The program is written in python. It implements Generation Algorithm with two basic classes, which are Individual and Game. Class Individual denotes each potential solution with a chromosome. Class Game has a population of individuals and does the parent-selection and crossover job. I will elaborate as follows:

All the tests were done with a population size of 100 and a generation limit of 10000.

1) Structure of the corresponding chromosome,

It’s an array of 8 positive integers ranging from 0 to 7. Each index indicates the column number of each queen and each number indicates the column number of each queen.

2) Fitness function

Find the number of clashes between two queens in the solution chromosome. Fitness value equals 0 when each queen clashes with the other 7 queens, on the contrary, fitness value equals 28 when there are no clashes happening, which means this chromosome is a real solution.

After a new generation is fully reproduced, we evaluate it by calculating the mean fitness value of the population.

3) Crossover operator consists of parent-selection and crossover

The first move is selecting two parents from the population. Here we introduced a method metaphorically imitating the roulette weighted by fitness value. For the individual of index I, its reproduction rate is:

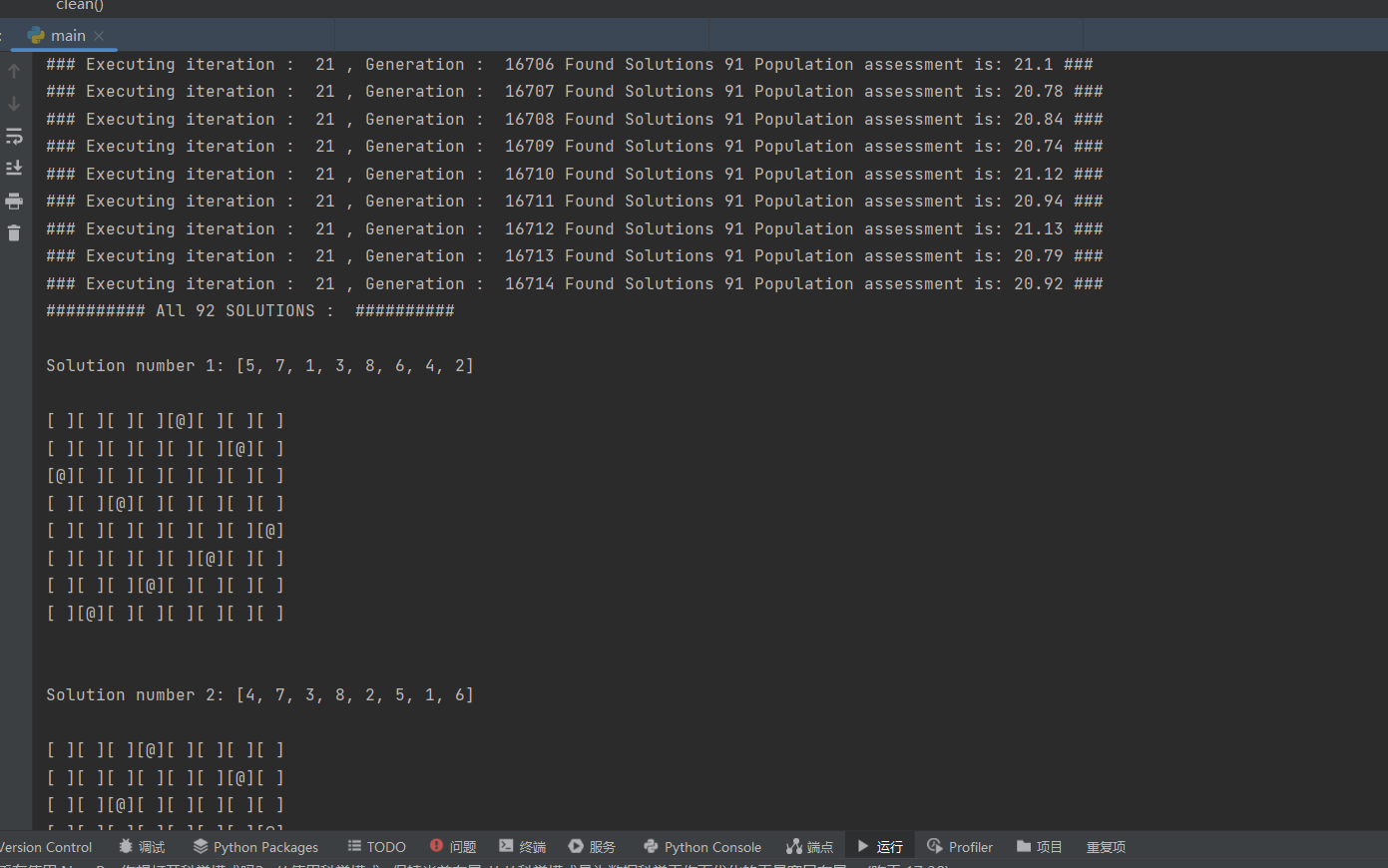
Rprate(i) = fitness(i) / sum(fitniess(k)) where k is the number of the population.

Second, we crossover the two parents at a random cut-off point and generate one or two offspring. The reason we present two versions here is that on the one hand, we found out that in the one-offspring version the algorithm rapidly converges to a high mean fitness value of around 27, which is very close to the real solution. But the program is likely stuck at a local optimum after the first solution is founded. We tuned the application by decreasing the generation limit dramatically to 2000. Through much more iterations, the program could find all 92 solutions in around 10 hours. On the other hand, when we generated two offspring by one crossover, the program avoided being stuck at a local optimum, but the mean fitness value of the population converged at around 20. Which is far beyond our expectations. However, it succeeds to found all 92 solutions in approximately 5 hours.

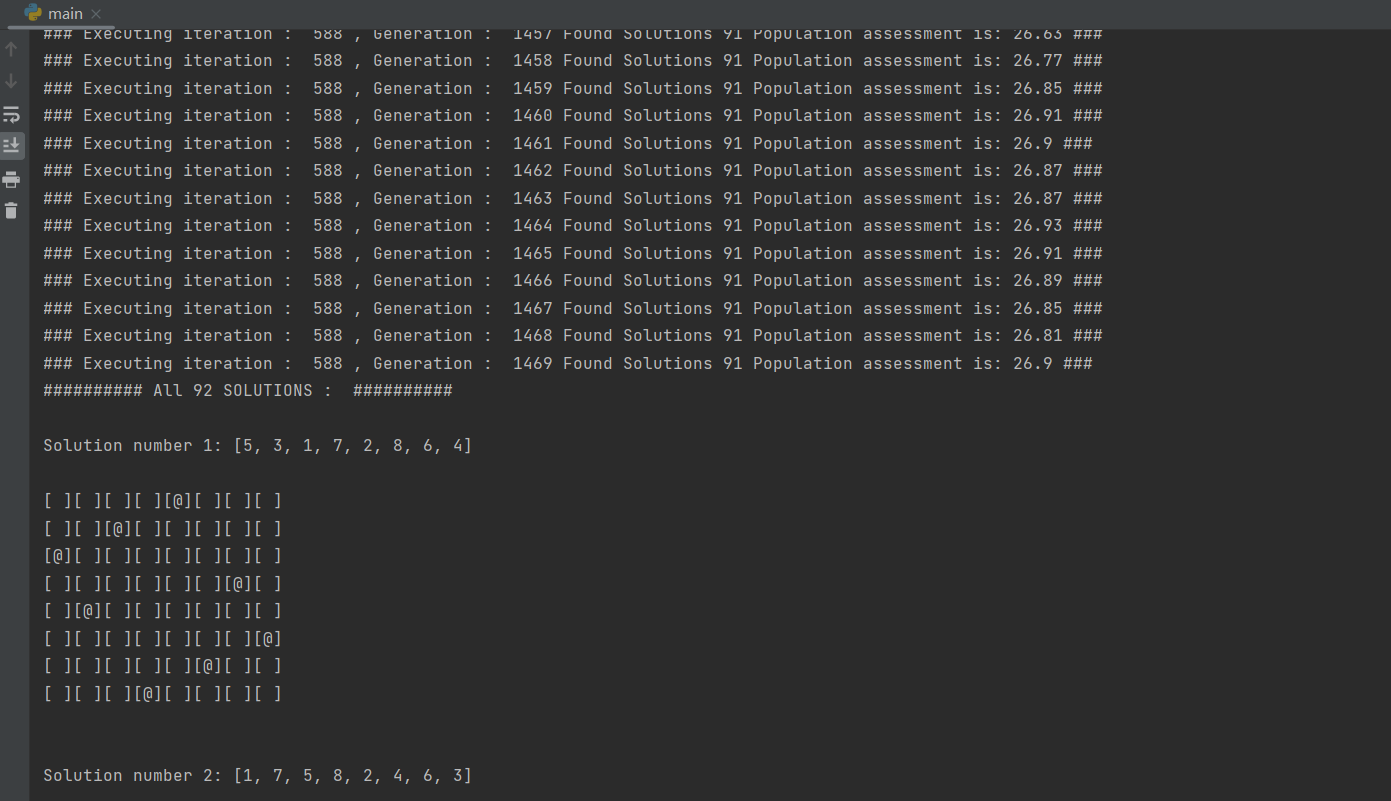
4) Mutation operator.

We set the mutation rate to 0.005.

The mutation function is implemented within the class Individual. For each offspring generated by crossover, we generate a random probability between 0 and 1, when it exceeds the threshold we modify a random gene to a random value in the chromosome array.

****

**Figure 1. The program of 2 offspring versions found all 92 solutions after 21 iterations, each iteration consisting of 20,000 generations. Population = 100, Mutation rate = 0.005, total time = 18765 seconds.**

****

**Figure 2. The program of 1 offspring version found all 92 solutions after 588 iterations, each iteration consisting of 2,000 generations. Population = 150, Mutation rate = 0.005, total time = 41291 seconds.**