Schempus

Giuseppe Sica

September 28, 2025

Contents

1 Introduction

Scheduling university courses is a highly complex task. Typically, a small group of individuals is responsible for organizing a large number of classes and coordinating with teachers, all while considering numerous variables and constraints. This process is time-consuming and prone to errors. **Schempus** was created to simplify and accelerate this process by leveraging the power of computational intelligence. By providing the necessary data to the system, Schempus can automatically generate optimized timetables, reducing the manual effort required.

1.1 Goal of the Project

The primary goal of this project is to develop an Artificial Intelligence (AI) system capable of generating a timetable for a university department.
The system must allocate all lessons to appropriate time slots and classrooms, ensuring optimal scheduling.
By addressing these constraints, the system aims to create a timetable that is efficient, complete, and free of errors.

2 Environment Analysis

2.1 P.E.A.S Specification

Agent Type:
Timetable resolver for university scheduling.

Performance Measure:

- **Classroom Capacity**: Each classroom must be large enough to accommodate all students in a given class, but not excessively large. For example, a class with 150 students cannot be scheduled in a room with only 100 seats, and a class with 50 students should not be scheduled in a room with 200 or more seats.
- **Collision Avoidance**: The AI must avoid collisions (more than one lesson at the same hour, day, and classroom).
- **Professor's Conflict Avoidance**: The AI must avoid overlapping lessons by the same professor. For example, one professor should not have two lessons scheduled at the same time in different classrooms.
- **Week Distribution**: The AI must ensure that each course has 2 hours of theory or 3 hours of laboratory each day, without exceeding one such block per day. For instance, it cannot allocate six consecutive hours to the same course in a single day, nor can it allocate both two hours of theory and three hours of laboratory in the same day.
- **Distribution in Day**: The AI must ensure that lessons for the same course on the same day are consecutive and in the same classroom. For instance, if a course has three hours of laboratory work, those hours should be consecutive in the same laboratory, without moving to another one.
- **Laboratory Allocation**: The AI must ensure that theory lessons are scheduled in normal classrooms, whereas laboratory work must be allocated to a laboratory. Furthermore, if the lesson is in a laboratory, it should be the correct one. For instance, a Chemistry class should be in a Chemistry lab, not in a Computer Science lab.

Environment:

- A software system with a user-defined input dataset that includes:
  - Classrooms and Laboratories
  - Courses
  - Professors

Actuators:

- Timetable entries: The system assigns lessons to specific time slots and classrooms.
- Output interface: The finalized timetable is presented to users for review.

Sensors:

- Current timetable state: The system continuously monitors the arrangement to check for conflicts or unmet constraints during the scheduling process.

2.2 Environment's Characteristics

The environment is:

- **Completely Observable:** The agent has access to all relevant information (e.g., constraints, room capacities) and can observe the entire timetable at any point.
- **Deterministic:** The agent's actions have predictable, specific effects on the timetable without randomness.
- **Sequential:** Each decision affects future actions, as the availability of resources (e.g., classrooms, time slots) depends on previous assignments.
- **Static:** The environment remains unchanged unless the agent takes actions to update it.
- **Discrete:** The environment has a finite, countable state space (e.g., specific classrooms, time slots, and lessons).
- **Single Agent:** The scheduling system operates as the sole agent responsible for creating the timetable.

3 Genetic Algorithm

3.1 Technologies

The technology used for this project is simple, pure Python. Given the specificity of the problem, finding an external library was nearly impossible, so the algorithm for the agent has been developed from scratch.

3.2 Agent Definition

Defining the agent was the most challenging part of the project. Initially, I tried a complex structure of classes and subclasses, but it quickly became too cumbersome and difficult to maintain.
My second approach was defining the timetable (the agent) as a list of Courses. Each Course was itself a list of Lessons, where each Lesson was a list of three integers: day, classroom, and hour, in that order.
This approach was correct because it facilitated the definition of functions such as Mutation and Crossover. However, after discussing with my AI professor, a new problem arose: how to define the laboratory hours for courses that have them.

Two methods were considered to define lab hours:

1. Create, for each course, a second "lab course" that doubles the number of courses in the schedule and treat it just like any other course.

2. Adapt the previous definition by adding a fourth element to each lesson (a boolean indicating whether it is a laboratory session).

The first approach was simpler but significantly increased the problem's complexity, especially with **week distribution**, because lab and theory hours would not be in the same course.
The second approach was to adapt the previous definition by adding the boolean element. To ensure integrity and consistency, this value must not be modified by the Crossover or Mutation functions, requiring special attention in those.
Despite that, I preferred to use the second approach because it was simpler to adapt and maintained closer ties between theory and lab hours.
Listing 1 shows how the agent is initialized.

The initialization of the agent is completely random, as shown here:

```
def __init__(self, classrooms, courses):
    """This method initializes the timetable with a list
    of Classroom and a list of Course"""
    import random as rd
    self.classrooms = classrooms
    self.courses = courses
    self.timetable = []
    # Generate a random timetable
    for course in courses:
        _course = []
        _lab_hours = course.lab_hours
        for _ in range(course.hours_for_week):
            # [day, classroom, hour, lab = True while lab_hours > 0 else False]
            _course.append([
                rd.randint(0, 4),
                rd.randint(0, len(classrooms)-1),
                rd.randint(0, 7),
                True if _lab_hours > 0 else False
            ])
            _lab_hours -= 1
        self.timetable.append(_course)
```

Listing 1: init of the agents

3.3 Crossover

The goal of the crossover function is to combine the genes of two agents, hoping the result will yield a better evaluation via the fitness function. Although this sounds straightforward, in this problem, we cannot simply take two agents and mix their timetables because each course must retain the user-defined number of hours.
This was the primary reason for defining the agent as explained above. In fact, this representation makes it possible to perform a crossover on a course-by-course basis, leaving the number of hours untouched and swapping lessons between two agents.
Because of the fourth element in each lesson (indicating whether it is a laboratory session), the crossover swaps only the first three elements (day, classroom, hour), leaving the boolean value unchanged to maintain consistency in lab hours.
Another question was whether to swap only a single course at a time or multiple courses. After some testing, I found that a **multi-course swap** led to better results, as shown in Figures 1 and 2.

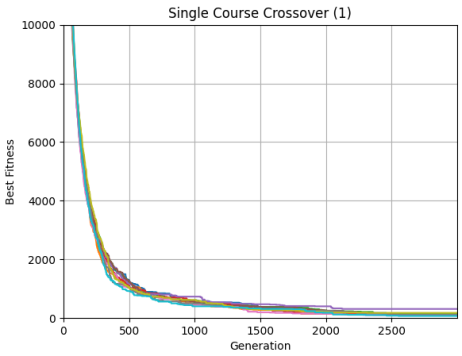

Figure 1: Mean result: 144.76

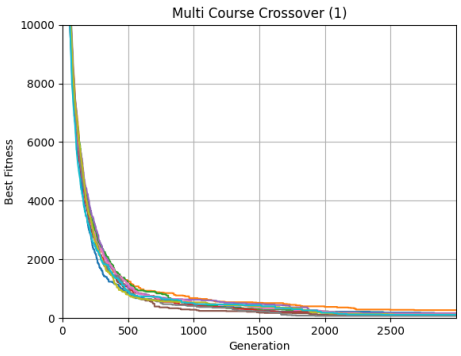

Figure 2: Mean result: 138.82

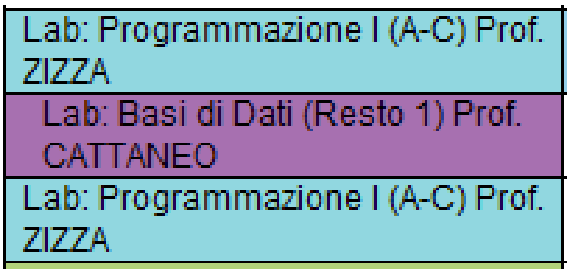

Figure 3: Example of intertwined lessons

3.4 Mutation

The mutation function is fundamental in any genetic algorithm. In this problem, I experimented with different approaches. In the first approach, a single random lesson was selected from the timetable and randomly re-assigned to a different time slot.
This was the first version:

```
def mutation(agent: Timetable, number_of_mutations=1):
    for _ in range(number_of_mutations):
        # Select a random lesson
        course_index = rd.randint(0, len(agent.timetable) - 1)
        lesson_index = rd.randint(0, len(agent.timetable[course_index]) - 1)
        old_lesson = agent.timetable[course_index][lesson_index]
        # Mutate the lesson
        new_lesson = [
            rd.randint(0, 5),
            rd.randint(0, len(agent.classrooms)),
            rd.randint(0, 7),
            old_lesson[3]
        ]
        agent.timetable[course_index][lesson_index] = new_lesson
```

Listing 2: Mutation 1.0

After extensive testing, I observed that the algorithm struggled to resolve situations in which a lesson was "intertwined" with another (Figure 3). This led me to consider a more dynamic approach that changes the randomness as generations progress.

I eventually adapted my function to emulate Simulated Annealing, which starts with a

high degree of randomness and gradually reduces it while converging on a more promising solution. In my case, I used this approach to "move" a lesson by "large steps" in the early generations and by "small steps" in later generations.

To implement this approach, I defined the randomness as follows:

```
day_mutation = NUMBER_OF_DAYS - int(NUMBER_OF_DAYS / (generations / i))
class_mutation = NUMBER_OF_CLASSES - int(NUMBER_OF_CLASSES / (generations / i))
hour_mutation = NUMBER_OF_HOURS - int(NUMBER_OF_HOURS / (generations / i))
```

Listing 3: Mutation's values definition

For example, consider `day_mutation`:

- `NUMBER_OF_DAYS` $= 5$ (the number of days in the schedule)
- `generations` = number of total generations (e.g., 100)
- `i` = current generation (from 1 to `generations`)

If `generations` $= 100$ and `i` $= 1$, then

$$\texttt{day\_mutation} = 5 - \texttt{int}\left(\frac{5}{\frac{100}{1}}\right) = 5 - \texttt{int}\left(\frac{5}{100}\right) = 5 - \texttt{int}(0.05) = 5 - 0 = 5.$$

Hence, the algorithm will generate a random number between 0 and 5.
If `generations` $= 100$ and `i` $= 79$, then

$$\texttt{day\_mutation} = 5 - \texttt{int}\left(\frac{5}{\frac{100}{79}}\right) = 5 - \texttt{int}\left(\frac{5}{1.26}\right) = 5 - \texttt{int}(3.9) = 5 - 3 = 2.$$

So, the algorithm will generate a random number between 0 and 2.
Lastly, if `generations` $= 100$ and `i` $= 80$, then

$$\texttt{day\_mutation} = 5 - \texttt{int}\left(\frac{5}{\frac{100}{80}}\right) = 5 - \texttt{int}\left(\frac{5}{1.25}\right) = 5 - \texttt{int}(4) = 5 - 4 = 1.$$

So, the algorithm will generate a random number between 0 and 1.

The mutation function then evolved as follows:

```
def mutation(agent: Timetable, day_mutation, class_mutation, hour_mutation,
    number_of_mutations=1):
    for _ in range(number_of_mutations):
        # Select a random lesson
        course_index = rd.randint(0, len(agent.timetable) - 1)
        lesson_index = rd.randint(0, len(agent.timetable[course_index]) - 1)
        old_lesson = agent.timetable[course_index][lesson_index]

        # Mutate the lesson
        new_lesson = [
            (old_lesson[0] + rd.randint(-day_mutation, day_mutation)) %
                NUMBER_OF_DAYS,
            (old_lesson[1] + rd.randint(-class_mutation, class_mutation)) % len(
                agent.classrooms),
            (old_lesson[2] + rd.randint(-hour_mutation, hour_mutation)) %
                NUMBER_OF_HOURS,
            old_lesson[3]
        ]
        agent.timetable[course_index][lesson_index] = new_lesson
```

Listing 4: Mutation 2.0

As shown by Fig.4 and Fig.5, the final results are similar, but the simulated-annealing-style approach gets better results at the end.

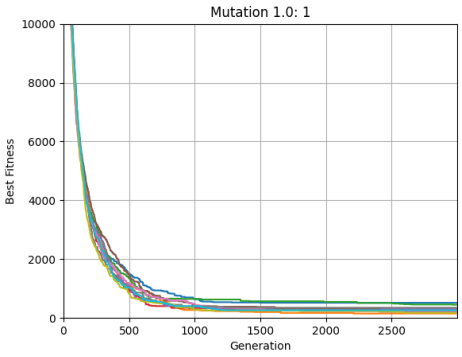

Figure 4: Mean result: 369.05

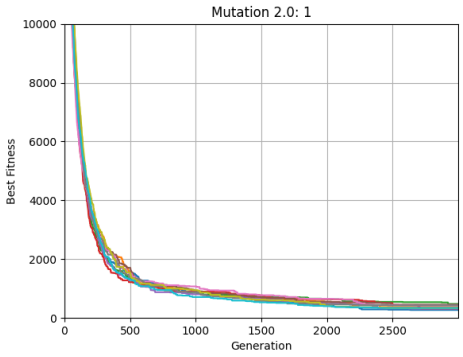

Figure 5: Mean result: 354.16

Despite the change, the problem of intertwined lessons was not solved. In fact, during the last generations, those lessons were difficult to fix because the agent could only change their positions (if a lesson was moved into an occupied position, it created a collision, so that attempt was discarded). To address this, the mutation was modified to check if the new position collides with another lesson. If so, the two lessons swap their positions. Figures 6 and 7 show the different speeds of convergence toward 0 and demonstrate better fine-tuning.

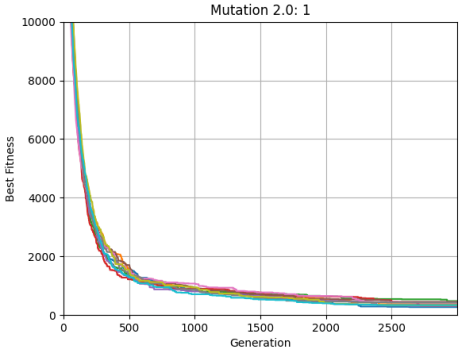

Figure 6: Mean result: 354.16

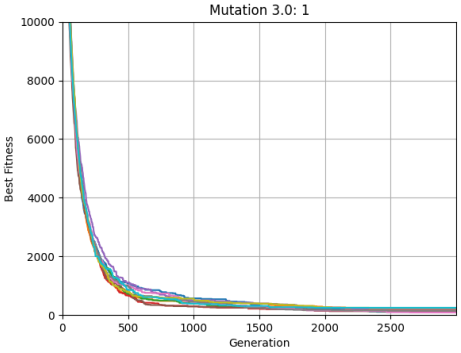

Figure 7: Mean result: 164.52

An important variable tested was the `mutation_rate`, which represents the probability (from 0 to 1) that an agent will mutate. I tested values from 0.1 to 1, in increments of 0.1. The results showed that the best value for this problem is 0.9, meaning 90% of agents mutate. Indeed, the agents generally performed better as the `mutation_rate` increased (Figures 8, 9, and 10). With `mutation_rate` = 1, the mean results are slightly better but the agents converge more slowly, so I decided not to use it (Figure 11).

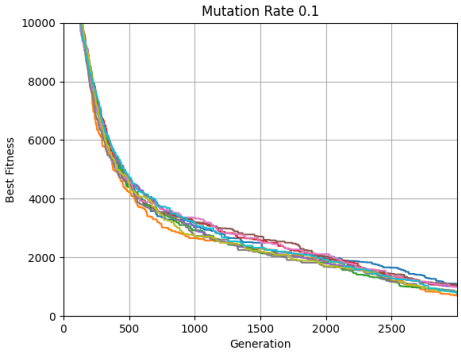

Figure 8: Mean result: 907.45

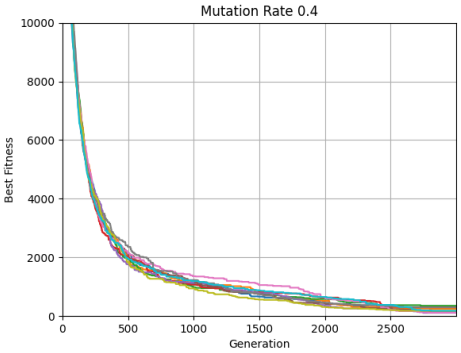

Figure 9: Mean result: 208.63

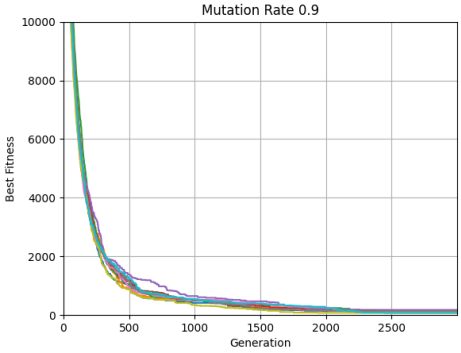

Figure 10: Mean result: 118.4

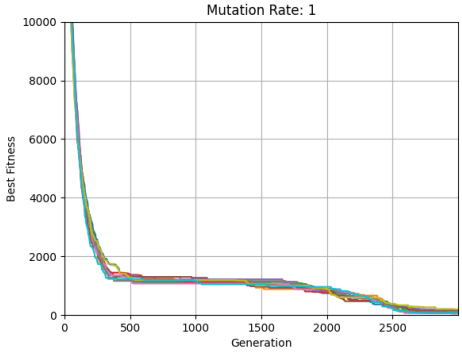

Figure 11: Mean result: 111.37

Finally, I also tested how many mutations to perform per generation, ranging from 1 to 10. The results showed that performing only one mutation per generation gave the best result, while a higher number of mutations performed worse (Figures 12 and 13).

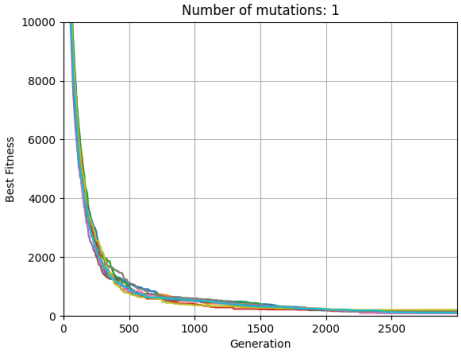

Figure 12: Mean result: 141.97

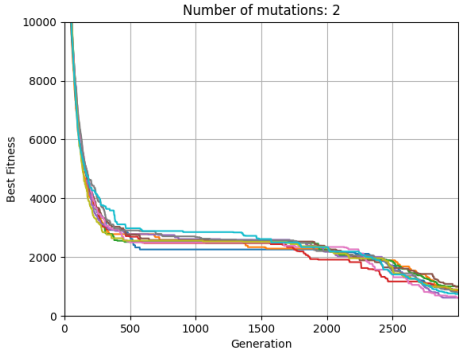

Figure 13: Mean result: 832.49

3.5 Selection

Selection is the process by which the best agents are chosen to produce the next generation. I used a K-way tournament method, where k random agents compete and only the best is selected, and this process is repeated m times.
It was important to do some statistical tests to determine suitable k and m. In my tests, the population size was 100, so I tried different values for k and m from 10 to 100 (in steps of 10) over 3000 generations, keeping $k = m$ for simplicity. Statistical results showed that $k = m = 60$ was the best value. Different results are shown in Figures 14, 15, 16, and 17.

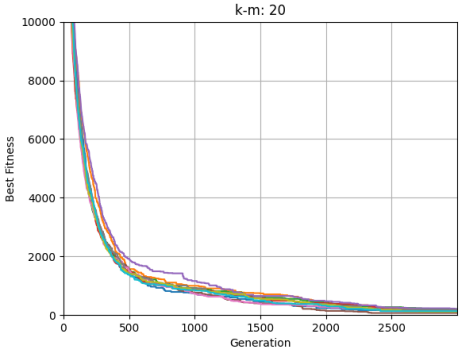

Figure 14: Mean result: 154.29

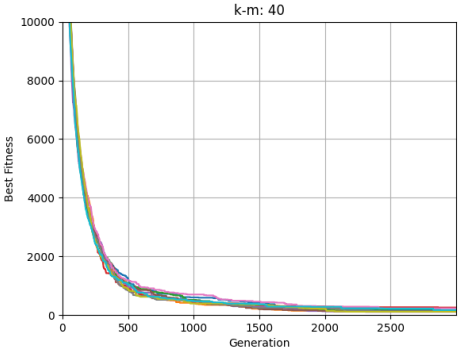

Figure 15: Mean result: 159.92

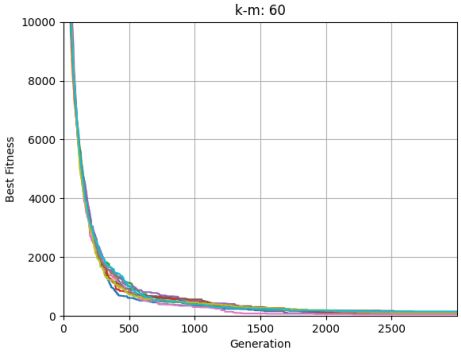

Figure 16: Mean result: 109.72

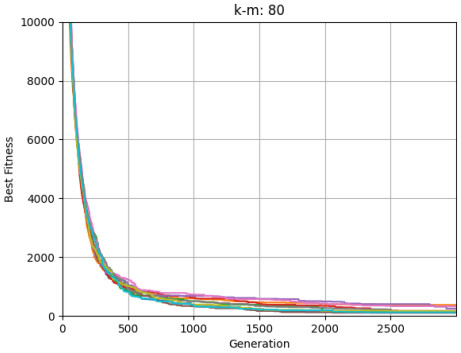

Figure 17: Mean result: 197.59

3.6 Fitness Function

As explained in the introduction, the problem involves several constraints to reach the global optimum. The fitness function is divided into six sub-fitness functions, each evaluating a different aspect of the schedule by counting how many violations the agent has: collisions, professor conflicts, capacity, week distribution, day distribution, and laboratory allocation.
The agent's goal is to minimize the fitness function, ideally reaching 0 (the global optimum).
Note that reaching 0 is almost impossible because the number of students in each course would have to perfectly match the seats in all classrooms. For example, a class of 99 students in a 100-seat classroom results in an error of 0.1, so we can consider the capacity error as the practical minimum.

The six sub-fitness functions are:

- **Collision:** Occurs when two lessons are scheduled in the same classroom, on the same day, and at the same hour. The number of collisions equals the number of overlapping lessons, minus one.
- **Professor Conflict:** Occurs when a professor has two simultaneous lessons.

- **Capacity:** Checks whether a classroom has enough seats for a given student class. If not, the error is three times the difference between the number of students and the room capacity. It also assigns a small error (1/100) for every empty seat, so the agent is discouraged from placing very small classes in very large classrooms.
- **Week Distribution:** Checks how lessons of a course are distributed throughout the week. Each course has a certain number of theory and / or laboratory hours. A course cannot exceed 2 theory hours or 3 laboratory hours per day, and it cannot have both theory and lab sessions on the same day.
- **Distribution in Day:** Checks whether lessons of the same course are consecutive and in the same classroom on that day (e.g., no gaps or room changes for a continuous block of hours).
- **Laboratory Allocation:** Checks whether theory lessons are allocated in laboratories, whether laboratory lessons are allocated in normal classrooms, or whether the wrong laboratory is assigned (e.g., a Biology class placed in a Physics lab).

To combine these sub-fitness values, weights are assigned to each type of violation and summed to yield a single overall score. These weights reflect the relative severity of each error. For example, conflicts and collisions are more severe than a small capacity overrun, whereas day-distribution or week-distribution issues are somewhat less critical than collisions but more severe than minor capacity issues. The weight of lab allocation is somewhat higher (71) to help resolve tricky situations where multiple lab lessons might otherwise be assigned incorrectly in the same day.
The weights used are as follows:

- Collision: 60
- Conflicts: 60
- Capacity: 1
- Week Distribution: 30
- Distribution in Day: 10
- Laboratory Allocation: 71

3.7 Other Improvements

3.7.1 Elitism

While implementing and testing the GA, I tried to add an improvement to the model through elitism. I noticed that the agents' fitness sometimes fluctuated during the final generations, so I hypothesized that keeping the best agent unchanged would eliminate this issue.
Surprisingly, adding elitism did not yield any improvement. In fact, the best overall result was obtained without elitism, as shown by Figures 18, 19, 20, and 21.

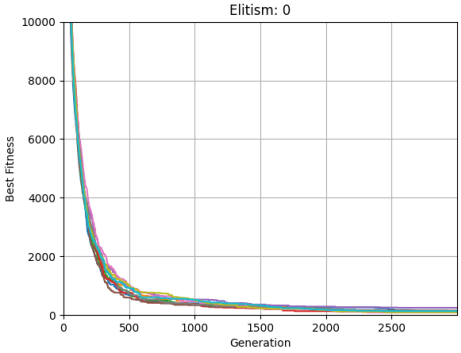

Figure 18: Mean result: 132.22

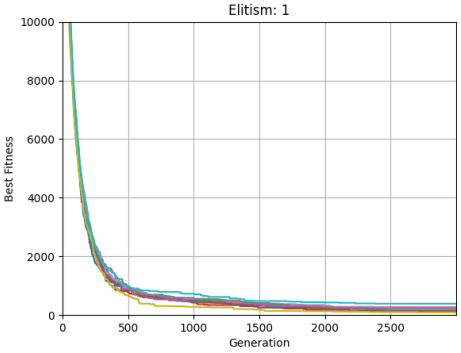

Figure 19: Mean result: 185.7

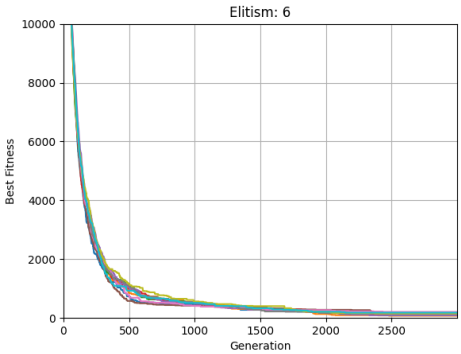

Figure 20: Mean result: 137.64

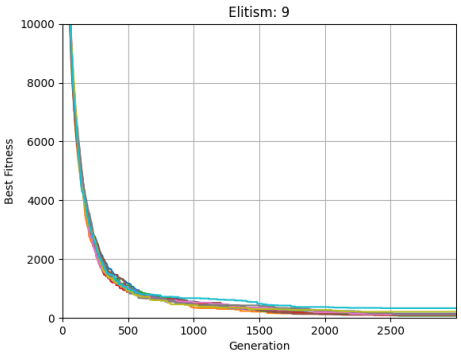

Figure 21: Mean result: 147.14

4 Conclusions

Schempus obtains excellent results on the majority of input datasets, and those that are not nearly perfect exhibit minor issues that a human can solve in seconds. In conclusion, Schempus is a great tool that makes scheduling a timetable an easy and fast process and, with minimal human oversight, produces near-perfect timetables in minutes.

GitHub link: https://github.com/Giuseppe0075/Schempus/