TimetableGene.Rator
Project Documentation

Victor-Alexandru Padurean

December 2019

Contents

1 Introduction

The **TimetableGene.Rator** project was developed as the personal semestrial laboratory project for the Artificial Intelligence Course. The tool of choice is represented by **Genetic Algorithms**, whereas the subject this tool was used for trying to solve a real life problem is the school **Timetable Generation** problem.

The next sections are organized as follows: Bibliographic Research shows how **Genetic Algorithms** are used currently relevant projects, with emphasis on projects developed by the Technical University of Cluj-Napoca and the chosen tool implementation is presented; Analysis and Design regards the formulation of the **Timetable Generation** problem as comprehensible for an agent; Implementation tells how the tool is used in order to generate a good enough timetable in presence of the given constraints; Testing and Validation presents some unit tests, the project testing environment and the obtained results; and lastly Conclusion concludes the documentation and looks forward to further development.

2 Bibliographic Research

Genetic Algorithms, as the majority of A.I., concepts have their origins back in the mid-1900s. John Holland introduced genetic algorithms in 1960 based on the concept of Darwin's theory of evolution; afterwards, his student David E. Goldberg extended GA in 1989. [1] It started losing some ground to Machine Learning as all of the other concepts of A.I. have, lately, but nonetheless, it is still widely used for finding good enough solutions for NP-hard problems.

A good example of a domain in which these are used is present at our own university, at the Computer Science Department. Genetic Algorithms are used for solving optimization problems for server allocation for virtual machines (VM). [4] uses as genes VM to server mappings, as fitness function the total energy consumption and VM migrations and applies the classical operators on the population: evaluation, selection, crossover and mutation. Particle Swarm Algorithms, another type of evolutionary algorithms, similar to Genetic Algorithms, are also used in this article. In another such article, [2], genetic algorithms are used in the Plan phase of the MAPE architecture, for allocating tasks to servers, with a fitness function telling the number of constraints violated and energy consumed. Lastly, in [3], genetic algorithms are used for solving a Mixed-Integer-Non-Linear Program optimization problem, arising from three unknown variables (which are represented as chromosomes) in the thermal flow simulation that enables data centers to be used as heaters for nearby neighborhoods. The fitness function is the euclidean distance between the requested heat and the supplied heat.

The Java Jenetics Library was used as an implementation for the main concepts of Genetic Algorithms in the **TimetableGene.Rator** project. The tool is comprehensively presented in [6] by its own author. Some capabilities and other fields in which such algorithms are used are mentioned in [5].

3 Analysis and Design

The main two players of the Timetable Generation problem are the School Students Classes (which will be named Groups from now on, due to Object Oriented Programming reasons) and Teachers. Each of them will have sets of imperative constraints, that must be respected, and more permissive constraints, which should preferably be respected, if they don't interfere with the hard constraints. Further on, some schools have a restricted number of classes, so the teachers have to share (obviously, not at the same time) classes. For that, a new player can be added, the Room, which will also have a set of constraints.

The primary element of the algorithm will be the Course, an atomic entity (even though in some experiments, a Course's inner variables could be accessed

and changed by the evolutionary engine, this being mentioned later on), comprising a unit of time (an hour within a day), a Teacher, a Group and a Room. A solution will be a set of Courses that represents the timetable for an entire school during a whole week.

For assessing the correctness of a solution, the Constraint notion is defined. The Constraint checks a given timetable (either the timetable of a group, of a teacher or of a room) and raise penalties if some incompatibilities appear with its own constants. This constitutes the main interface for the user, that is, a user must specify at the beginning of the program which are their constraints for the given teachers and groups.

As mentioned before, there are different types of constraints. The hard constraints, if not respected, bring a higher penalty, whereas the soft constraints bring a lower one. The first constraint that was defined was the Superposition Constraint. This must be added by the agent for each player, as it states that no two different Courses shall take place in the same day, at the same hour, for any player. Another naturally occurring constraint is the Weekly Constraint for a Group, so that the students will take part to the classes that should be held in a week. This must be specified by the user for each Group, as different groups will have different classes to attend to.

Another type of constraint could the the Continuity Constraint, which enables the agent to group the courses tightly, so that school or high-school students will not have hour-long breaks and they will start their day from a reasonable hour, say 9 in the morning. This could also be applied for teachers, but the user will have to specify this, and it will be treated as a soft constraint, representing a teacher's preference for staying only a predetermined amount of time at school.

By adding an affiliation to the Teacher player, this can ensure that the student's day will be more diverse, so that fatigue and boredom doesn't take their place so fast. This brings about another type of soft constraint, the Diversity Constraint, which checks whether a day of a Group is diverse enough regarding the Teacher's Affiliation.

Figure 1 presents the problem as designed above, with the relationships between Course, Room, Teacher, Group and Constraints.

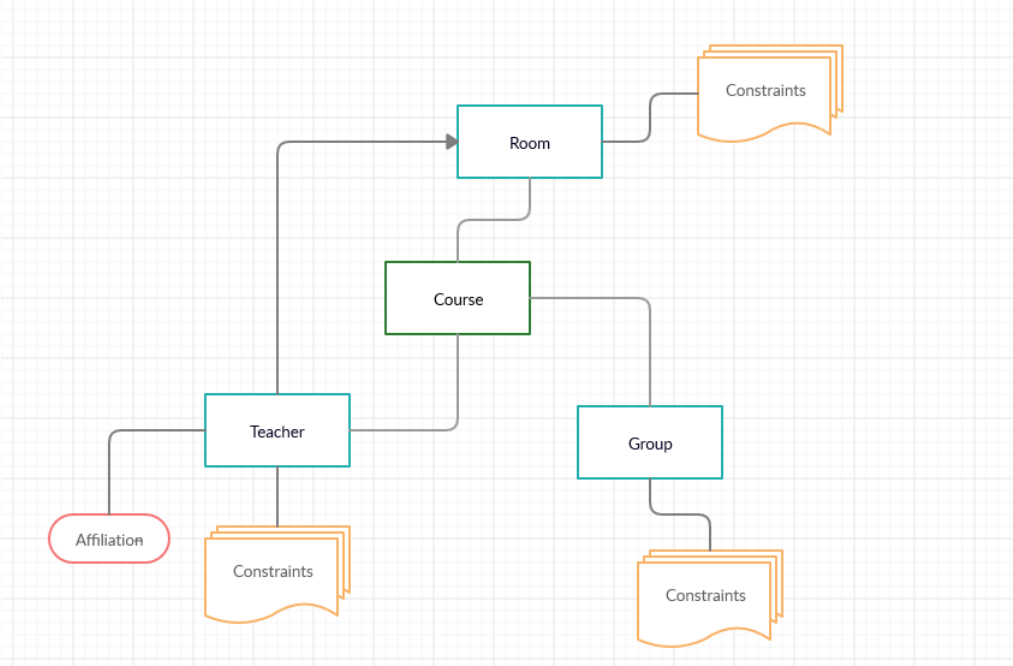

Figure 1: Problem Design

As observed, the name of the courses is nowhere mentioned. This is because with the current formulation of the problem, there is no need, as the subject will be represented by the name of a Teacher. The following pseudo-code will present how a user may interact with the agent, stating the players and constraints.

$Teacher_1 \leftarrow name_1; affiliation_1; rooms_1; continuityConstraints_1$
$Teacher_2 \leftarrow name_2; affiliation_2; rooms_2; continuityConstraints_2$
...
$Teacher_n \leftarrow name_n; affiliation_n; rooms_n; continuityConstraints_3$

$Group_1 \leftarrow name_{n+1}; teacherAmounts_{n+1} < teacher_a, i >, ..., < teacher_b, j >$
$Group_2 \leftarrow name_{n+2}; teacherAmounts_{n+2} < teacher_c, k >, ..., < teacher_d, l >$
...
$Group_m \leftarrow name_{n+m}; teacherAmounts_{n+m} < teacher_e, o >, ..., < teacher_f, p >$

4 Implementation

The players and constraints presented in Analysis and Design where implemented as classes and interfaces with regard to Object Oriented Programming

paradigm and Design Patterns. Of course, additional auxiliary classes were also used, but this section emphasises the implementation of the main classes that enable the evolutionary engine to be used, mainly: CourseGene, TimetableChromosome, Fitness and Genetic. Classes that are needed by the reader to fully understand the implementation will also be briefly presented.

CourseGene is the custom gene of the algorithm, implementing the library's Gene interface, and having as allele (internal information) a Course class. The Course class respects the design mentioned above. The present methods are essential for the integration of the custom gene class into the evolutionary process brought by the library. The UML diagram of the Course class is presented in Figure 2.

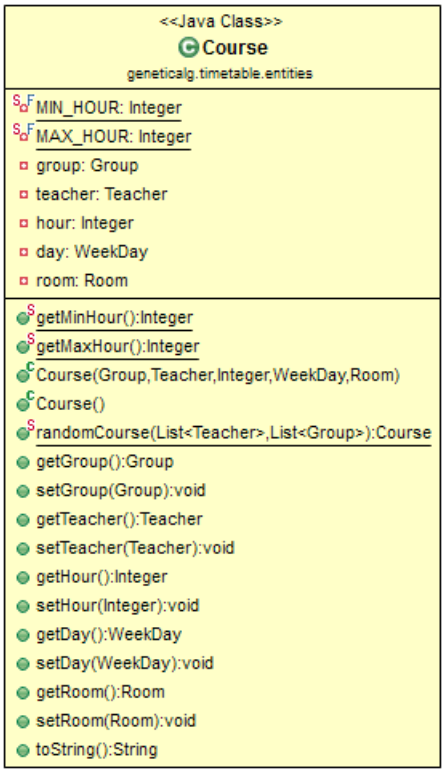

Figure 2: Course Class UML

TimetableChromosome is the custom chromosome made as a sequence of CourseGenes, and it implements the library's Chromosome interface. It overrides the necessary methods for it to be integrated into the evolutionary engine. Notably, the *newInstance()* method uses the Course's *randomCourse()* method for creating a random course by picking teachers, groups and other necessary

attributes randomly. An individual will have only one chromosome and it will represent the timetable for the entire school over the course of a whole week.

Fitness is the class responsible for assessing the fitness of one individual. It collects the information present in the chromosome of an individual, a list of courses, and groups the courses for each teacher, group and room. Then, by using the Constraint's interface *checkFitnessOneChromosome()* method, it penalizes the individual proportional to the number of broken constraints. Listing 1 shows the implementation of this method.

```
1  public static Long checkFitnessOneChromosome(Genotype<CourseGene>
       genotype) {
2      TimetableChromosome c = genotype.getChromosome().as(
       TimetableChromosome.class);
3      Map<Teacher, List<Course>> teacherTimetables = new HashMap<
       Teacher, List<Course>>();
4      Map<Group, List<Course>> groupTimetables = new HashMap<Group,
       List<Course>>();
5      Map<Room, List<Course>> roomTimetables = new HashMap<Room, List
       <Course>>();
6      c.stream().forEach(x -> {
7        Course course = x.getAllele();
8        if (course != null) {
9          if (teacherTimetables.get(course.getTeacher()) == null) {
10           teacherTimetables.put(course.getTeacher(), new ArrayList<
       Course>());
11         }
12         teacherTimetables.get(course.getTeacher()).add(course);
13         if (groupTimetables.get(course.getGroup()) == null) {
14           groupTimetables.put(course.getGroup(), new ArrayList<
       Course>());
15         }
16         groupTimetables.get(course.getGroup()).add(course);
17         if (roomTimetables.get(course.getRoom()) == null) {
18           roomTimetables.put(course.getRoom(), new ArrayList<Course
       >());
19         }
20         roomTimetables.get(course.getRoom()).add(course);
21       } else {
22         System.out.println(c);
23       }
24     });
25     Long sum = 0L;
26     sum += teacherTimetables.entrySet().stream()
27         .collect(Collectors.summingLong(x -> x.getKey().
       checkConstraints(x.getValue())));
28     sum += groupTimetables.entrySet().stream()
29         .collect(Collectors.summingLong(x -> x.getKey().
       checkConstraints(x.getValue())));
30     sum += roomTimetables.entrySet().stream()
31         .collect(Collectors.summingLong(x -> x.getKey().
       checkConstraints(x.getValue())));
32     if (sum < 0) {
33       System.out.println("???");
```

```
    }
    if (sum < 0)
      sum = Long.MAX_VALUE;
    return sum;
  }
```

Listing 1: Method that assesses an individual

The Constraint implemented as an interface is a key concept, as it enables players have a set with all their constraints, no matter what they check, and treat them all the same. They only have to check each Constraint on the given timetable. This means that multiple classes will have to implement the Constraint interface and override the *checkConstraint()* method. This approach is depicted in Figure 3.

The method for penalizing the failure in respecting the given constraints can be chosen, as the Constraint interface uses the PenaltyStrategy interface (based on the Strategy Design Pattern). There are two options: Exponential Penalty and Multiplicative Penalty. Both of them count how many Courses from the given timetable break a constraint at the same time and then use the constants *HARD_CONSTRAINT* and *SOFT_CONSTRAINT* to compute the penalty. The exponential one uses the obtained number as exponent for the corresponding constant, whereas the multiplicative one uses it as a factor.

Genetic is the class that assembles all the details of the evolutionary engine. First, it computes how long a chromosome should be, that is how many Courses are there in a week for the whole school, then it instantiates the Jenetics Library's Genotype Factory. The factory will be used by the engine to provide individuals for the evolutionary process. The Engine is then set up, receiving the fitness function used for assessing individuals, the factory, selectors and alterers. Listing 2 presents the *runEvolution()* method of the Genetic class.

```
public static void runEvolution(Integer minutes) {

    Integer totalHours = Group.getGroups().stream().collect(
    Collectors.summingInt(x -> x.getNoHours()));

    Factory<Genotype<CourseGene>> factory = Genotype
        .of(TimetableChromosome.of(CourseGene.seq(totalHours, Group
    .getGroups(), Teacher.getTeachers())));

    Engine<CourseGene, Long> engine = Engine.builder(Fitness::
    checkFitnessOneChromosome, factory).minimizing()
        .survivorsSelector(new EliteSelector<>())
        .offspringSelector(new RouletteWheelSelector<>())
        .alterers(new Mutator<>(0.6), new MultiPointCrossover
    <>(0.4)).build();

    Consumer<? super EvolutionResult<CourseGene, Long>> statistics
    = EvolutionStatistics.ofNumber();

```

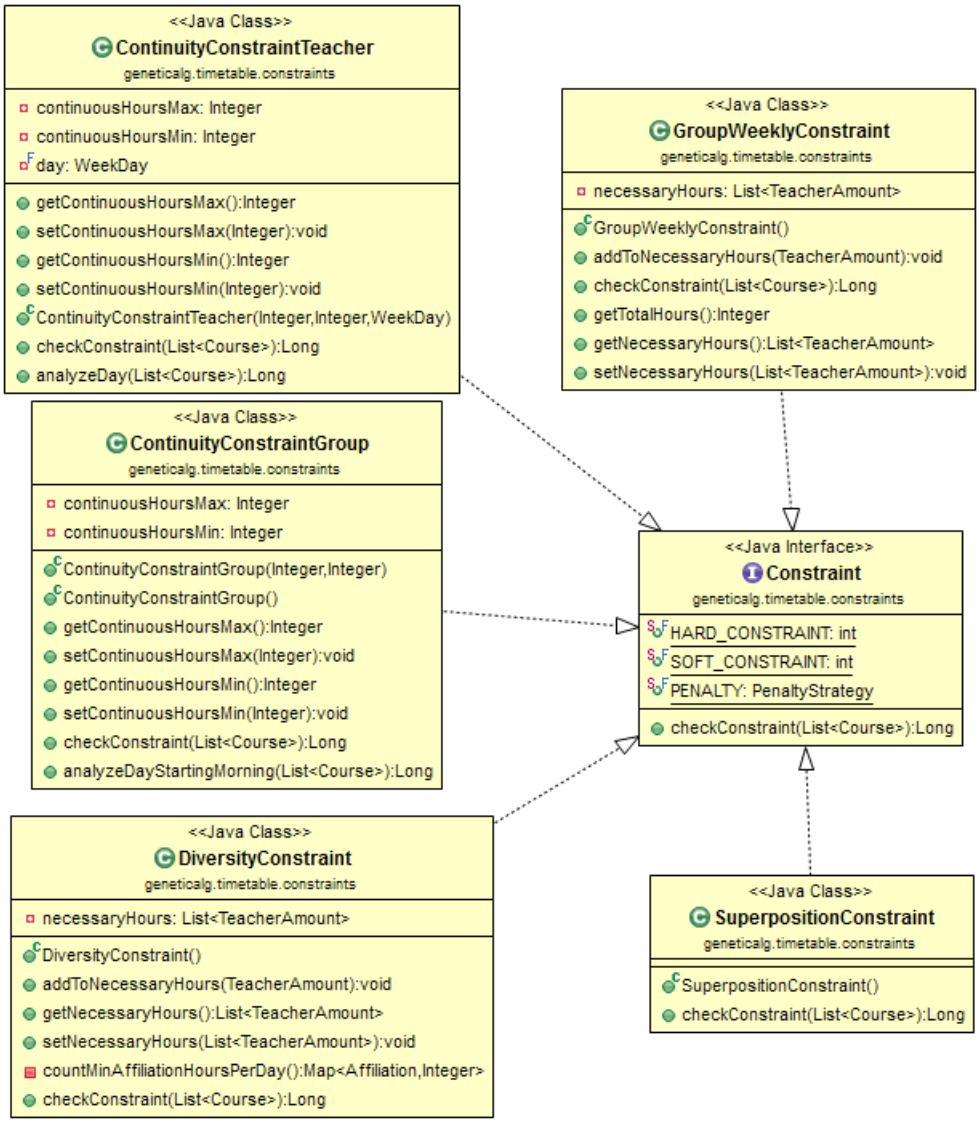

Figure 3: Constraint and Realization classes UML

```
15      Genotype<CourseGene> result = engine.stream().limit(Limits.
        byExecutionTime(Duration.ofMinutes(minutes)))
16          .limit(Limits.byFitnessThreshold(1L)).limit(Limits.
        bySteadyFitness(50000)).peek(statistics).collect(
        EvolutionResult.toBestGenotype());
17
18      SolutionHandler.handleResult(result);
19      System.out.println(Fitness.checkFitnessOneChromosome(result));
20      System.out.println(statistics);
21  }
```

Listing 2: Method that sets up and runs the evolution process

The reader should also notice that the score of an individual should be minimized, that is individuals with a lower penalty given by the fitness function are more fit for solving the problem. By using an elite selector, the most fit individual found will be always preserved. This is useful because, if the evolution goes on an unpromising path, making the individual less fit, the algorithm keeps track of the best solution found. This can lead to local solutions, though. The roulette wheel is a classical selection process in genetic algorithms. There should be a high chance of mutation as this is the only way of bringing new courses to the timetable if the current set cannot constitute a good enough solution. The multi point crossover is used for a more rapid evolution. There was an attempt of creating a custom crossover method, which could swap the insides of two genes by accessing the individual variables inside a Course. The Testing and Validation section will show why this implementation was not used.

Finally, the result is obtained by limiting the duration of the evolution in time, or if the perfect timetable was reached, obtaining a score less than 1, or if a better solution could not be found for 50000 generations. The result is handled by a special class and pretty printed in the console. The statistics are also shown, this being a built in feature of the library.

5 Testing and Validation

Testing the constraints classes was of utter importance, as the fitness function, which is based on these, may be the most important implementation part for this genetic algorithm. As a consequence, some drivers were implemented to prove that each constraint and penalization computation works correctly. There is a tester class for each of the Constraint interface implementation.

The main tester class simulates a simple scenario, where a school with only two groups and several teachers and rooms needs a timetable. Each group has the natural, superposition constraint, and the constraint added by the user, the weekly constraint and the continuity constraint. The diversity constraint is not explicitly added by a user, but it should be automatically added with the weekly constraint. The tester also adds simple constraints for two teachers, limiting the number of hours their are available in a single and in every day

of the week, respectively. Listing 3 may help the reader understand the inner workings better.

```
public static void main(String[] args) {
    Teacher t1 = new Teacher("Maths1", Affiliation.Sciences);
    Teacher t2 = new Teacher("English1", Affiliation.Languages);
    Teacher t3 = new Teacher("Physics", Affiliation.Sciences);
    Teacher t4 = new Teacher("Maths2", Affiliation.Sciences);
    Teacher t5 = new Teacher("ComputerScience", Affiliation.
    Sciences);
    Teacher t6 = new Teacher("Music", Affiliation.Other);
    Teacher t7 = new Teacher("GraphicDesign", Affiliation.Other);
    Teacher t8 = new Teacher("French", Affiliation.Languages);
    Teacher t9 = new Teacher("Geography", Affiliation.Anthropology)
    ;
    Teacher t10 = new Teacher("History", Affiliation.Anthropology);
    t1.addRoom(new Room("1"));
    t2.addRoom(new Room("2"));
    t3.addRoom(new Room("3"));
    t4.addRoom(new Room("1"));
    t5.addRoom(new Room("5"));
    t6.addRoom(new Room("4"));
    t7.addRoom(new Room("4"));
    t8.addRoom(new Room("6"));
    t9.addRoom(new Room("7"));
    t10.addRoom(new Room("7"));
    t9.addRoom(new Room("8"));
    t8.addRoom(new Room("2"));
    List<TeacherAmount> how1 = Arrays.asList(new TeacherAmount(t1,
    5), new TeacherAmount(t2, 5),
        new TeacherAmount(t3, 3), new TeacherAmount(t5, 4), new
    TeacherAmount(t6, 2), new TeacherAmount(t7, 1),
        new TeacherAmount(t8, 3), new TeacherAmount(t9, 2), new
    TeacherAmount(t10, 2));
    List<TeacherAmount> how2 = Arrays.asList(new TeacherAmount(t4,
    4), new TeacherAmount(t2, 4),
        new TeacherAmount(t3, 5), new TeacherAmount(t6, 1), new
    TeacherAmount(t7, 3), new TeacherAmount(t8, 4),
        new TeacherAmount(t9, 2), new TeacherAmount(t10, 2));
    GroupWeeklyConstraint c1 = new GroupWeeklyConstraint();
    c1.setNecessaryHours(how1);
    GroupWeeklyConstraint c2 = new GroupWeeklyConstraint();
    c2.setNecessaryHours(how2);
    ContinuityConstraintGroup cc = new ContinuityConstraintGroup(4,
     7);
    ContinuityConstraintTeacher ct = new
    ContinuityConstraintTeacher(3, 4, WeekDay.TUESDAY);
    ContinuityConstraintTeacher ct2 = new
    ContinuityConstraintTeacher(2, 3, WeekDay.WORKDAYS);
    t3.addConstraint(ct);
    t2.addConstraint(ct2);
    DiversityConstraint dc1 = new DiversityConstraint();
    dc1.setNecessaryHours(how1);
    DiversityConstraint dc2 = new DiversityConstraint();
    dc2.setNecessaryHours(how2);
    Group g1 = new Group("12A");
```

```
    g1.addConstraint(c1);
    g1.addConstraint(cc);
    g1.addConstraint(dc1);
    Group g2 = new Group("12B");
    g2.addConstraint(c2);
    g2.addConstraint(cc);
    g2.addConstraint(dc2);
    Genetic.runEvolution(5);
  }
```

Listing 3: Simple tester

The results of the tests are pleasing, as the algorithm succeeds in solving most of the constraints and coming up with a good looking timetable in a reasonable amount of time. Of course, if the example is more complex, the amount of time needed for coming up with a good solution grows. Figures 4 and 5 show the results of a run of the tester class.

```
      12B|          9||         10||         11||         12||         13||         14||         15||         16|
   MONDAY|  3  Physics|  4 GraphicDesign|  3  Physics|  2  English1|  8  Geography|             |             |
  TUESDAY|  4    Music|  6    French|  6    French|  7    History|  6    French|  1   Maths2|             |
WEDNESDAY|  3  Physics|  4 GraphicDesign|  6    French|  1    Maths2|  2  English1|  2  English1|             |
 THURSDAY|  2 English1|  4 GraphicDesign|  8 Geography|  3   Physics|             |             |             |
   FRIDAY|  1   Maths2|  3   Physics|  7   History|  1   Maths2|             |             |             |

      12A|          9||         10||         11||         12||         13||         14||         15||         16|
   MONDAY|  4    Music|  1   Maths1|  7   History|  1   Maths1|  2  English1|  2  English1|  5 ComputerScience|
  TUESDAY|  2   French|  2  English1|  7 Geography|  3   Physics|  1   Maths1|             |             |
WEDNESDAY|  4    Music|  3   Physics|  5 ComputerScience|  2  English1|             |             |             |
 THURSDAY|  3  Physics|  2  English1|  4 GraphicDesign|  1   Maths1|  7 Geography|  1   Maths1|             |
   FRIDAY|  7   History|  2   French|  6    French|  5 ComputerScience|  5 ComputerScience|             |             |
```

Figure 4: The timetable obtained for two classes

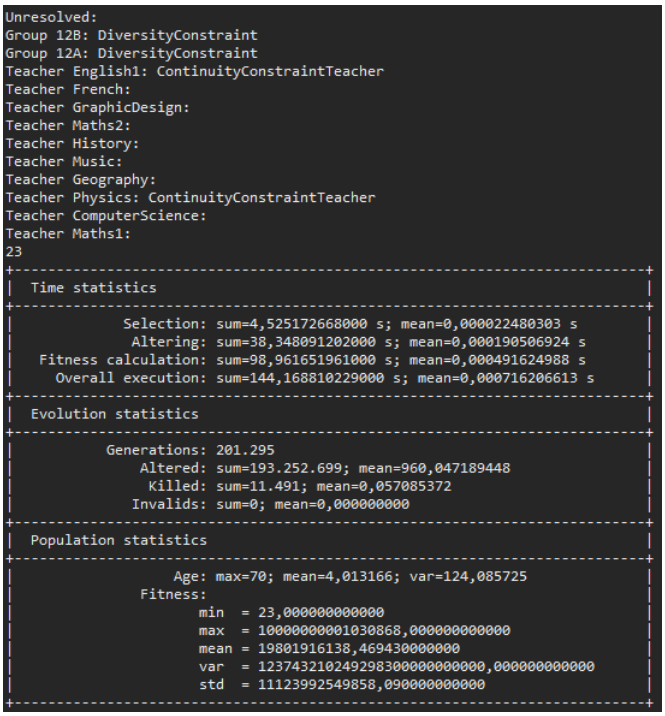

Figure 5: The unresolved constraints and the statistics

To obtain better results, two modifications were tried: eliminating inviable individuals (those that do not satisfy hard constraints), and the custom, gene modifying crossover. They both failed in surpassing the main approach, even obtaining displeasing results, as the penalty score was much higher. Consequently, the main approach was kept. In addition, the multiplicative penalty strategy was tested along with the exponential one. The exponential strategy obtained better results.

6 Conclusion

The **TimetableGene.Rator** project demonstrates how genetic algorithms can be used for solving quite complex problems, like the timetable generation, which annoys school teachers for quite some days at the beginning of each school year.

Further on, the approach can be improved by testing whether, given constraints, a timetable can be generated in the first place, or if the constraints are too tight for this. Another improvement that can be added is a whole new problem that can be solved: given a timetable and some further constraints, find the minimum number of swaps that transform the timetable in order to

satisfy the given constraints.

References

[1] Genetic algorithm - wikipedia.

[2] Marcel Antal, Adelina Burnete, Claudia Pop, Tudor Cioara, Ionut Anghel, and Ioan Salomie. Self-adaptive task scheduler for dynamic allocation in energy efficient data centers. *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2017.

[3] Marcel Antal, Tudor Cioara, Ionut Anghel, Claudia Pop, and Ioan Salomie. Transforming data centers in active thermal energy players in nearby neighborhoods. *Sustainability*, 2018.

[4] Antal Marcel, Pintea Cristian, Pintea Eugen, Pop Claudia, Tudor Cioara, Ionut Anghel, and Salomie Ioan. Thermal aware workload consolidation in cloud data centers. *2016 IEEE 12th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2016.

[5] Victor-Alexandru Padurean. Java jenetics library - capabilities presentation.

[6] Franz Wilhelmstötter. *Jenetics Library User's Manual 5.0*.