TIMETABLE GENERATION USING GENETIC ALGORITHM

Ishita B. Elton B. Ankur B.

Index

[1. TERMS 3](#_Toc500626760)

[2. PARAMETERS 4](#_Toc500626761)

[3. PSEUDO CODE FOR A BASIC GENETIC ALGORITHM 5](#_Toc500626762)

[4. THE PROCESS 6](#_Toc500626763)

[5. THE PROBLEM : TIMETABLE GENERATION 7](#_Toc500626764)

[6. REST API - SWAGGER OUTPUT 9](#_Toc500626765)

[7. JUNIT TEST CASES 14](#_Toc500626766)

[8. OBSERVATIONS & FINDINGs 19](#_Toc500626767)

[LITERATURE CITED 25](#_Toc500626768)

**ABSTRACT**

Genetic algorithms were built on the concepts of biological evolution, so if you’re familiar with the terminology found in evolution, you’ll likely notice overlap in the terminology found when working with genetic algorithms. The similarities between the fields are of course due to evolutionary algorithms, and more specifically, genetic algorithms being analogous to processes found in nature.

In this project, we will create a genetic algorithm that schedules classes for a college timetable. We will examine a couple of different scenarios in which a class scheduling algorithm may be used, and the constraints that are usually implemented when designing a timetable. Finally, we will build a simple class scheduler, which can be expanded to support more complex implementations. In artificial intelligence, class scheduling is a variation of the constraint satisfaction problem. This category of problem relates to problems, which have a set of variables that need to be assigned in such a way that they avoid violating a defined set of constraints.

Constraints fall into two categories: *hard constraints—constraints* which need to be satisfied to produce a functional solution, and *soft constraints—constraints* which are preferred but not at the expense of the hard constraints. For instance, when manufacturing a new product, the product’s functional requirements are hard constraints and specify the important performance requirements. Without these constraints, you have no product. A phone that can’t make calls is hardly a phone! However, you may also have soft constraints, which, although unrequired, are still important to consider, such as the cost, weight, or aesthetics of the product. When creating a class-scheduling algorithm there will typically be many hard and soft constraints that need to be considered. Some typical hard constraints for the class-scheduling problem are: • Professors can only be in one class at any given time • Classrooms need to be big enough to host the class • Classrooms can only host one class at any given time • Classrooms must contain any required equipment

## 

## 

## 1. TERMS

Population - This is simply just a collection of candidate solutions which can have genetic operators such as mutation and crossover applied to them.

Candidate Solution – A possible solution to a given problem.

Gene – The indivisible building blocks making up the chromosome. Classically a gene consists of 0 or a 1.

Chromosome – A chromosome is a string of genes. A chromosome defines a specific candidate solution. A typical chromosome with a binary encoding might contain something like, “01101011”.

Mutation – The process in which genes in a candidate solution are randomly altered to create new traits.

Crossover – The process in which chromosomes are combined to create a new candidate solution. This is sometimes referred to as recombination.

Selection – This is the technique of picking candidate solutions to breed the next generation of solutions.

Fitness – A score which measures the extent to which a candidate solution is adapted to suit a given problem.

## 

## 

## 2. PARAMETERS

Although all genetic algorithms are based on the same concepts, their specific implementations can vary quite a bit. One of the ways specific implementations can vary is by their parameters. A basic genetic algorithm will have at least a few parameters that need to be considered during the implementation. The main three are the rate of mutation, the population size and the third is the crossover rate.

Mutation Rate

The mutation rate is the probability in which a specific gene in a solution’s chromosome will be mutated.

If the mutation rate is too low, the algorithm can take an unreasonably long time to move along the search space hindering its ability to find a satisfactory solution

Population Size

The population size is simply the number of individuals in the genetic algorithm’s population in any one generation.

Crossover Rate

The frequency in which crossover is applied also has an effect on the overall performance of the genetic algorithm.

A high rate allows for many new, potentially superior, solutions to be found during the crossover phase. A lower rate will help keep the genetic information from fitter individuals intact for the next generation.

## 

## 

## 3. 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

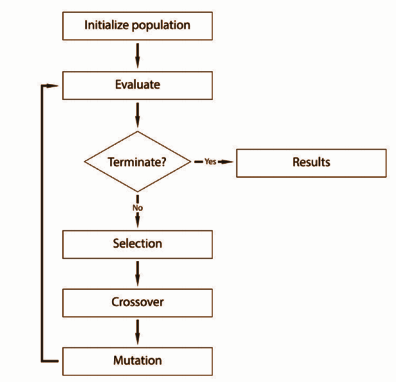
4:parents = selectParents(population[generation]); 5:population[generation+1] = crossover(parents);

6:population[generation+1]=mutate(population[generation+1]);

7: evaluatePopulation(population[generation]); 8: generation++;

9: End loop;

## 4. THE PROCESS



1. Genetic algorithms begin by initializing a population of candidate solutions. This is typically done randomly to provide an even coverage of the entire search space.

2. Next, the population is evaluated by assigning a fitness value to each individual in the population. In this stage we would often want to take note of the current fittest solution, and the average fitness of the population.

3. After evaluation, the algorithm decides whether it should terminate the search depending on the termination conditions set. Usually this will be because the algorithm has reached a fixed number of generations or an adequate solution has been found.

4. If the termination condition is not met, the population goes through a selection stage in which individuals from the population are selected based on their fitness score – the higher the fitness, the better chance an individual has of being selected. Figure.

5. The next stage is to apply crossover and mutation to the selected individuals. This stage is where new individuals are created for the next generation.

6. At this point the new population goes back to the evaluation step and the process starts again. We call each cycle of this loop a generation.

7. When the termination condition is finally met, the algorithm will break out of the loop and typically return its finial search results back to the user.

## 5. THE PROBLEM : TIMETABLE GENERATION

We will create a genetic algorithm that schedules classes for a college timetable. We will examine a couple of different scenarios in which a class scheduling algorithm may be used, and the constraints that are usually implemented when designing a timetable. Finally, we will build a simple class scheduler, which can be expanded to support more complex implementations. In artificial intelligence, class scheduling is a variation of the constraint satisfaction problem. This category of problem relates to problems, which have a set of variables that need to be assigned in such a way that they avoid violating a defined set of constraints.

Constraints fall into two categories: hard constraints—constraints which need to be satisfied to produce a functional solution, and soft constraints—constraints which are preferred but not at the expense of the hard constraints. For instance, when manufacturing a new product, the product’s functional requirements are hard constraints and specify the important performance requirements. Without these constraints, you have no product. A phone that can’t make calls is hardly a phone! However, you may also have soft constraints, which, although unrequired, are still important to consider, such as the cost, weight, or aesthetics of the product. When creating a class-scheduling algorithm there will typically be many hard and soft constraints that need to be considered.

Some typical hard constraints for the class-scheduling problem are:

• Professors can only be in one class at any given time

• Classrooms need to be big enough to host the class

• Classrooms can only host one class at any given time

• Classrooms must contain any required equipment

To keep things simple in this implementation we will consider only hard constraints for now; however there would typically be many more hard constraints depending on the timetable specifications. There would also likely be a number of soft constraints included in the specification, which for now we will ignore. Although not necessary, considering soft constraints can often make a big difference in the quality of the timetables produced by the genetic algorithm

IMPLEMENTATION

Encoding of a Gene

Class :

Time\_Slot\_Id Room\_Id Prof\_Id

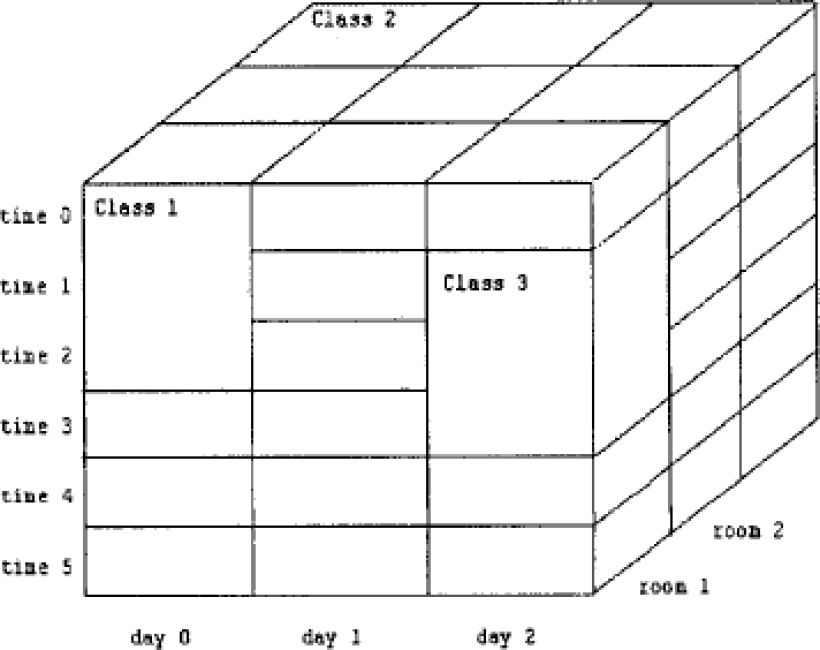
Population:

1 3 2

2 5 2

6 5 4

Calculation of Fitness based on Clashes



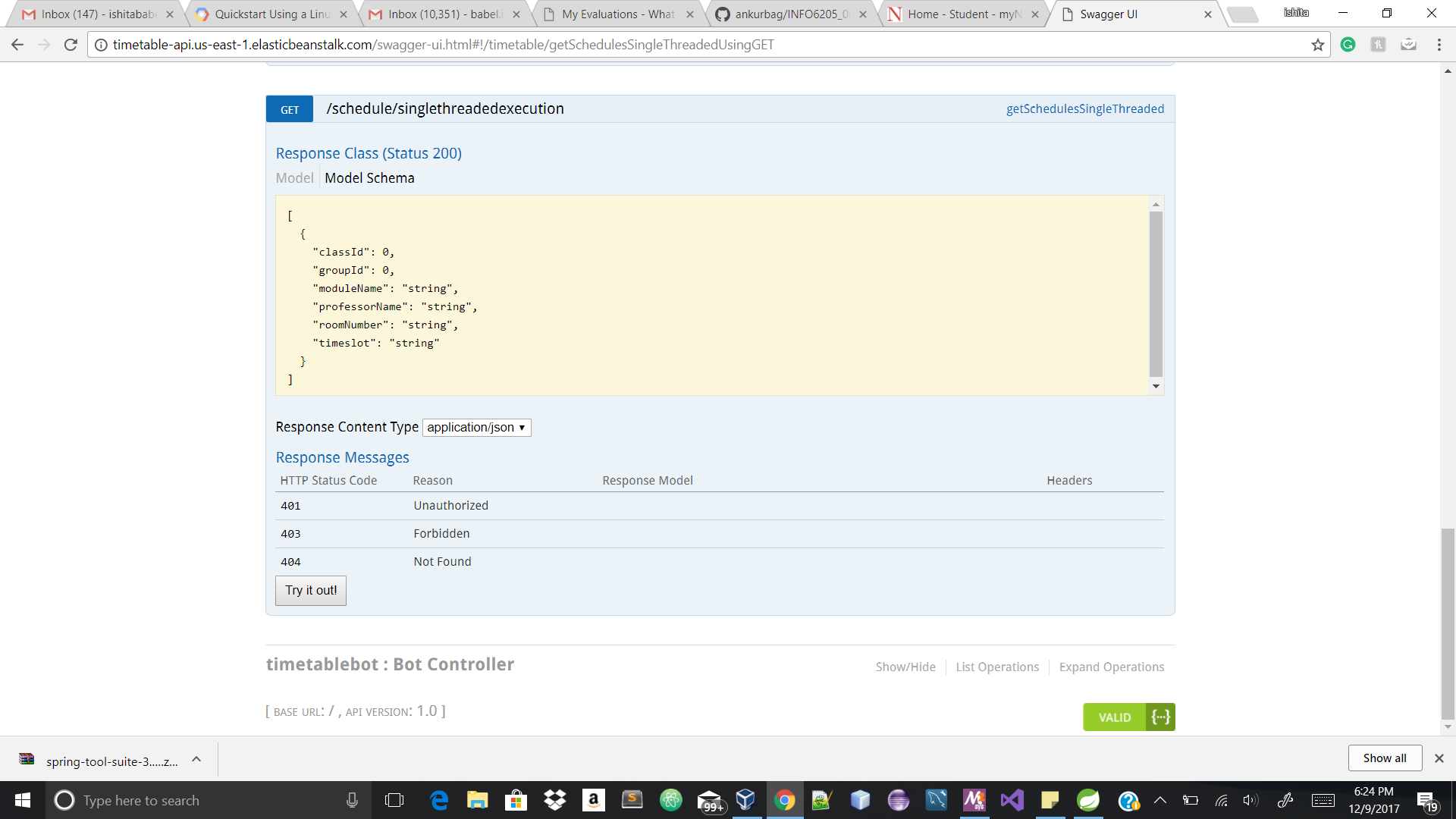
The fitness value of an individual is calculated as:

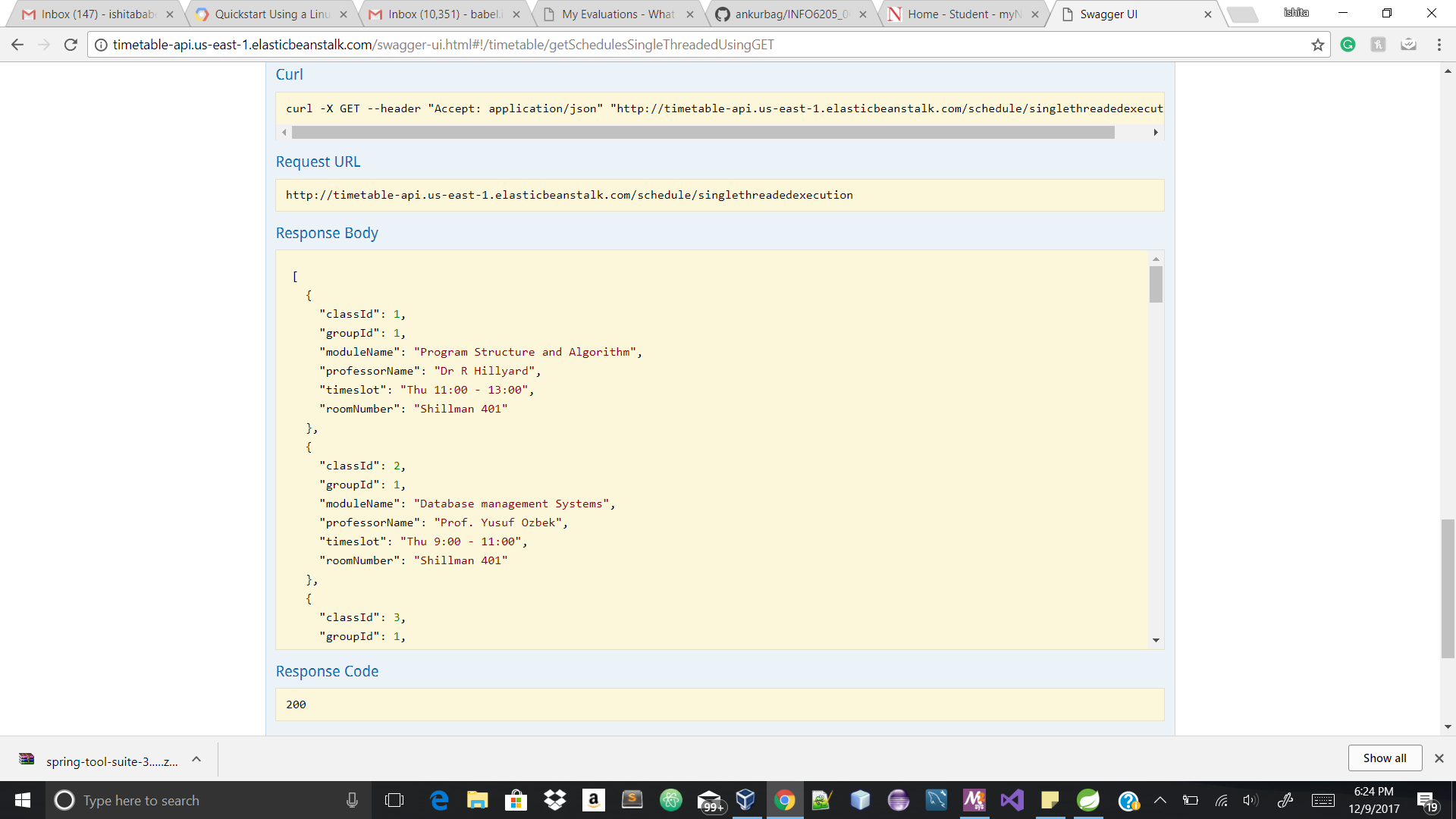
fitness(individual) :1/(Number of clashes)

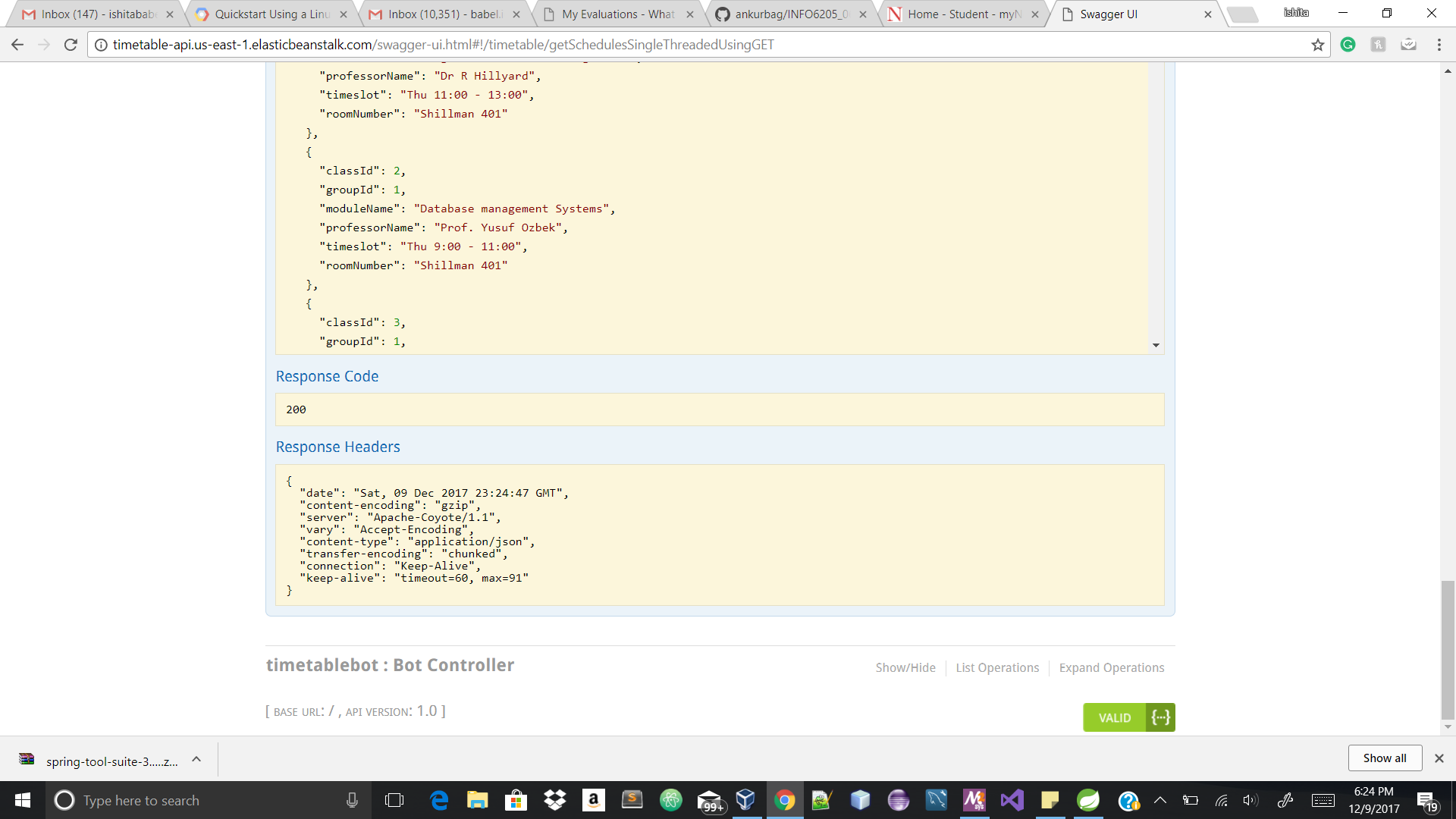
## 6. REST API - SWAGGER OUTPUT

Swagger is the world’s largest framework of API developer tools for the OpenAPI Specification(OAS), enabling development across the entire API lifecycle, from design and documentation, to test and deployment.

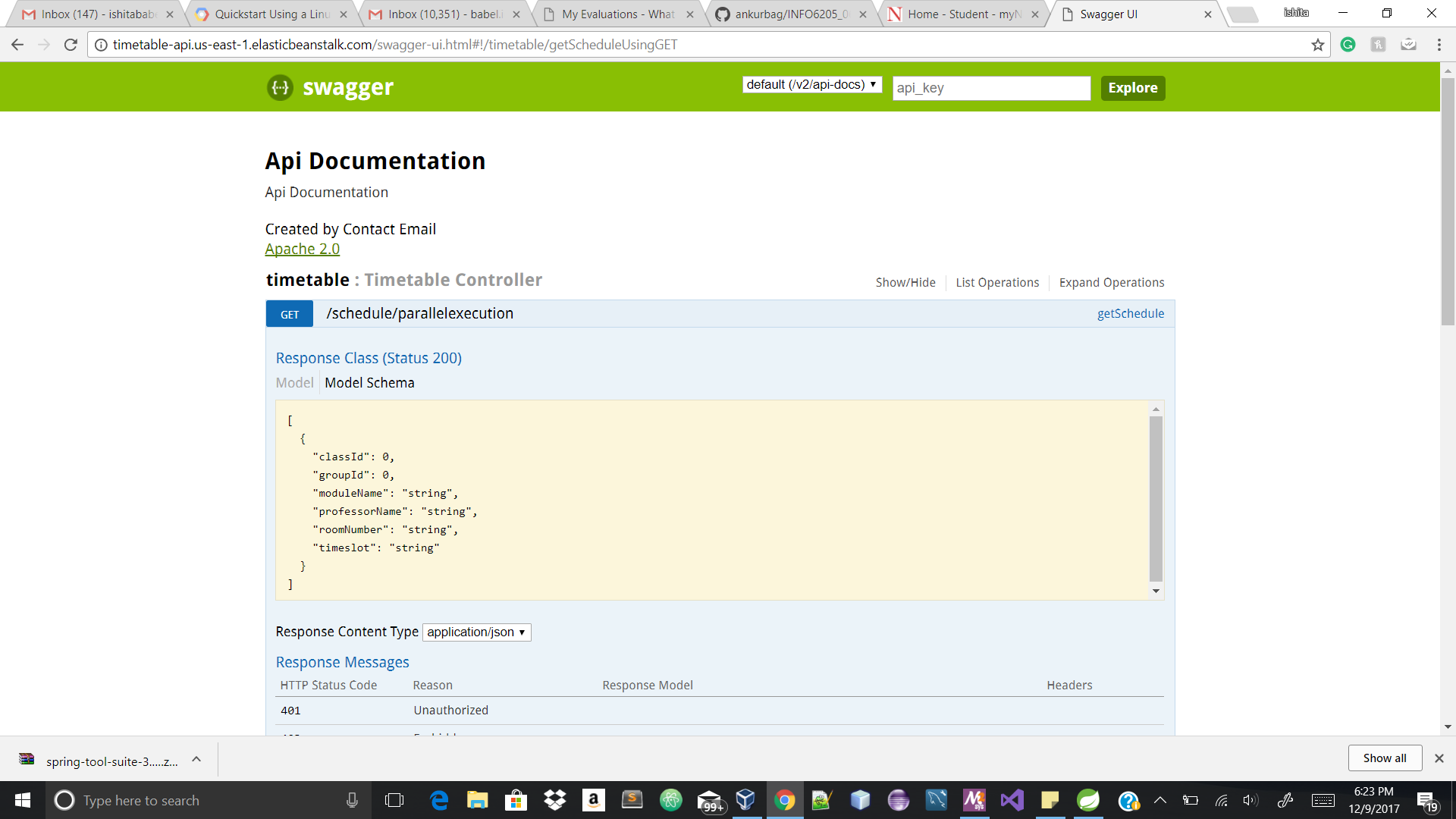
# SINGLE THREADED EXECUTION

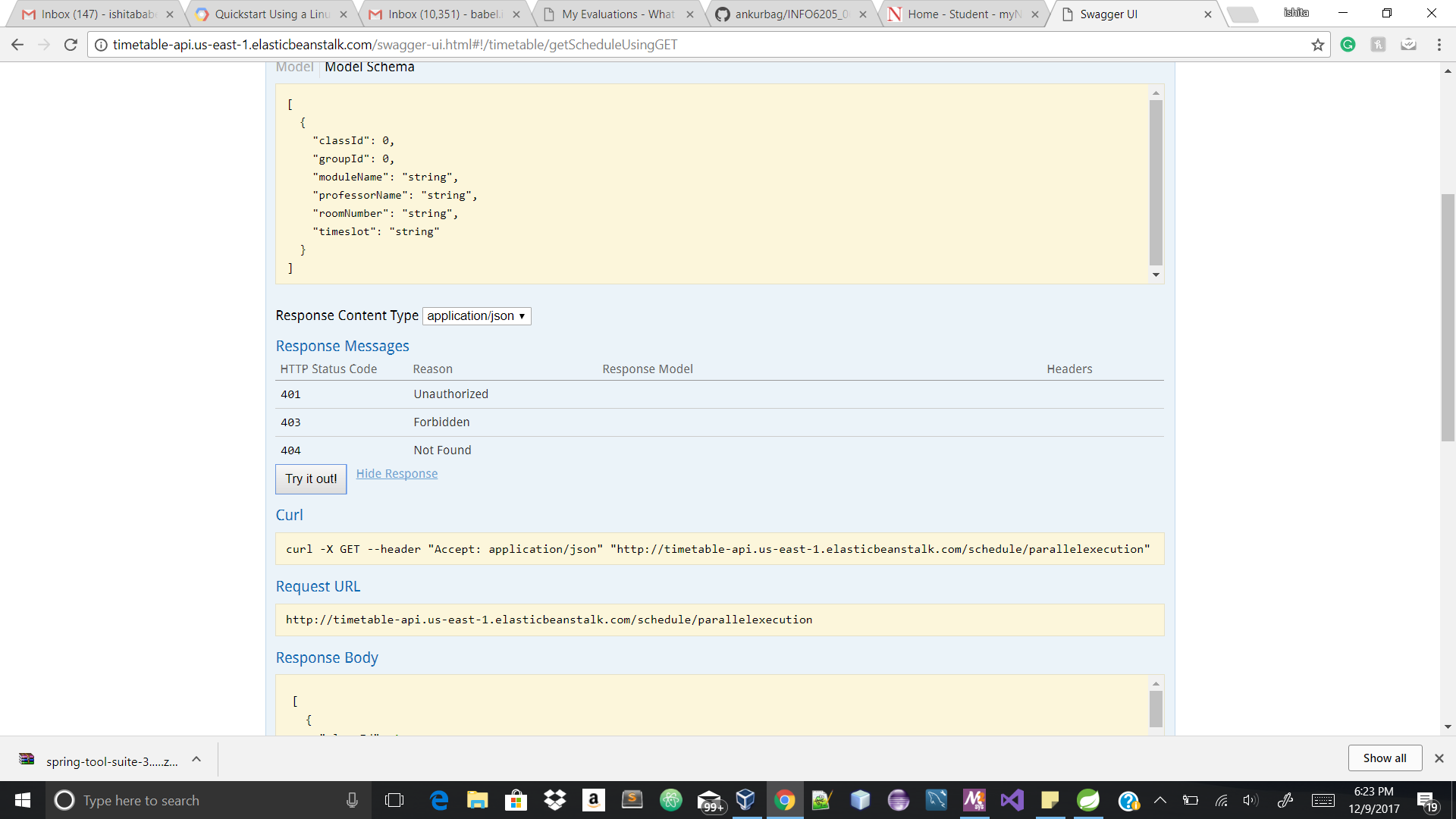


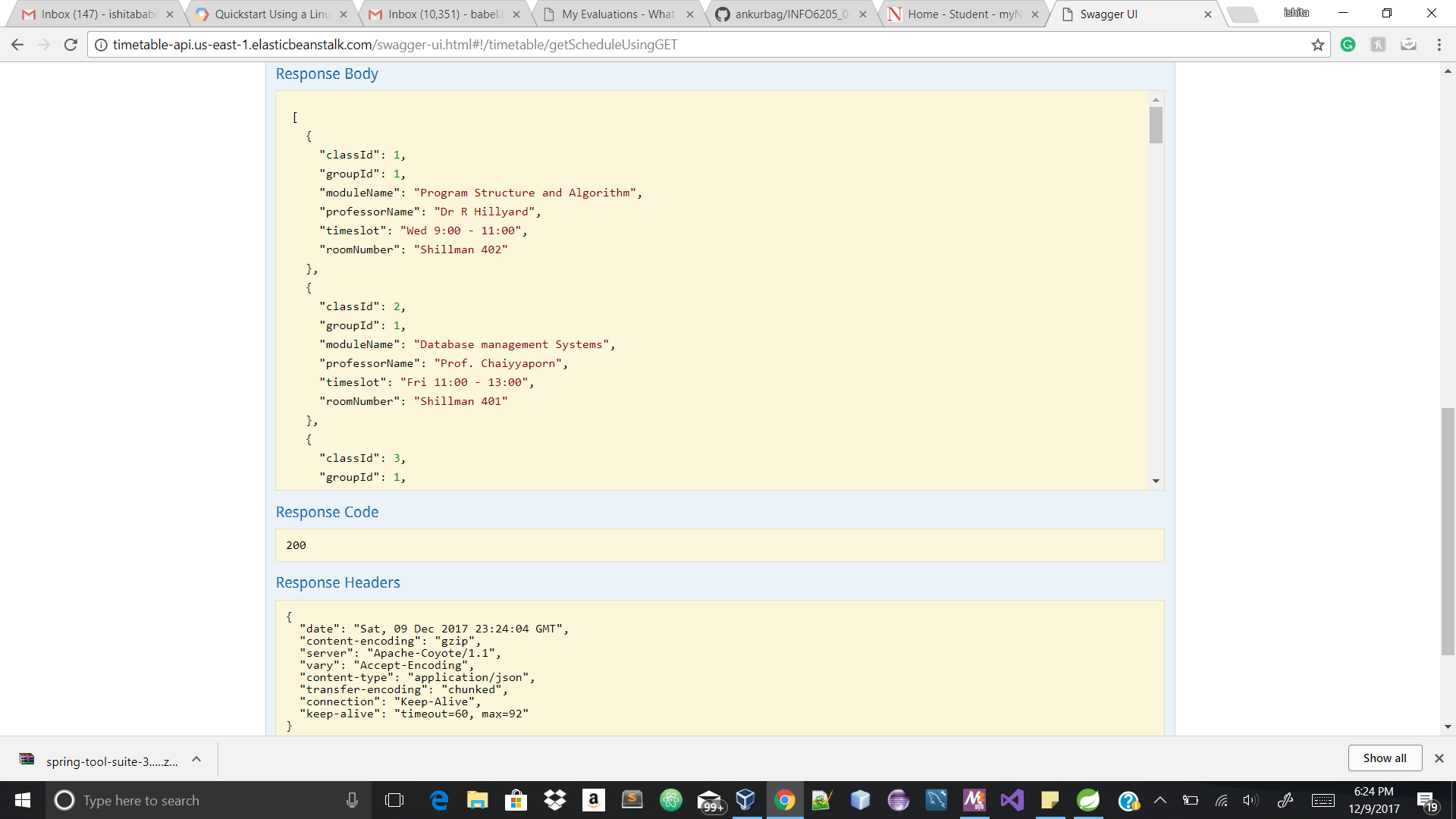




# PARALLEL EXECUTION







About the output – We are generating JSON data for Schedule

[

{

"classId": 0,

"groupId": 0,

"moduleName": "string",

"professorName": "string",

"roomNumber": "string",

"timeslot": "string"

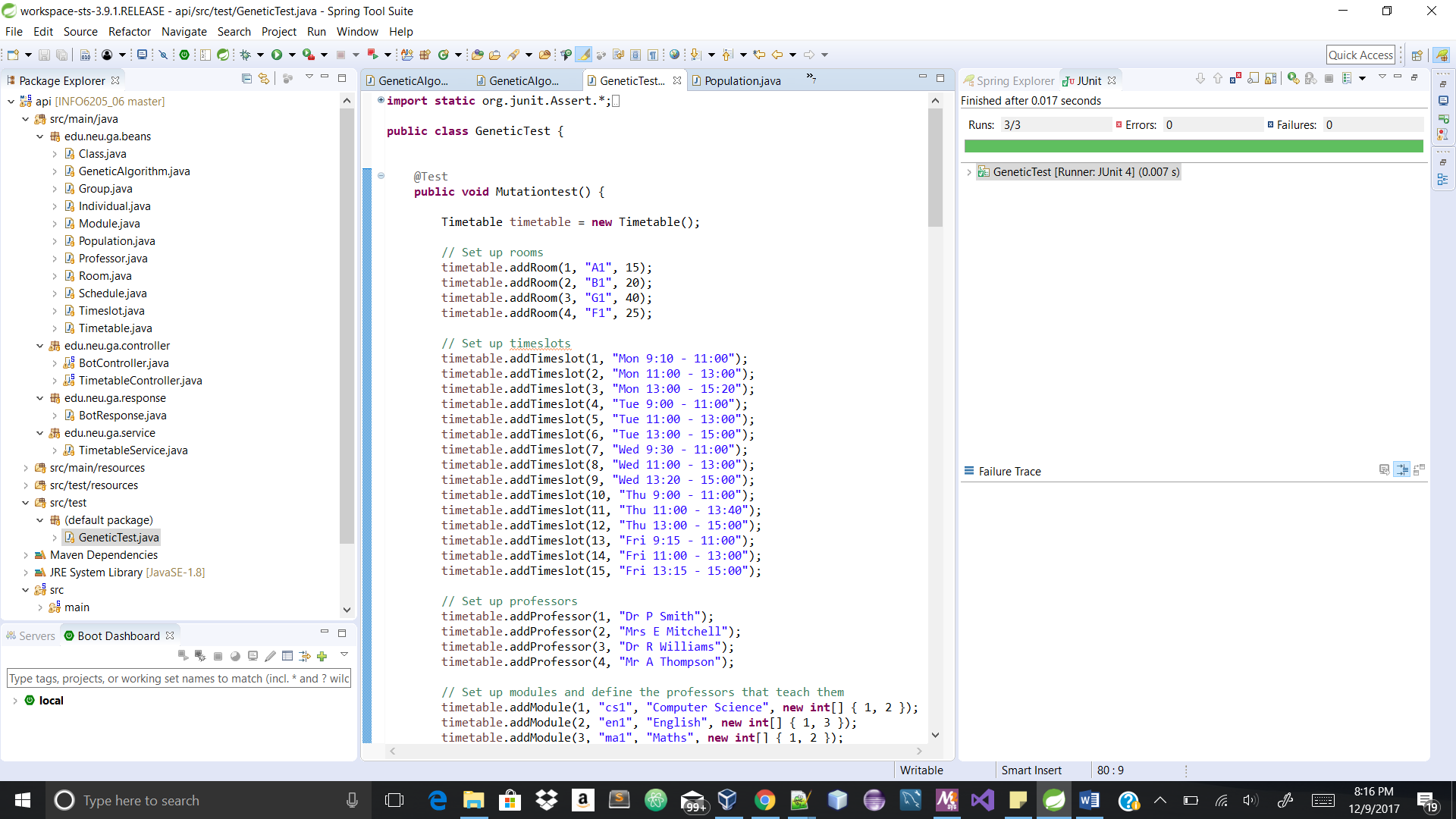
}

]

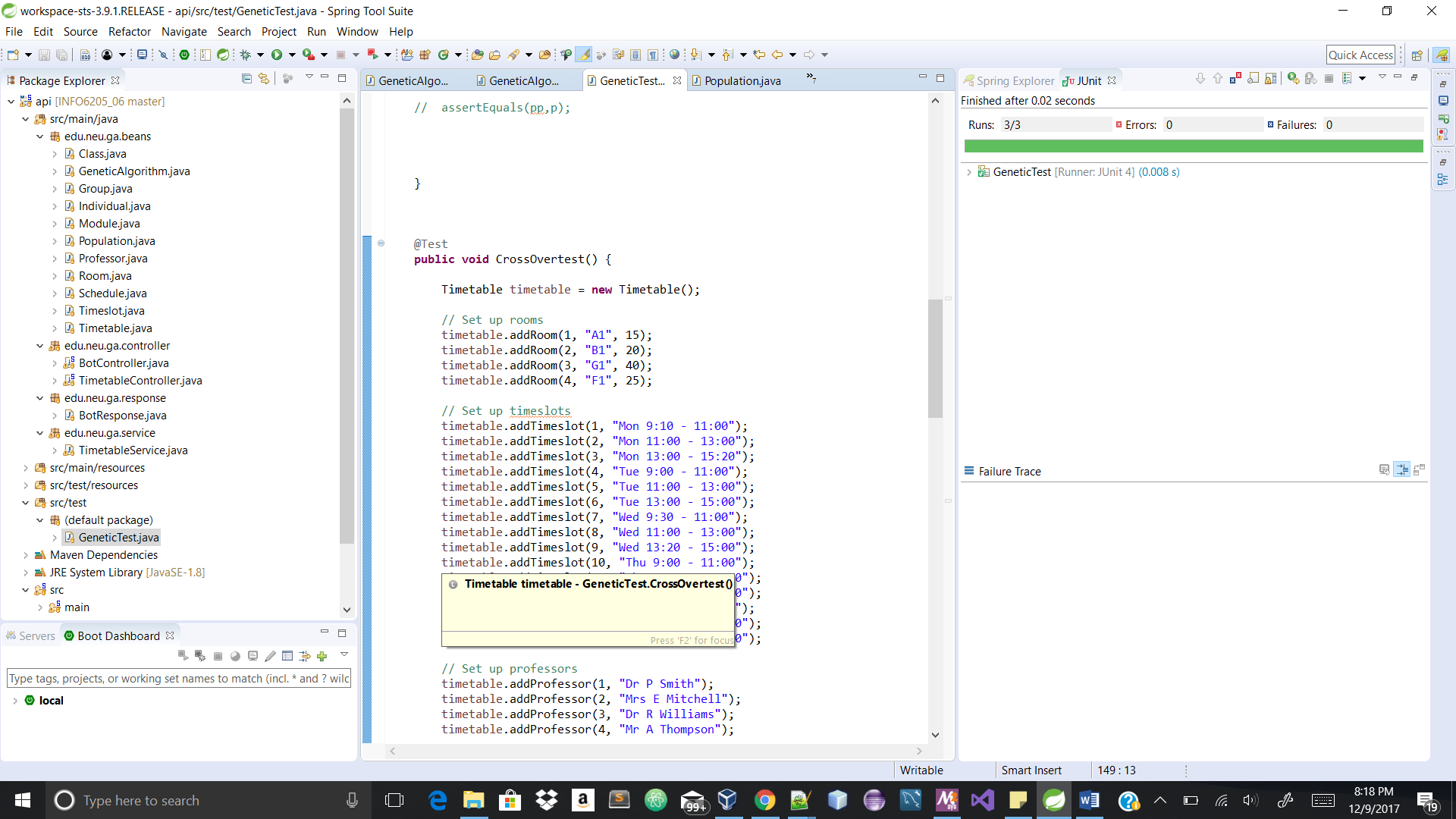
## 7. JUNIT TEST CASES

1)To check population before and after mutation

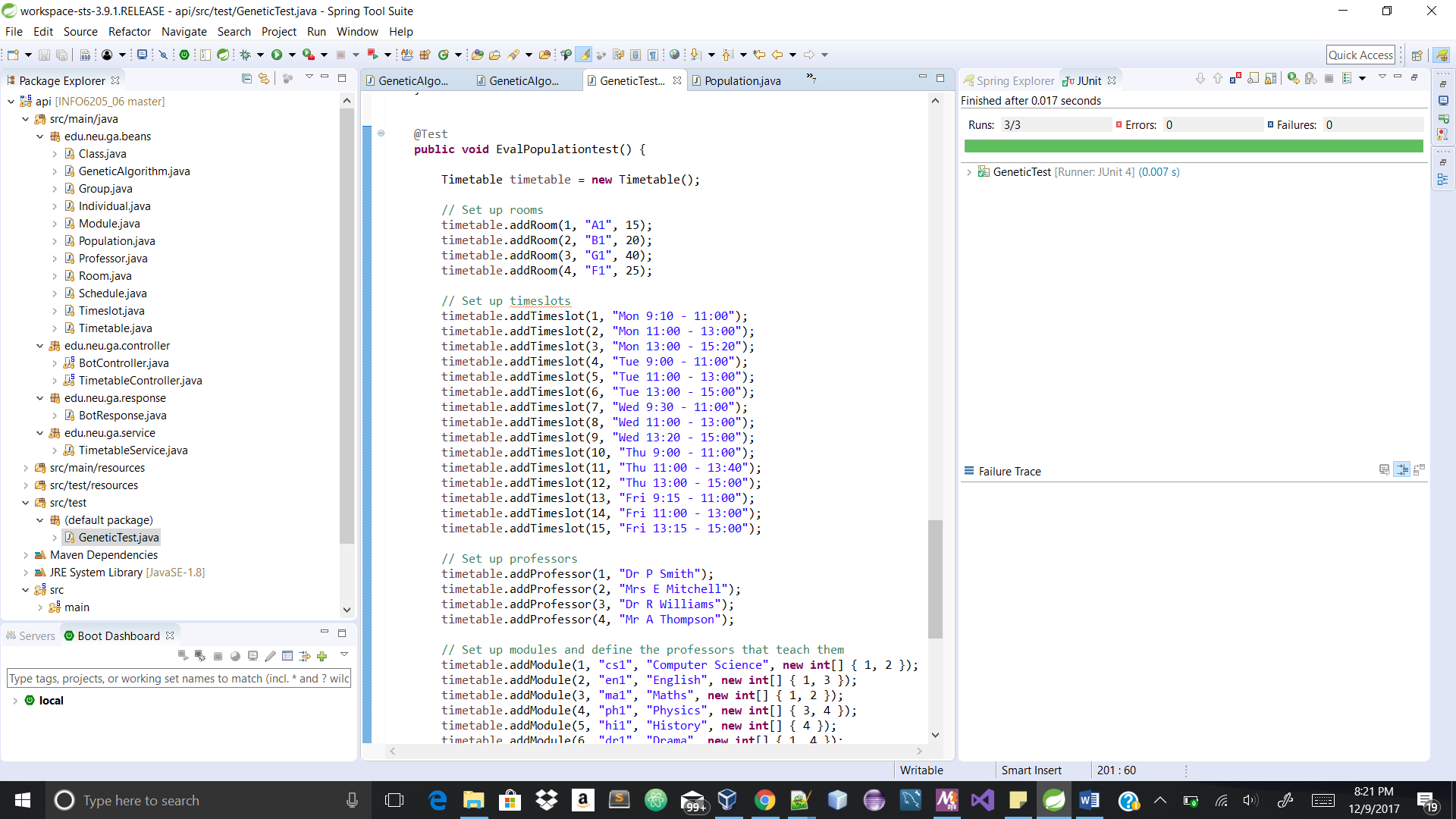
OUTPUT-



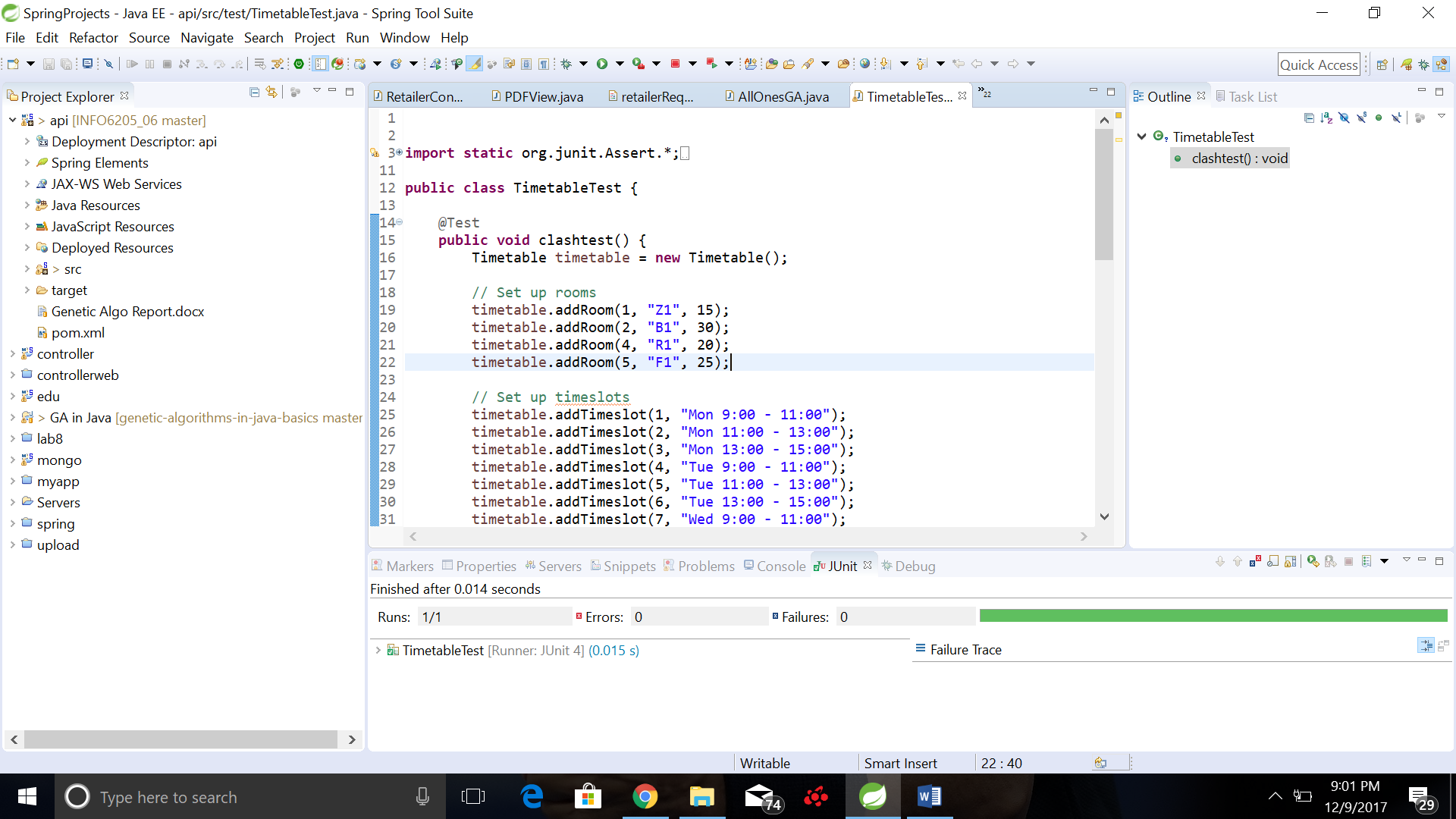
2) To check the population before and after CrossOver



3) To check whether the population fitness is null or not.

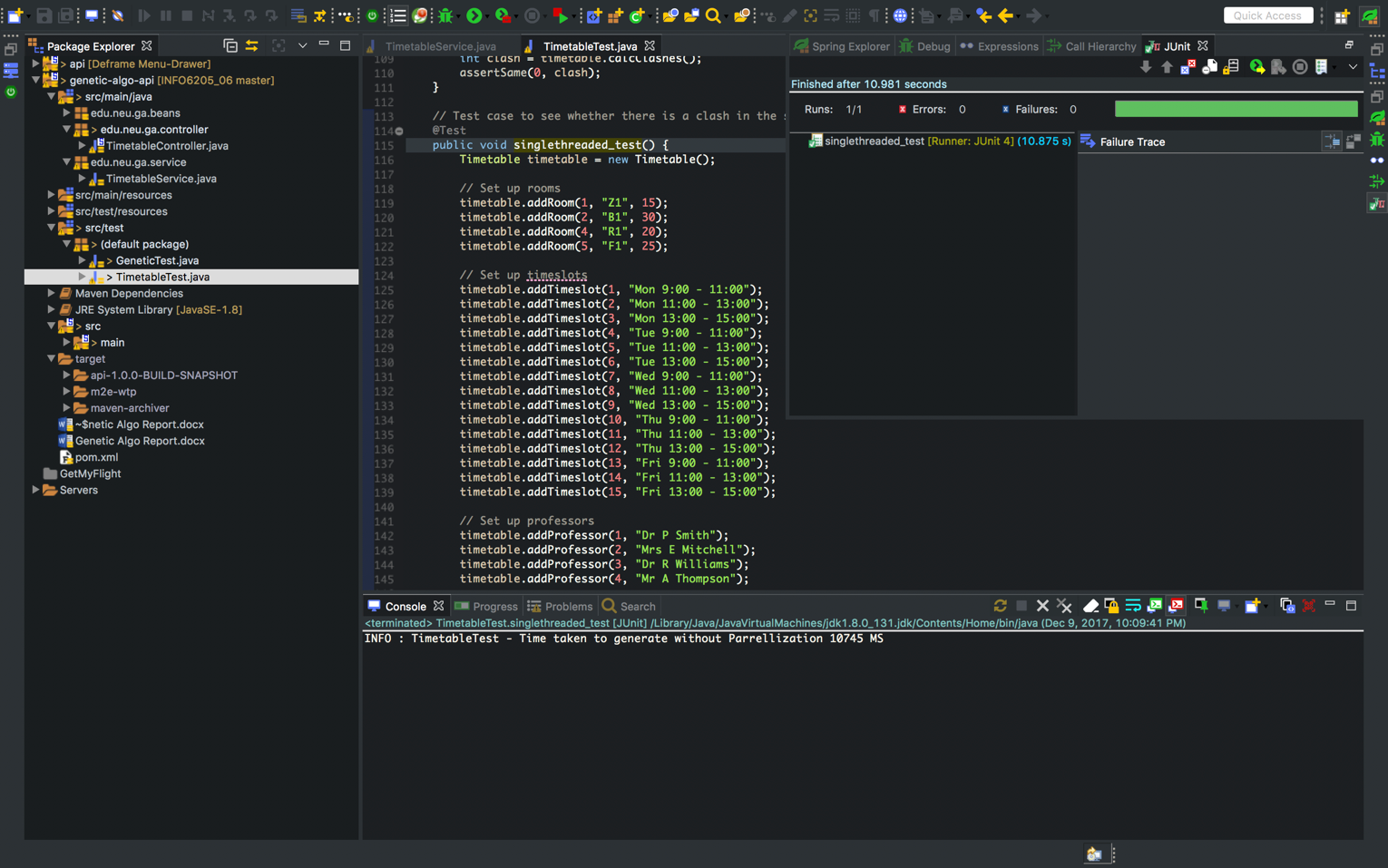


4) To check if number of clashes is getting calculated

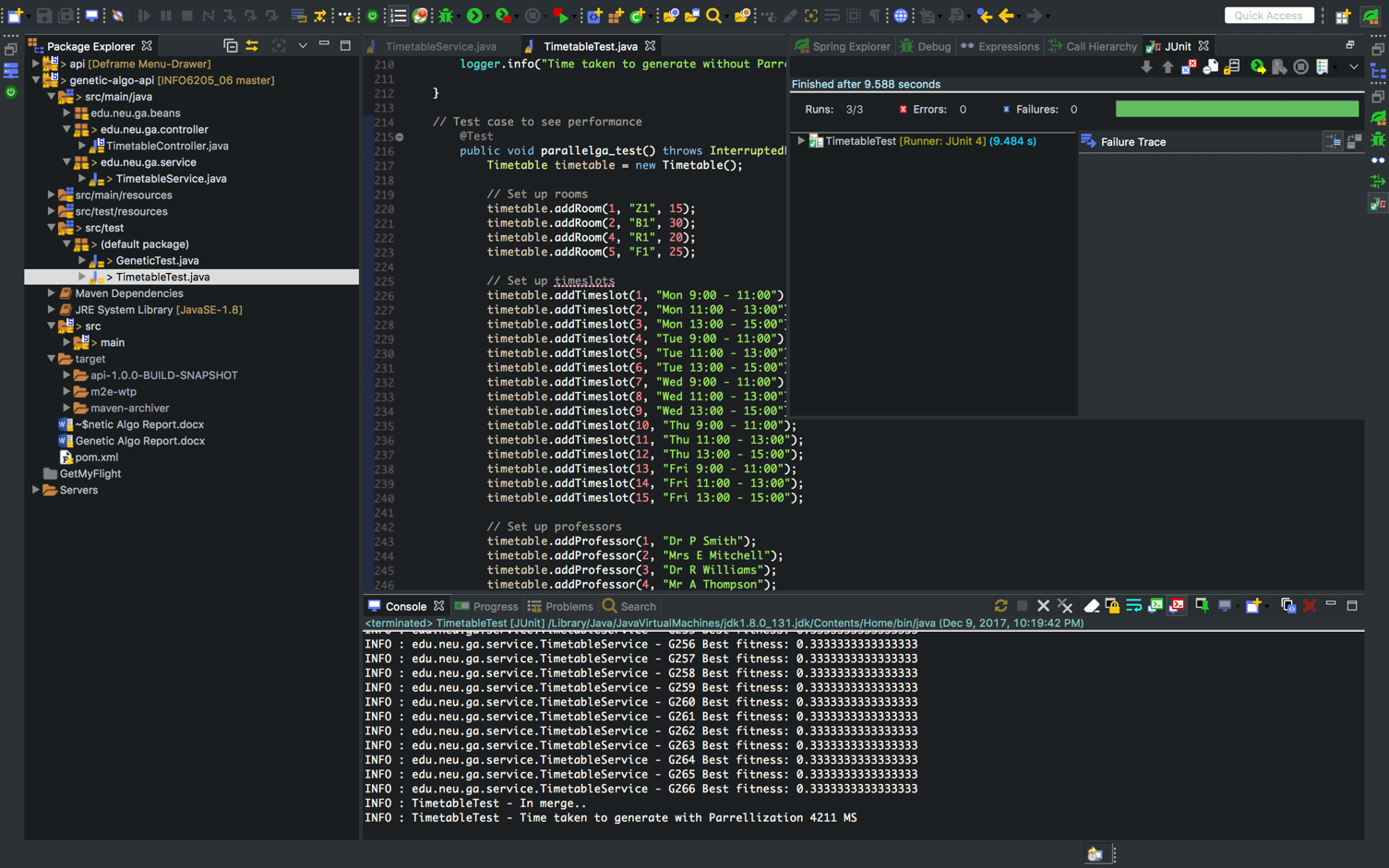


4) To Test Performance with various parameters

1. Single Threaded GA Test



1. Parallelized GA Test

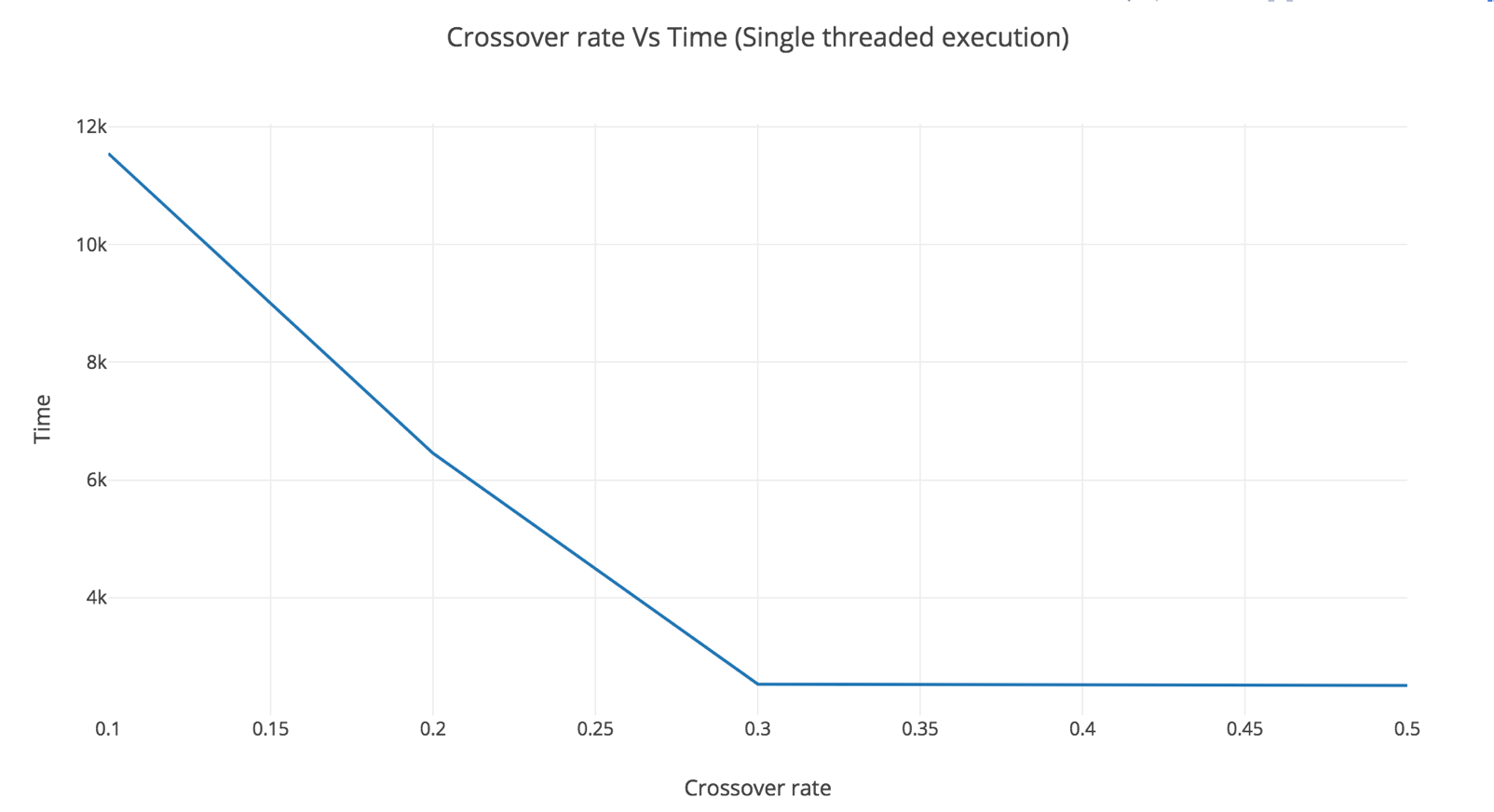


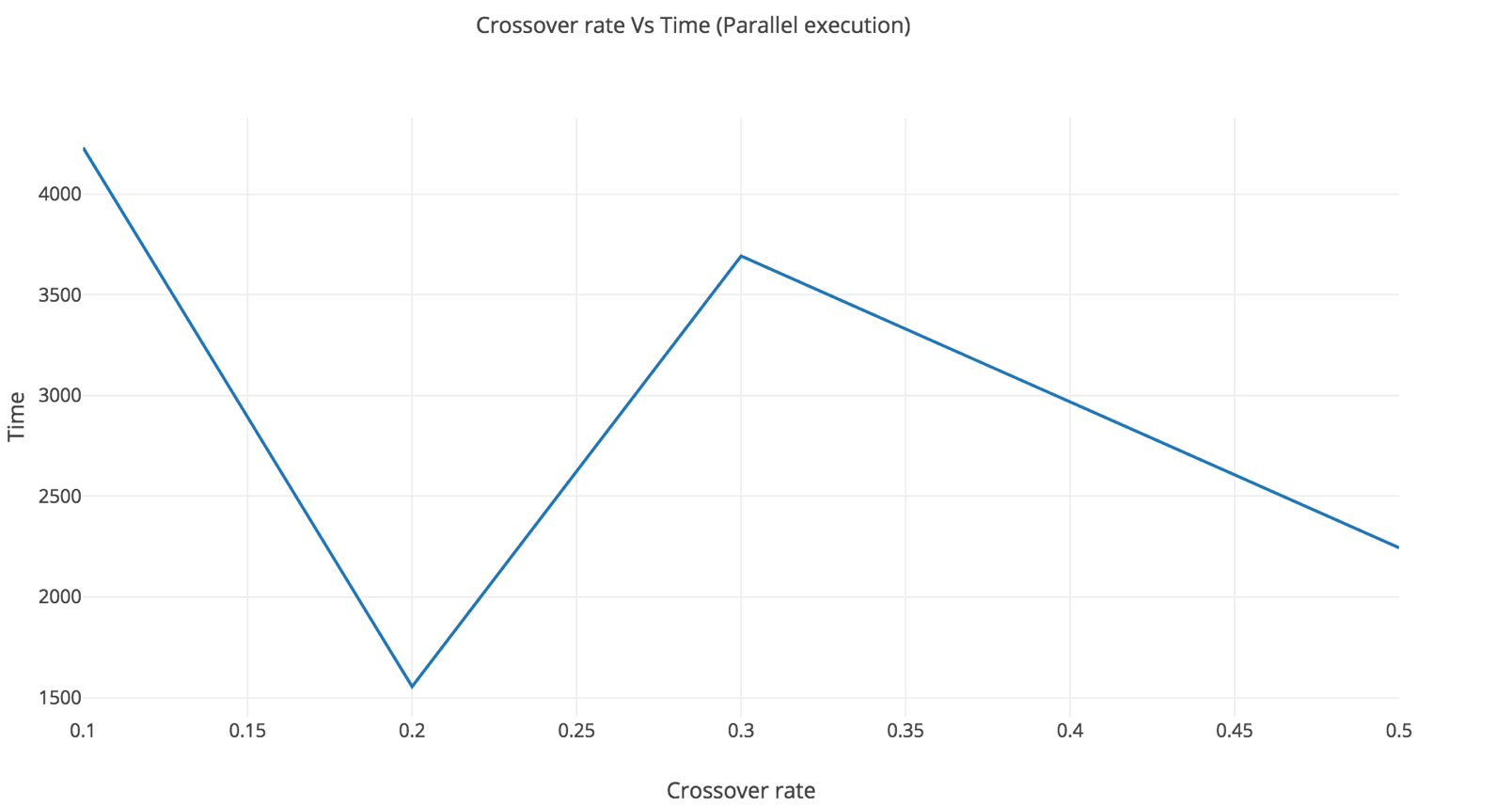
## 8. OBSERVATIONS & FINDINGs

1. When crossover is changed

GA – Population - 1000

|  |  |  |
| --- | --- | --- |
| **Crossover Rate** | **Single threaded execution(time in MS)** | **Parallel execution(time in MS)** |
| 0.1 | 11546 | 4231 |
| 0.2 | 6453 | 1555 |
| 0.3 | 2534 | 3693 |
| 0.5 | 2511 | 2244 |





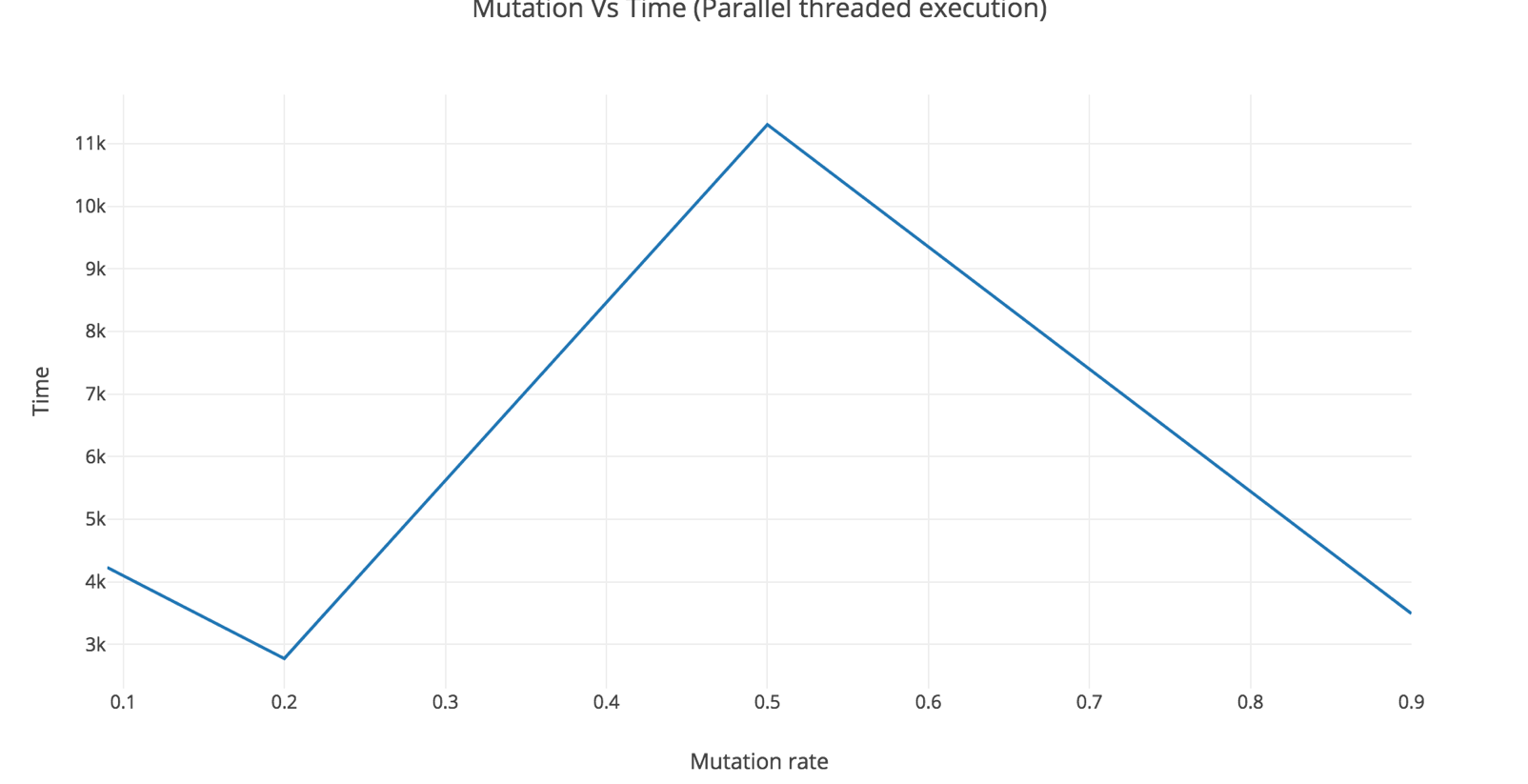
With changing crossover rates, there is change in execution time. It is found that the **parallel execution executes faster.**

1. When Mutation rate is changed

GA – Population – 1000

|  |  |  |
| --- | --- | --- |
| **Mutation Rate** | **Single threaded execution(time in MS)** | **Parallel execution(time in MS)** |
| 0.09 | 11546 | 4231 |
| 0.2 | 8427 | 2777 |
| 0.5 | 2730 | 11304 |
| 0.9 | 1762 | 3494 |

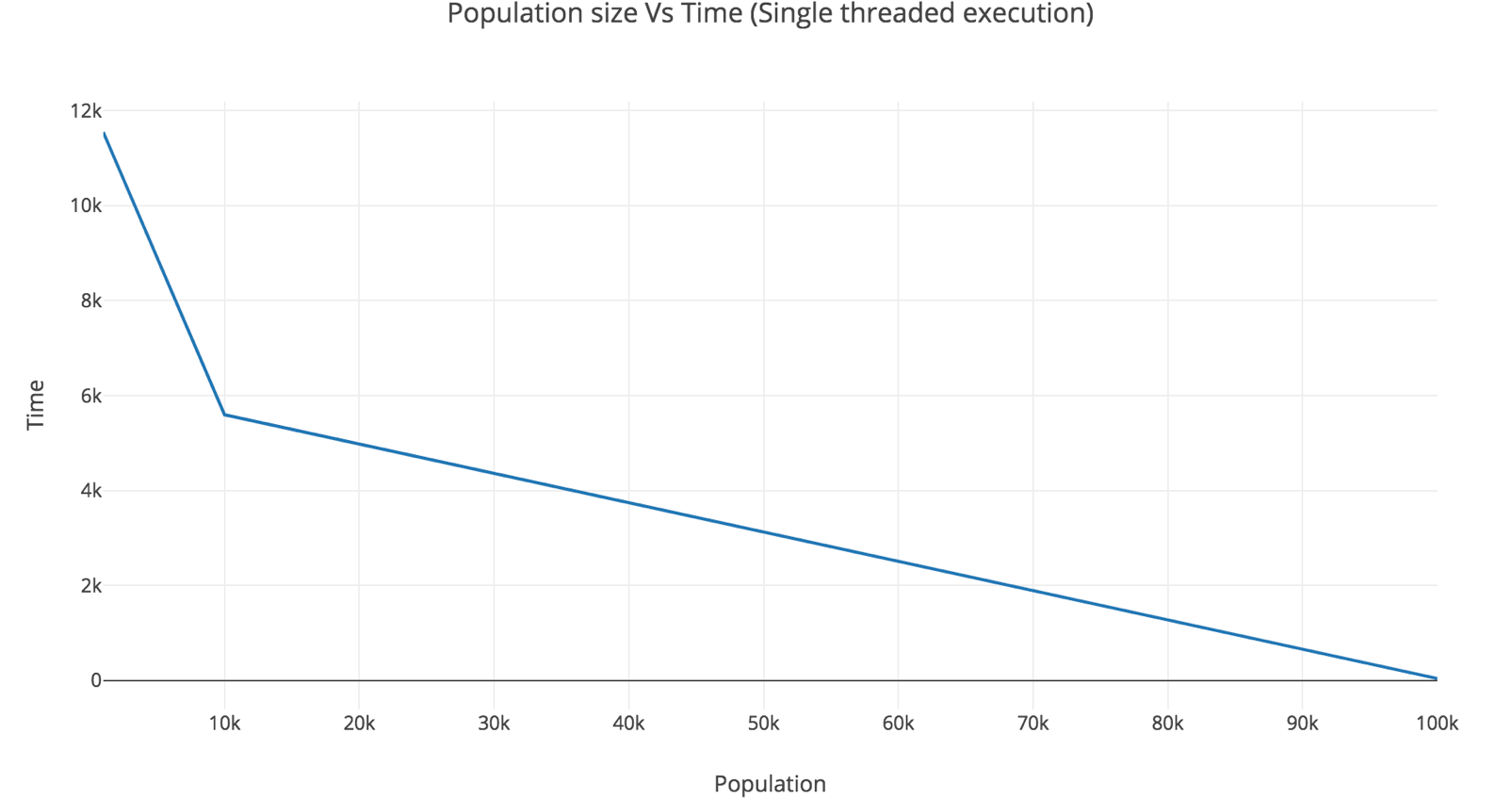


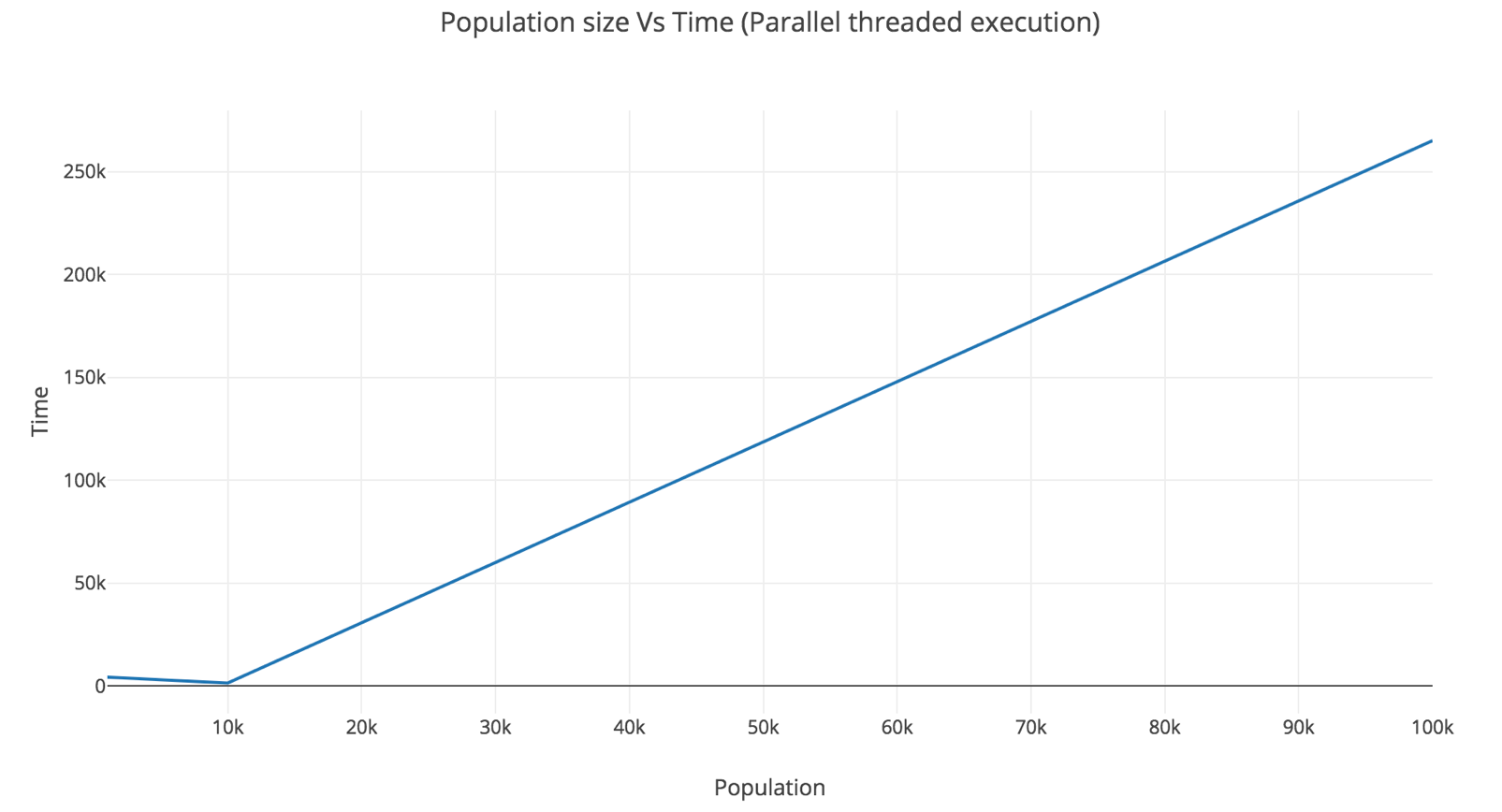


With changing mutation rates, there is change in execution time. It is found that the **parallel execution executes faster at lower mutation rate and single threaded executes faster at higher mutation rate.**

1. When Population is increased at CR – 0.1 and M.R 0.09

|  |  |  |
| --- | --- | --- |
| **Population size** | **Single threaded execution(time in MS)** | **Parallel execution(time in MS)** |
| 1000 | 11546 | 4231 |
| 10000 | 5597 | 1435 |
| 100000 | 44 | 265080 |

It is seen that with larger population the generations are evolved quickly and very efficiently. It is proved from the results that **Parallel execution** with larger generation performs well, but it seems that there may be a spike in performance at 100000. **While Single threaded execution is observed to be more stable of the two.**



## LITERATURE CITED

[1] Solving timetable scheduling problem using genetic algorithms [http://ieeexplore.ieee.org/document/1225396/](http://ieeexplore.ieee.org/document/1225396/?reload=true)

[2] Wikipedia

<https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications>

[3] Basics of Genetic Algorithm

<http://choonsiong.com/public/books/Algorithm/Genetic%20Algorithms%20in%20Java%20Basics.pdf>

[4] Demo of the project <https://www.codeproject.com/Articles/23111/Making-a-Class-Schedule-Using-a-Genetic-Algorithm>