Project Report

Timetable Scheduling

I21-0483 Moeed Asif

Problem

Timetable scheduling problem is a classical problem in the university environment where a

timetable is to be made individually for each semester ensuring minimum number of clashes between sections, professors, and rooms. Thus, we have a timetabling problem where time slots are assigned to each section in a particular room to be taught by a particular professor to teach a particular course. Please note that each week, there are 2 classes per theory course (each of 1 hour 20 mins length), and lab courses will have a single long session of 3 hours. The following constraints need to be followed:

Solution

Overview of the Code

The code defines a class **Schedule** which encapsulates all the properties of a university class schedule, including course details, whether it's a lab or theory session, section information, and the logistics of each lecture like day, timeslot, and room. It includes methods for initializing populations, generating random schedules, calculating fitness, selecting parents, performing crossover and mutation, and evolving the population through generations.

Key Components

1. **Schedule Class**:
    - Stores all relevant attributes for a course's schedule.
    - Methods for generating random schedules and calculating fitness based on room and professor conflicts.
2. **Genetic Algorithm Implementation**:
    - Population initialization with random schedules.
    - Fitness calculation negatively scores schedules based on conflicts, aiming to minimize these.

    - Selection method based on fitness proportionate selection (roulette wheel selection).
    - Crossover and mutation operations that introduce variability and new genetic combinations into the population.

3. **Visualization and Reporting**:
    - Final output includes a visual representation of the best timetable using **matplotlib** and prints the timetable details to the console.

Functionality

- **Initialization**: Starts by creating a randomly generated population of timetables.
- **Evolution Process**: Through specified generations, it evolves the population using genetic operators. It continuously assesses fitness, selecting the best candidates for reproduction and applying crossover and mutation to produce new generations.
- **Fitness Calculation**: Checks for two main types of conflicts: room availability and capacity issues, and professor scheduling conflicts. A lower number of conflicts results in a higher fitness score.
- **Output**: The best schedule (least conflicts) is displayed in a table format and detailed printouts.

Potential Issues and Improvements

1. **Fitness Function**:
    - The fitness function only considers conflict minimization. It could be enhanced by including additional factors like optimal room utilization, student and professor preferences, or minimizing idle timeslots.
    - Debugging statements in **calculate_fitness** might slow down execution and could be toggled with a verbosity flag.
2. **Genetic Operators**:
    - The crossover function splits parent timetables at a random point, which could disrupt important combinations that lead to high fitness. A more sophisticated crossover technique, perhaps one that is aware of the daily structure of schedules, might preserve and mix these traits more effectively.

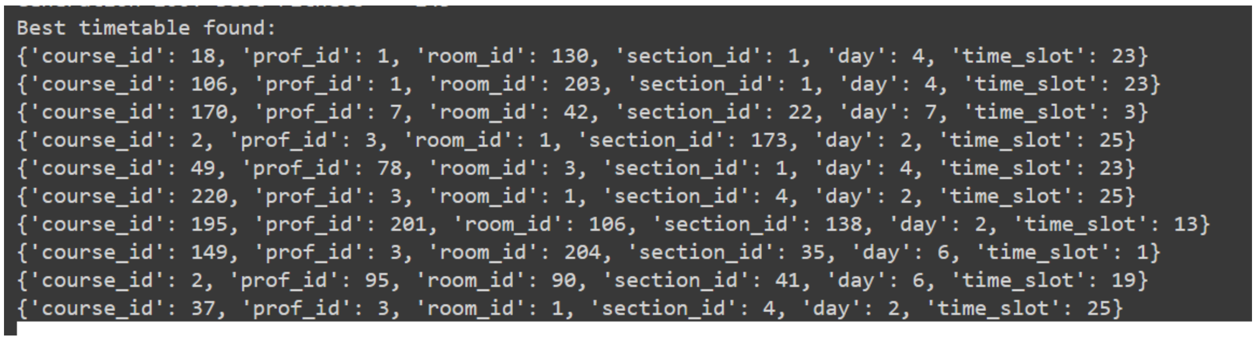
- Mutation is simple and could potentially disrupt schedules significantly. Implementing less invasive mutations or mutations that target specific aspects of a schedule might improve performance.

3. **Scalability and Performance**:
    - The current setup with extensive printing and plotting within the main evolutionary loop can be optimized for better performance, especially for larger populations or more generations.
    - Consider implementing parallel processing for fitness calculation or using a more efficient data structure for managing schedules and conflicts.
4. **Robustness**:
    - Additional validation checks for input values and parameter configurations can make the system more robust and user-friendly.
    - Introducing mechanisms to escape local optima, like simulated annealing techniques or incorporating elitism, where the best individuals are carried over to the next generation unchanged.

Summary

The code provides a functional genetic algorithm for solving the complex problem of university timetable scheduling. While it demonstrates the basic principles of genetic algorithms effectively, there is room for enhancement in efficiency, robustness, and the sophistication of its genetic operators and fitness function to better handle real-world complexities of timetable scheduling.

Output of The Code:



```
Best timetable found:
{'course_id': 18, 'prof_id': 1, 'room_id': 130, 'section_id': 1, 'day': 4, 'time_slot': 23}
{'course_id': 106, 'prof_id': 1, 'room_id': 203, 'section_id': 1, 'day': 4, 'time_slot': 23}
{'course_id': 170, 'prof_id': 7, 'room_id': 42, 'section_id': 22, 'day': 7, 'time_slot': 3}
{'course_id': 2, 'prof_id': 3, 'room_id': 1, 'section_id': 173, 'day': 2, 'time_slot': 25}
{'course_id': 49, 'prof_id': 78, 'room_id': 3, 'section_id': 1, 'day': 4, 'time_slot': 23}
{'course_id': 220, 'prof_id': 3, 'room_id': 1, 'section_id': 4, 'day': 2, 'time_slot': 25}
{'course_id': 195, 'prof_id': 201, 'room_id': 106, 'section_id': 138, 'day': 2, 'time_slot': 13}
{'course_id': 149, 'prof_id': 3, 'room_id': 204, 'section_id': 35, 'day': 6, 'time_slot': 1}
{'course_id': 2, 'prof_id': 95, 'room_id': 90, 'section_id': 41, 'day': 6, 'time_slot': 19}
{'course_id': 37, 'prof_id': 3, 'room_id': 1, 'section_id': 4, 'day': 2, 'time_slot': 25}
```