4/15/2018

Class Scheduler

Implemented using Genetic Algorithms

Ankit Yadav
NUID- 001271369
PROJECT TEAM- INFO6205_303

Table of Content

Class Scheduler using Genetic Algorithm

1. Introduction

Genetic algorithm is one of the most important tool for making computer evolve to solve a specific type of problems.

One of the major application of GA is found in scheduling events/class/meetings. Class scheduling is a classic NP complete problem which can be easily solved by using Genetic Algorithm. It is a variation of constraint satisfaction problem. In which a set of variables is assigned in such a way that they avoid some constraints or follow some rules.

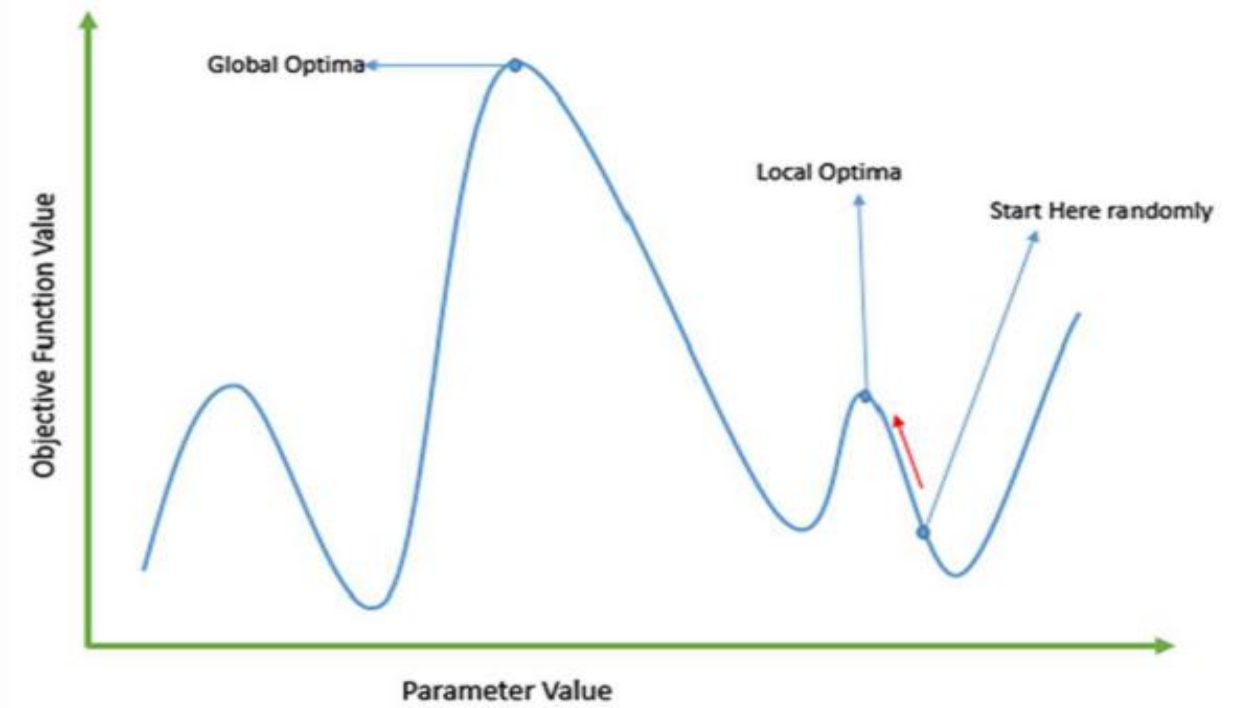
The constraint can be for following two types:

i) Hard Constraint- these are important requirements which need to be fulfilled always.

ii) Soft Constraint- these constraints are desirable but only after hard constraints are fulfilled.

Like, for class scheduling, hard constraint can be that at a given time a professor cannot be in different class rooms, class rooms should have enough space to host a given class etc.

Soft constraints can be preferred time slot for professors to take class or preferred class room of a professor.



2. Problem Statement

The class scheduling problem that we will be solving is equivalent to a college class scheduler which will make a schedule based on available information like professors available, rooms, time slots and student groups.

As this is a typical college class schedule, students can have free periods depending on the modules they have enrolled in.

In the schedule, each class will have a professor, a timeslot, a room and group of students enrolled in that class.

Total number of classes = (Sum of total number of student groups) * (Number of modules of each student group)

We will be using following hard constraint for developing our class scheduling application:

- Classes can only be scheduled in free classrooms
- A professor can only teach one class at any one time
- Classrooms must be big enough to accommodate the student group

Note: *To optimize my solution I have used **Parallel Processing** to calculate fitness functions and used **Hash Table** to store the values of fitness function to improve the running time of my application.*

3. Implementation Details

Pseudo Code for a Basic Genetic Algorithm

The pseudo code for a basic genetic algorithm is as follows:

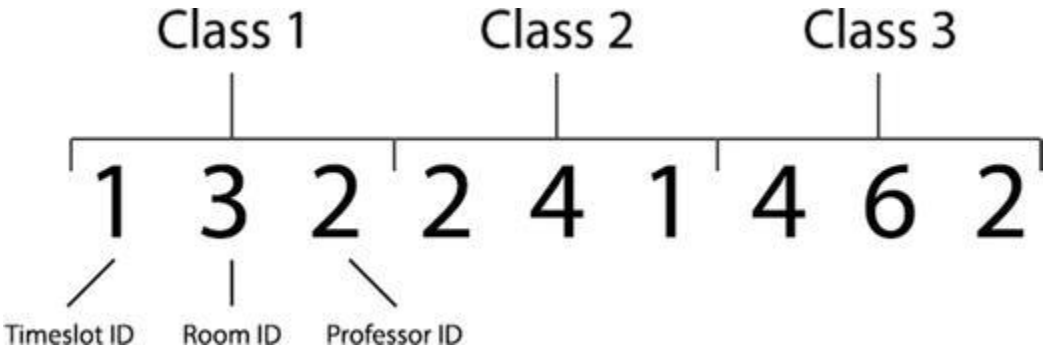
```
1: generation = 0;
2: population[generation] = initializePopulation(populationSize);
3: evaluatePopulation(population[generation]);
3: While isTerminationConditionMet() == false do
4:     parents = selectParents(population[generation]);
5:     population[generation+1] = crossover(parents);
6:     population[generation+1] = mutate(population[generation+1]);
7:     evaluatePopulation(population[generation]);
8:     generation++;
9: End loop;
```

Encoding The genetic code that we will use need to have all the information that is we need to know about a class like professor, class room and time slot.

I have given numeric Integer ids to all the three parameters.

Now we can design the chromosome based on this information containing an array of integers.

If we need any information regarding class then we just need to split the given integer into



Initialization

I made separate class for room, class, group, professor, module and timeslot. We can get data from large database but for now I have used dummy values for this.

A. Helper Classes for Class Schedule implementation:

Professor

`Professor` class has ID and name of professor. It also contains list of classes that professor teaches.

ClassRoom

`ClassRoom` class will have class room number and class room capacity. It also has class room id.

Course

`Course` is the subject that are being taught in the university. It has course id, course code and course number. It also has list of professors teaching that course.

StudentGroup

`StudentsGroup` class has ID and number of students (size of group). It also contains list of classes that group attends.

TimeSlot

`Timeslot` gives the available timeslots available for courses in the university. It has timeslot id and timeslot information.

UniversityClass

It holds the information of course that is being taught in the university. It will have professor id, course id, class room id of the room in which the class is being taught, student group attending the class and time slot information about the given courses.

B. Main Classes:

Schedule

After we are done creating the basic helper class for our scheduling application, we create a Schedule class. Its purpose is to encapsulate all these objects into a single schedule object. It is the main class with which our Genetic Algorithm will interact for scheduling function. It understands how the constraints work with respect to one another.

It is also used to create *Genotype* and *Phenotype* and create a candidate solution to evaluate and give fitness score.

This class contains method to add class rooms, timeslots, professors, courses and student groups to the schedule. So, the schedule class object knows all the information needed to schedule a class.

But, the most important methods present in this class are-

createClasses() method takes a chromosome taking the data of courses and group of students create many classes for those courses and student groups. The method then starts reading the chromosome and assigns the variable information (a timeslot, a room, and a professor) to each one of those classes.

Therefore, this method makes sure that every course and student group is accounted for and uses resultant genotype to try various combinations of class rooms, timeslots and professors.

It saves this information in cache for later use(this.univClasses).

calcClashes() method checks each class and then count the number of clashes. Any hard constraint violation is counted as a clash. This number of clash is then used by Genetic Algorithm's fitness function.

ScheduleGA

This is the class having public static void main() method.

It will initialize the population with following parameters

```
POPULATION SIZE = 100
MUTATION RATE = 0.01
CROSSOVER RATE = 0.9
ELITE INDIVIDUAL COUNT = 2
TOURNAMENT SIZE = 5
```

C. Fitness Evaluation:

Once our population has been initialized we need to evaluate the population and assign fitness function to each individual. Our goal is to create a schedule by breaking the hard constraints as less as possible i.e. having 0 number of clashes.

For this we use the createClass and calcClashes method of the schedule class and using these 2 methods we can easily write our fitness function as follows.

```
Schedule threadSchedule = new Schedule(schedule);
threadSchedule.createUnivClasses(individual);

// Calculate fitness
int clashes = threadSchedule.calcClashes();

double fitness = 1 / (double) (clashes + 1);

individual.setFitness(fitness);
```

The calcFitness method clones the Schedule object given to it, calls the createClasses method, and then calculates the number of clashes via the calcClashes method. The fitness is defined as the inverse as the number of clashes—0 clashes will result in a fitness of 1.

D. Terminating Condition:

For terminating our program, we use two terminating condition.

a. maximum number of generation

b. fitness of population.

The first isTerminationConditionMet call limits us to 1,000 generations, while the second checks to see if there are any individuals with a fitness of 1 in the population.

E. Parent Selection:

Fitness Proportionate Selection is one of the most popular ways of parent selection. In this every individual can become a parent with a probability which is proportional to its fitness. Therefore, fitter individuals have a higher chance of mating and propagating their features to the next generation.

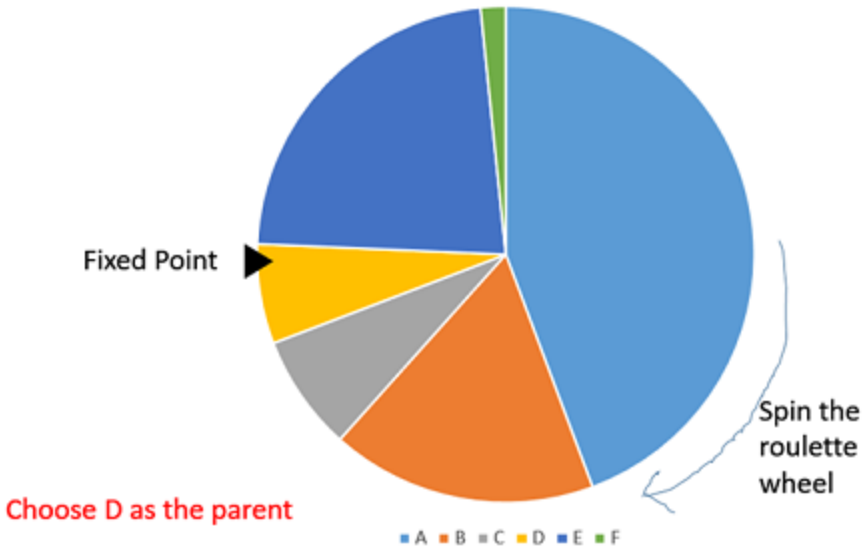
Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time.

Consider a circular wheel. The wheel is divided into n pies, where n is the number of individuals in the population. Each individual gets a portion of the circle which is proportional to its fitness value.

Two implementations of fitness proportionate selection are possible –

Roulette Wheel Selection

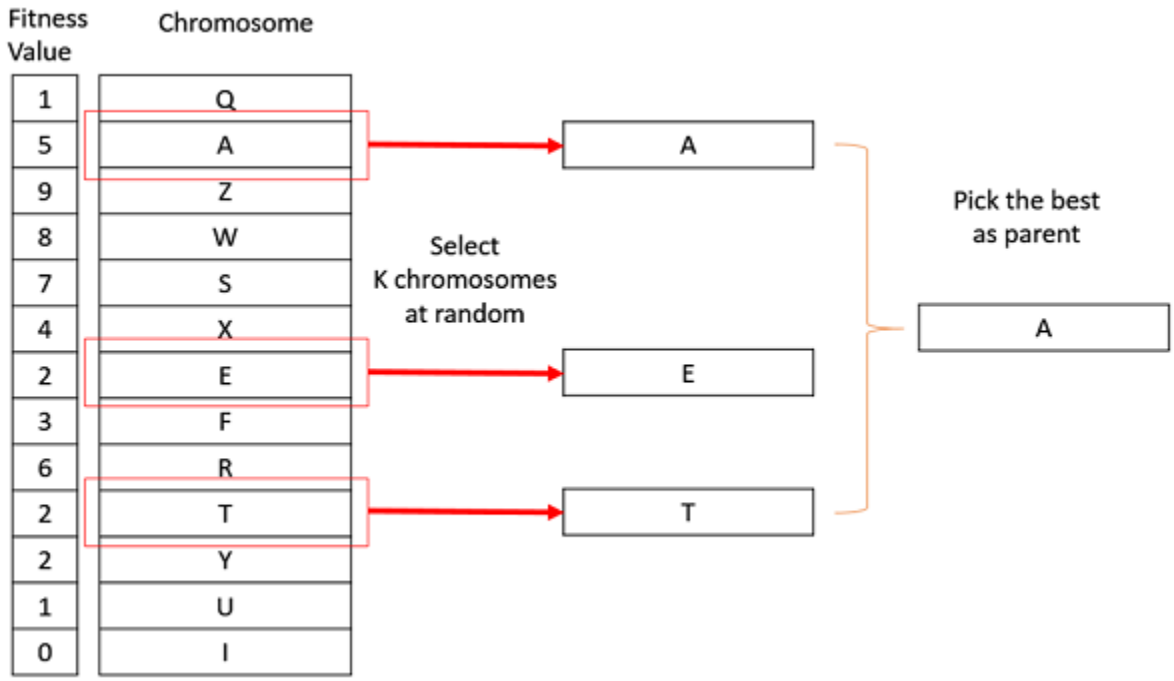
In a roulette wheel selection, the circular wheel is divided as described before. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

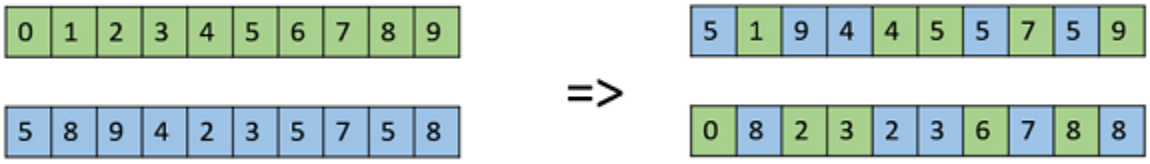
Tournament Selection

In K-Way tournament selection, we select K individuals from the population at random and select the best out of these to become a parent. The same process is repeated for selecting the next parent. Tournament Selection is also extremely popular in literature as it can even work with negative fitness values.



F. Crossover Function:

I have used tournament selection for parent selection and after that we comes to crossover part. Crossover can also be done by various methods but the method that I have used is Uniform Crossover. In uniform crossover each gene has 50% chance of either coming from 1st parent or second parent.



Crossover Pseudo Code

Now that we have a selection and a crossover method, let's look at some pseudo code which outlines the crossover process to be implemented.

```
1: For each individual in population:
2:     newPopulation = new array;
2:     If crossoverRate > random():
3:         secondParent = selectParent();
4:         offspring = crossover(individual, secondParent);
5:         newPopulation.push(offspring);
6:     Else:
7:         newPopulation.push(individual);
8:     End if
9: End loop;
```

G. Elitism:

A basic genetic algorithm will often lose the best individuals in a population between generations because of the crossover and mutation operators. However, we need these operators to find better solutions.

One simple optimization technique used to tackle this problem is to always allow the fittest individual, or individuals, to be added unaltered to the next generation's population. This way the best individuals are no longer lost from generation to generation.

This process of retaining the best for the next generation is called elitism.

Typically, the optimal number of 'elite' individuals in a population will be a very small proportion of the total population size. This is because if the value is too high, it will slow down the genetic algorithm's search process due to a lack of genetic diversity caused by preserving too many individuals.

Implementing elitism is simple in both crossover and mutation contexts.

```
Public Population crossoverPopulation( Population population){
// some code
// Apply crossover to this individual?
if(this.crossoverRate> Math.random() && populationIndex >= this.elitismCount) {
    // more codes to follow
                }
}
```

Crossover is only applied if *both* the crossover conditional is met *and* the individual is not considered elite.

At this point, the individuals in the population have already been sorted by their fitness, so the strongest individuals have the lowest indices. Therefore, if we want three elite individuals, we should skip indices 0-2 from consideration.

H. Mutation Function:

As our genotype is built of specific class room, professor id and timeslots; we cannot choose a random number for mutation function. Also, we cannot choose random number from 0 to X for that case.

For mutation we need to take the same approach that we took during crossover that is uniform crossover.

Mutation can be implemented in a similar manner.

Instead of choosing a random number for a random gene in the chromosome, we can create a new random but valid individual and essentially run uniform crossover to achieve mutation!

That is, we can use our Individual(Schedule) constructor to create a brand new random Individual, and then select genes from the random Individual to copy into the Individual to be mutated.

This technique is called uniform mutation, and makes sure that all our mutated individuals are fully valid, never selecting a gene that doesn't make sense.

4. Optimization:

To make our genetic algorithm fast in converging to optimum solution we can use several optimization techniques. These techniques help us in getting to result faster and can also prevent our program to stop at local minima.

In my program I have used following optimizations:

A- Parallel Processing

Using the Java 8 functionality we can achieve the parallel processing. I have used IntStrem function to achieve parallel processing on my fitness function.

```
 public void evalPopulation(Population population, Schedule schedule){

IntStream.range(0, population.size()).parallel()
         .forEach(i -> this.calcFitness(population.getIndividual(i),
schedule));
```

B- Using Hash Table for storing Fitness Values

As fitness is most important and consuming function in a genetic algorithm we can use fitness value hashing by storing the previously computed fitness values in a hash table.

So, each time fitness value need not to be calculate while our algorithm is running and same can be taken from the hash table.

```
// Create fitness hashtable
private Map<Individual, Double> fitnessHash
Collections.synchronizedMap( new LinkedHashMap<Individual, Double>() {
        @Override
        protected boolean removeEldestEntry(Entry<Individual, Double>
eldest) {
            // Store a maximum of 1000 fitness values
            return this.size() > 1000;
        }
    });
```

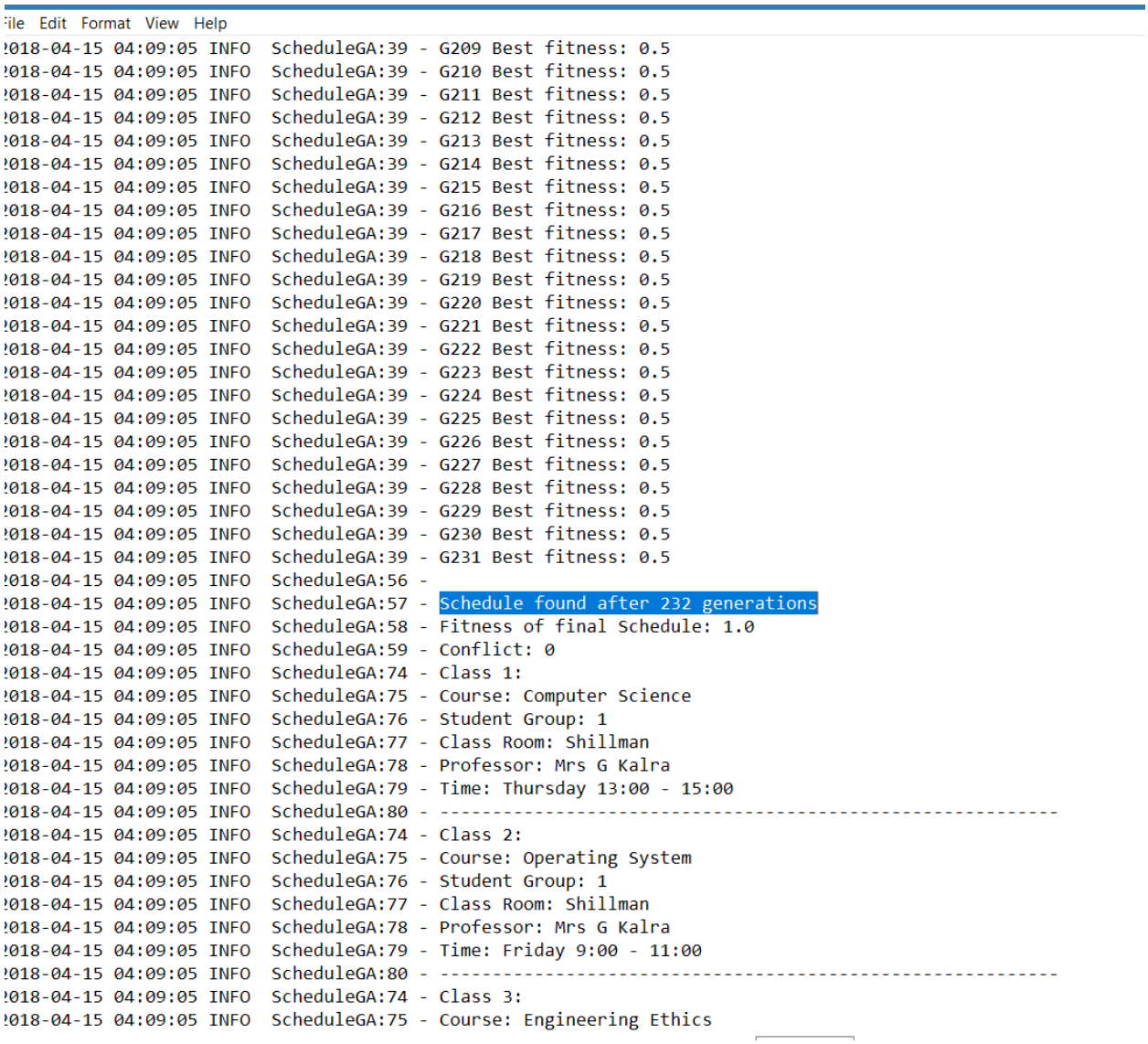
As we are using Individual class object for hash function I have overridden the equals and hashcode method of Individual class.

```
/**
     * Generates hash code based on individual's
     * chromosome
     *
     * @return Hash value
     */
    @Override
    public int hashCode() {
        int hash = Arrays.hashCode(this.chromosome);
        return hash;
    }

    /**
     * Equates based on individual's chromosome
     *
     * @return Equality boolean
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }

        if (getClass() != obj.getClass()) {
            return false;
        }

        Individual individual = (Individual) obj;
        return Arrays.equals(this.chromosome,
individual.chromosome);
    }
```

5. Program Output:

The output has been saved in log file.

The screenshots over various runs are as follows.



File Edit Format View Help

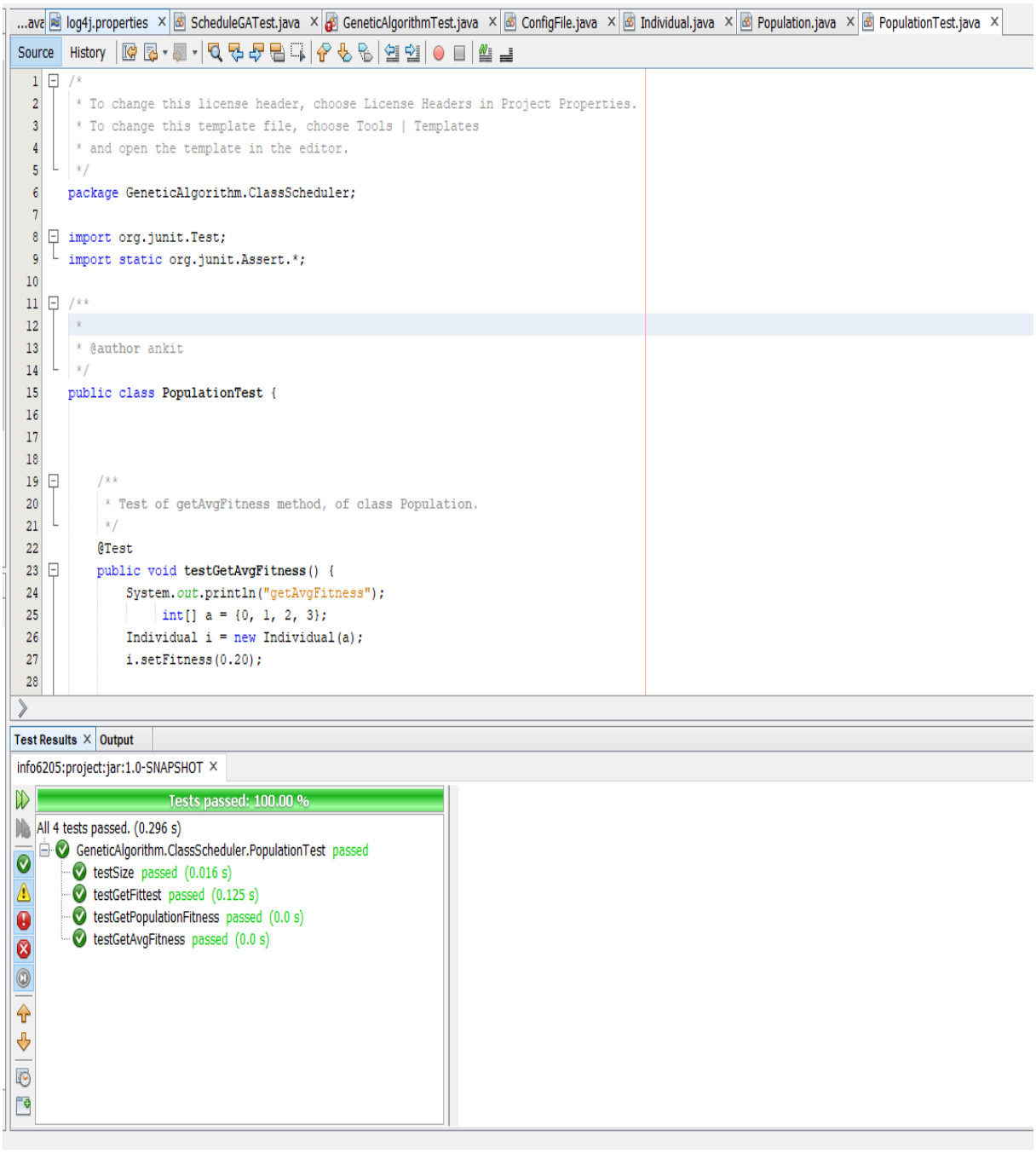
```
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G209 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G210 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G211 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G212 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G213 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G214 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G215 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G216 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G217 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G218 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G219 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G220 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G221 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G222 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G223 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G224 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G225 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G226 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G227 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G228 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G229 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G230 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:39 - G231 Best fitness: 0.5
2018-04-15 04:09:05 INFO  ScheduleGA:56 -
2018-04-15 04:09:05 INFO  ScheduleGA:57 - Schedule found after 232 generations
2018-04-15 04:09:05 INFO  ScheduleGA:58 - Fitness of final Schedule: 1.0
2018-04-15 04:09:05 INFO  ScheduleGA:59 - Conflict: 0
2018-04-15 04:09:05 INFO  ScheduleGA:74 - Class 1:
2018-04-15 04:09:05 INFO  ScheduleGA:75 - Course: Computer Science
2018-04-15 04:09:05 INFO  ScheduleGA:76 - Student Group: 1
2018-04-15 04:09:05 INFO  ScheduleGA:77 - Class Room: Shillman
2018-04-15 04:09:05 INFO  ScheduleGA:78 - Professor: Mrs G Kalra
2018-04-15 04:09:05 INFO  ScheduleGA:79 - Time: Thursday 13:00 - 15:00
2018-04-15 04:09:05 INFO  ScheduleGA:80 - ------------------------------------------------------------
2018-04-15 04:09:05 INFO  ScheduleGA:74 - Class 2:
2018-04-15 04:09:05 INFO  ScheduleGA:75 - Course: Operating System
2018-04-15 04:09:05 INFO  ScheduleGA:76 - Student Group: 1
2018-04-15 04:09:05 INFO  ScheduleGA:77 - Class Room: Shillman
2018-04-15 04:09:05 INFO  ScheduleGA:78 - Professor: Mrs G Kalra
2018-04-15 04:09:05 INFO  ScheduleGA:79 - Time: Friday 9:00 - 11:00
2018-04-15 04:09:05 INFO  ScheduleGA:80 - ------------------------------------------------------------
2018-04-15 04:09:05 INFO  ScheduleGA:74 - Class 3:
2018-04-15 04:09:05 INFO  ScheduleGA:75 - Course: Engineering Ethics
```

6. Test Case Passed:

There are 2 test files in my project

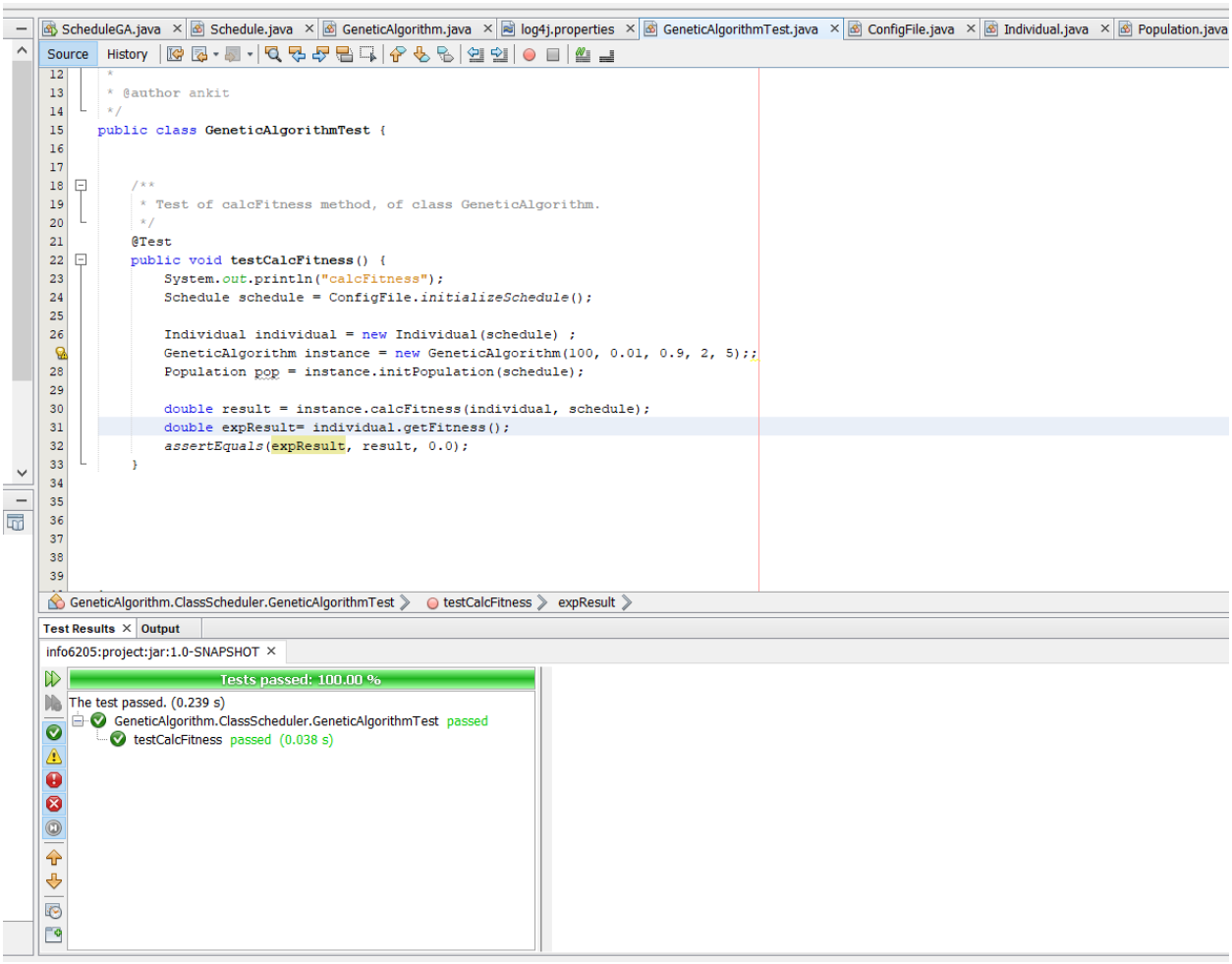
i) PopulationTest

This class is used to initialize population, calculate fitness of population and sort the population with respect to fitness in descending order.



ii) GeneticAlgorithmTest

In this class the fitness of an individual is calculated with respect to schedule class.



7. Conclusion:

One of the major different that Scheduling problem have over other NP complete problem like travelling salesman is that there can be many cases in which the algorithm did not give any solution.

For example, in travelling salesman each candidate solution is a solution even if not optimum one. But, in scheduling problem if the timeslot clashes then it is not an acceptable solution.

So, we must take extra care while making our program to consider all the constraints that have been given to it.

After experimenting with crossover rate and mutation rate our system was giving correct solution in about 25 to 35 generations.

We can add the complexity by adding more classes, or by adding some more soft constraints.

Scheduling implementation using Algorithm have huge application in various fields like operation research and production planning.

8. References

1. https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications

2. https://en.wikipedia.org/wiki/Genetic_algorithm_scheduling

3. Daniel Schiffman's excellent book *The Nature of Code*- http://natureofcode.com

4. An Introduction to Genetic Algorithms by Melanie Mitchell

5. Ahuja, Rabindra, and Orlin, James (2002). Very Large-Scale Neighborhood Search in Airline Fleet Scheduling. SIAM News, November 2002.

6. Goldberg, David (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley Longman, Inc.

7. Arjan Tijms, drs. And Johan van der Meulen (2002). Genetic Algorithm, University Leiden.

8. Levi, Delon (2000). HereBoy: A Fast-Evolutionary Algorithm. The second NASA/DoD Workshop on Evolvable Hardware (EH'00). July 13-15, 2000. Palo Alto California.