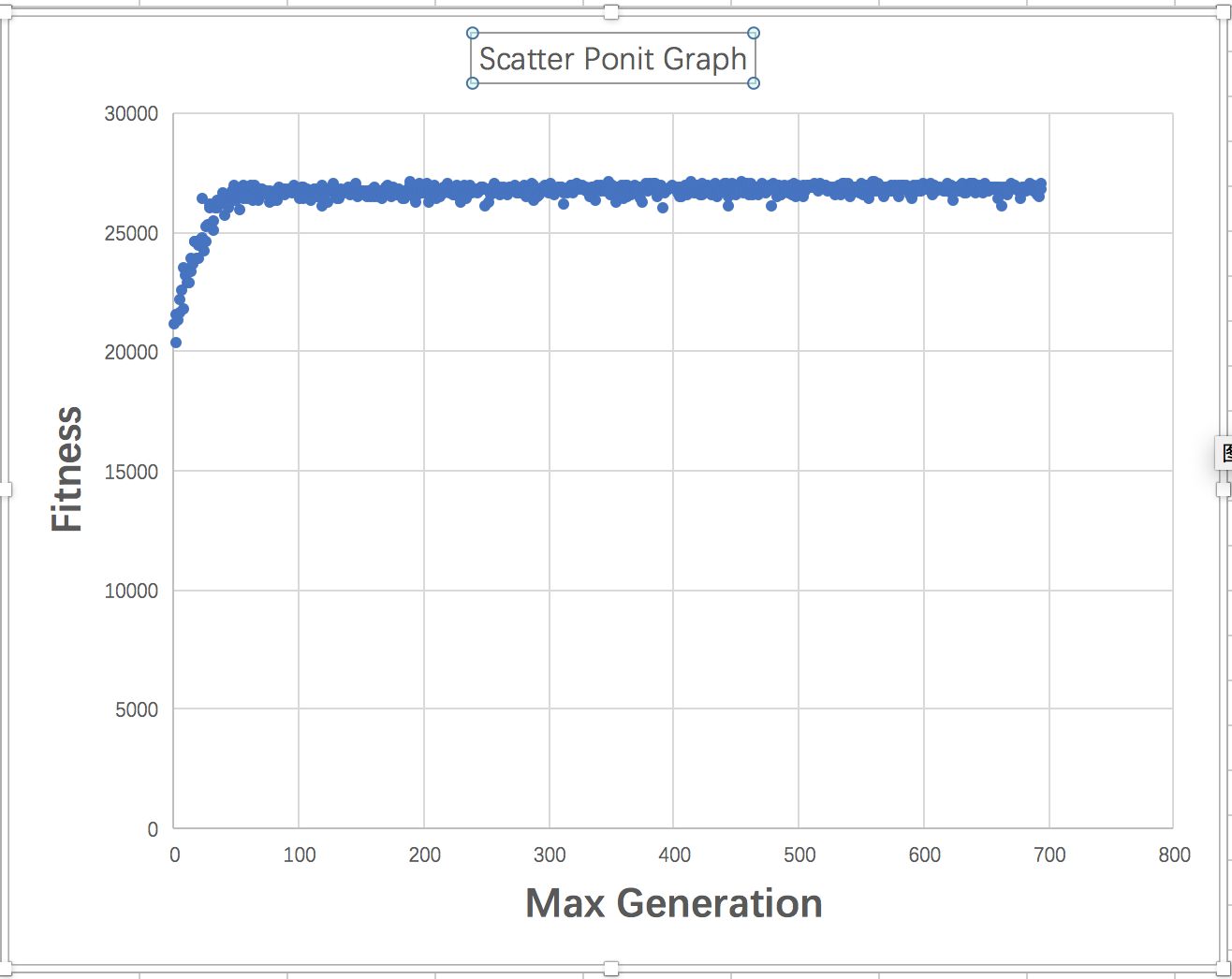
Fifth row: the capacity of a bag or knapsack that used to contain objects.

Sixth row: the number of the population.

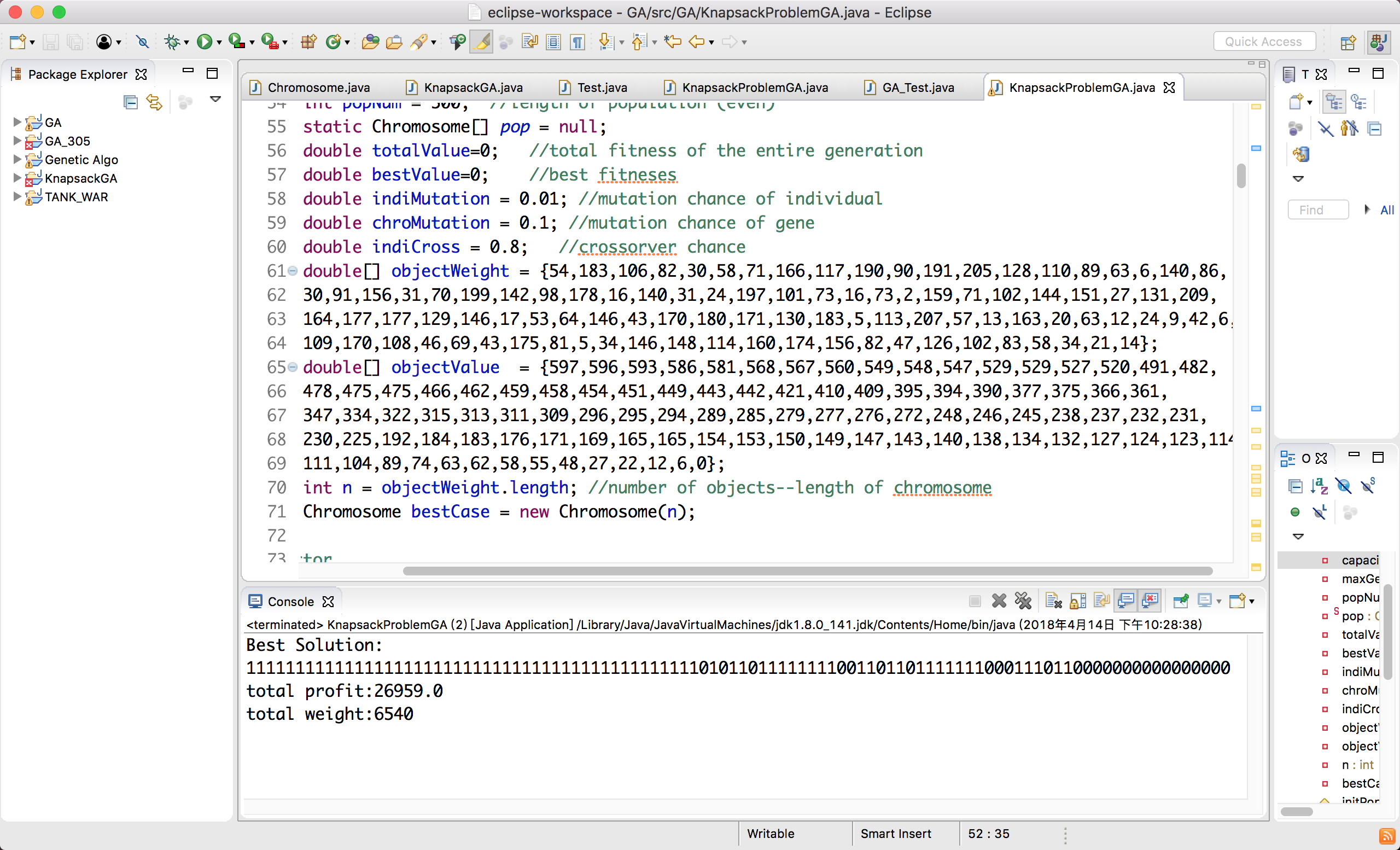
Seventh row: Due to all the numbers are written by myself, and I specifically designed some number with small weight and big value, so I could tell the best solution.

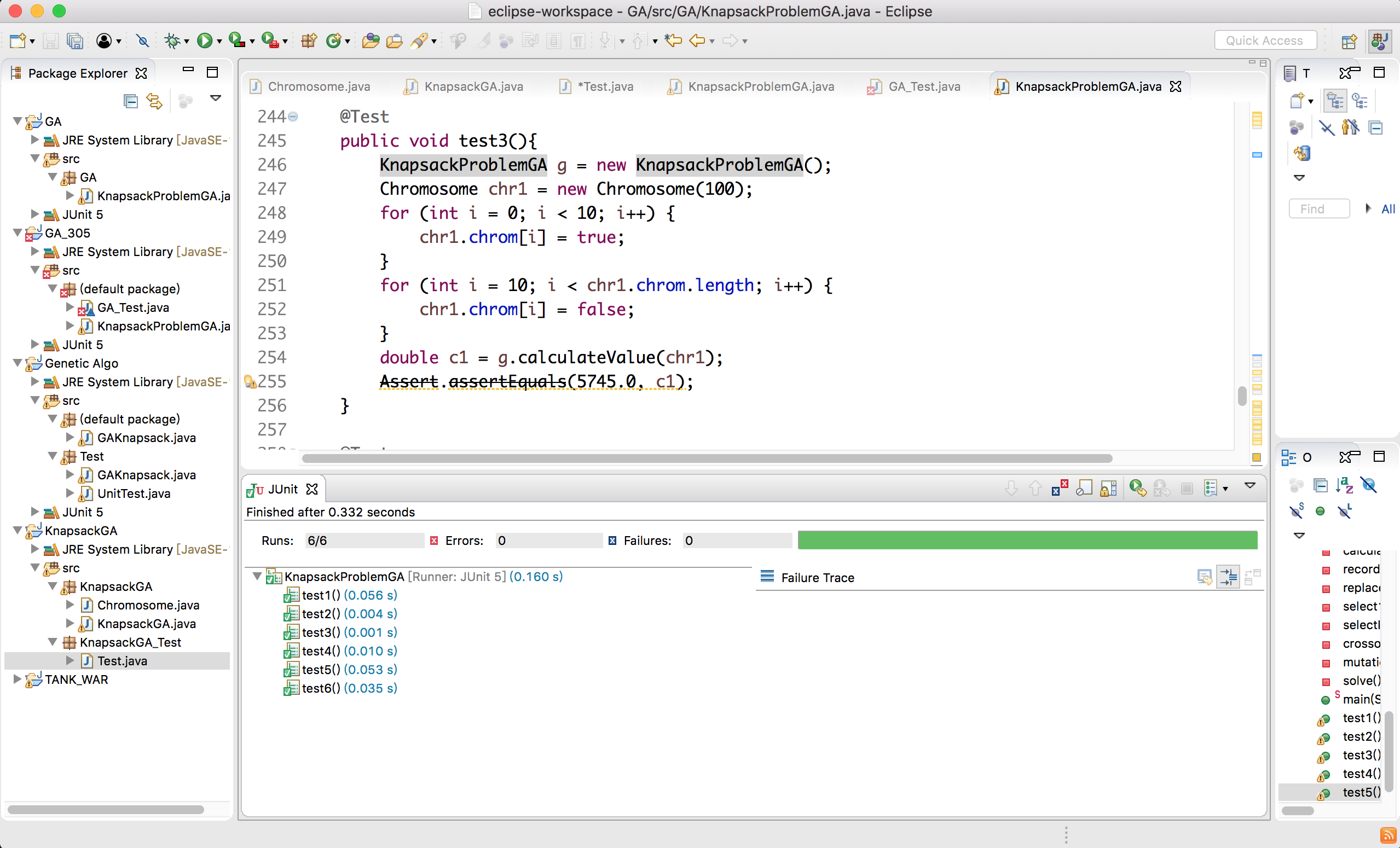
As we can see, the best fitness is very close to the theoretical best fitness.

Thirdly, I start from 1 generation and end at 700 generation to see the develop of the best fitness, due to long time of running, I stopped it after 700 generation, and only take two points at 1000 generation and 2000 generation instead. They give me the same answer of the best fitness which is around 27000.



Screenshots of unit tests and results.





I tested fitness function with different cases, and checked if the value is right.

Also, I checked the mutation function and see if there is difference in gene code before and after the function called.

As to the crossover function, I checked if there is consistence in gene code before and after the function called.