Artificial Intelligence

Our project involves the development of a genetic algorithm, which is used to find the optimal timetable for a high school. It takes as input two JSON files containing the lessons' data (how many hours each lesson should appear) and the teacher data (how many hours each teacher is available to teach and which subjects they can teach).

How to use

Inside **chromosome.h**, you can freely define the maximum number of hours, days, and the number of classes per grade. These are shared across all grade levels. At the beginning of main.cpp, you define the **POPULATION_SIZE** and the allowed number of generations (**MAX_GEN**) that the algorithm will run. You can also set the paths for the input data files using the variables **LESSON_DATA_PATH** and **TEACHER_DATA_PATH**. Once the algorithm finishes, either because it found the perfect schedule or because the **MAX_GEN** limit was reached, it generates a JSON file containing the school's timetable.

Capabilities and Architecture

Our algorithm calculates the **score** (also known as *fitness*) using 8 functions, each of which checks a different constraint:

1. Each subject appears the required number of times in every grade and every class
calculateSatisfyLessonHoursScore(chrom, lessons);

2. Each teacher works within their available daily and weekly hours
calculateDailyWeeklyLimitScore(chrom, teachers);

3. No teacher is assigned to more than one class at the same time
calculateTeacherConflictScore(chrom);

4. There are no gaps between lessons in a single school day
calculateNoFreePeriodsScore(chrom);

5. Each teacher can work up to 2 consecutive hours
calculateConsecutiveHoursScore(chrom);

6. The hours of each class are evenly distributed throughout the week
calculateAverageUniformityScore(chrom);

7. Each lesson is evenly spread across the week
calculateAllLessonHourSpreadScore(chrom, lessons);

8. Teachers work similar hours across the week (with 30% flexibility)
teachSimilarHoursPerWeek(chrom, teachers);

Each of these methods returns a fraction indicating how close it is to an acceptable solution. Each result is multiplied by 1000 to produce a reasonable range of integer scores.

Each chromosome consists of **nGrades * nClassesPerGrade * nDaysPerWeek * nHoursPerDay** cells, where each cell corresponds to one time slot for one class and contains a pair: a subject and a teacher. From a random number, a *lesson* **ID** is selected, and from the selected *lesson*, a *teacher* **ID** is chosen among the available *teachers*.

For **crossover**, two chromosomes are randomly selected—the higher their score, the more likely they are to be chosen. They are split into two random parts, and two new chromosomes are created, each consisting of a mix from both parents.

For **mutation**, each gene has a probability equal to **MUTATION_PROB**, of being changed by selecting a random subject and then a random available teacher for that subject.

We use 64-bit random number generation to improve randomness in both data selection and algorithm behavior.

Implementation methods

We experimented with various methods for adjusting the mutation rate and the scoring function:

Mutation Rate

- Constant rate
- Stepwise reduction
- Linear reduction
- Constant rate applied only to chromosomes with below-average fitness
- Fitness-based mutation: increase mutation rate when the average fitness does not improve for several generations

Scoring

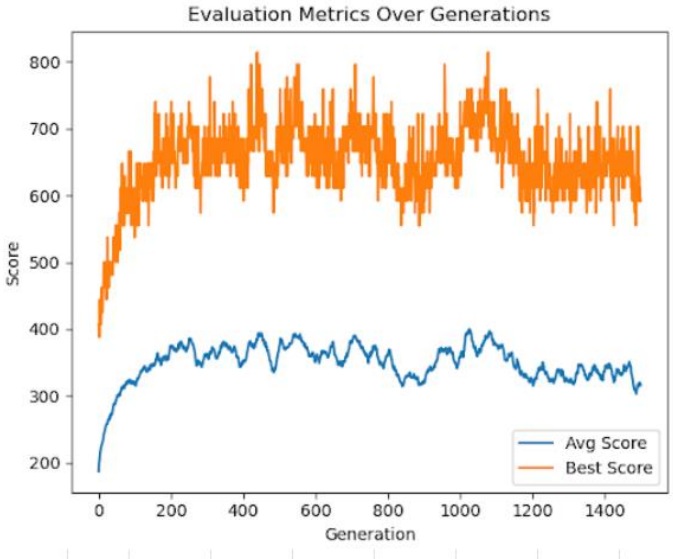
- Linear
- Exponential (e.g., power of 2, 3, 4, 6)

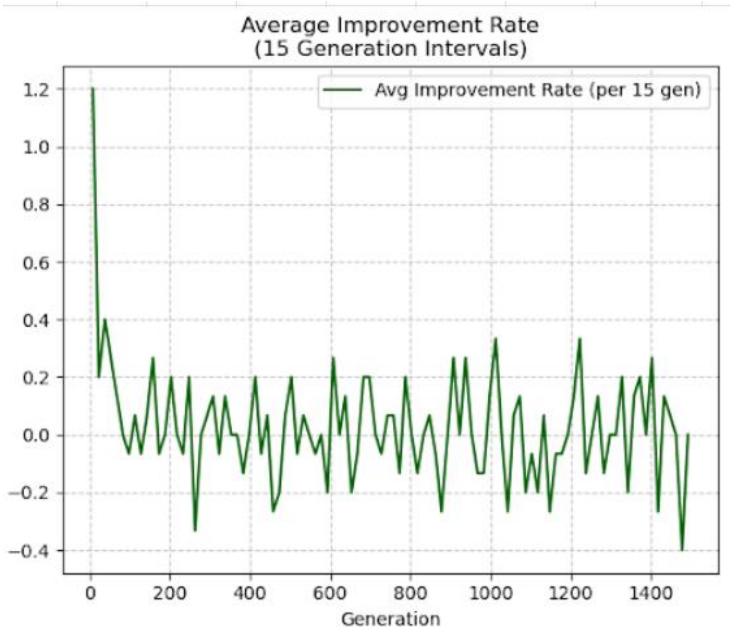
Experiments

A) Experiments using a singular fitness function: satisfy Lesson Hours

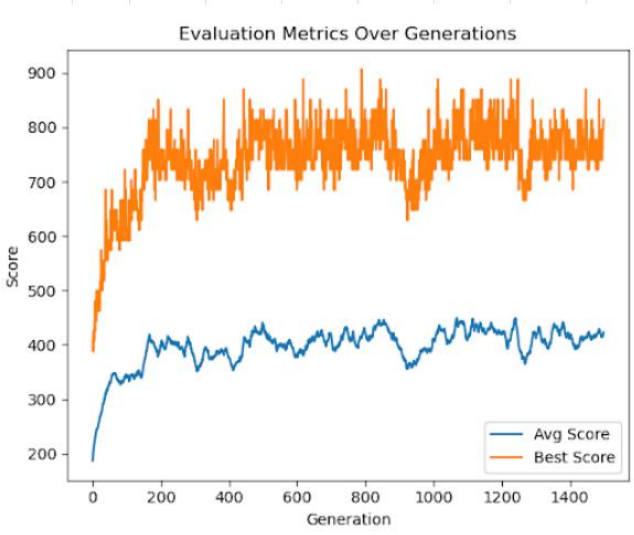
** using simpleLessons.json and simpleTeachers.json dataset*

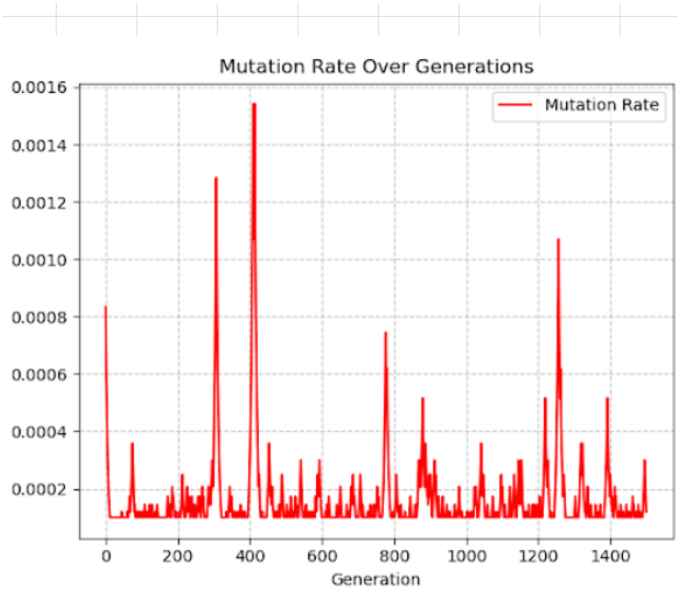
A1. Constant Mutation Rate
Best Fitness Achieved: 81.4%
Population: 2.000
Generations: 1.500
Constant Mutation Rate: 0.001
Seed: 2358873219

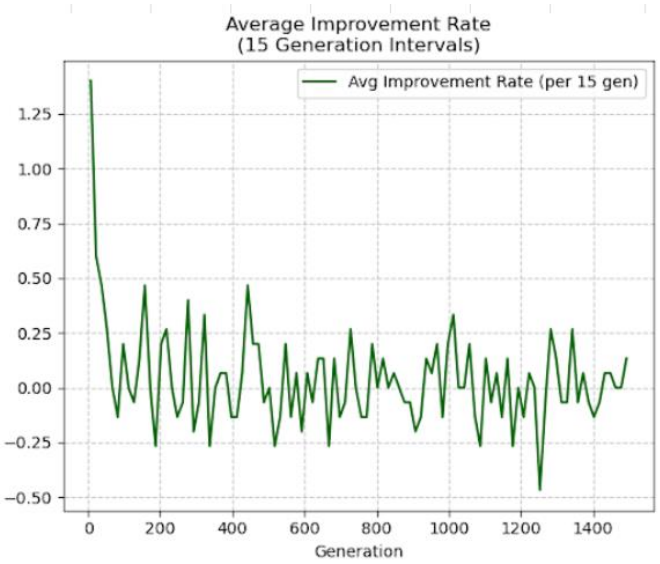




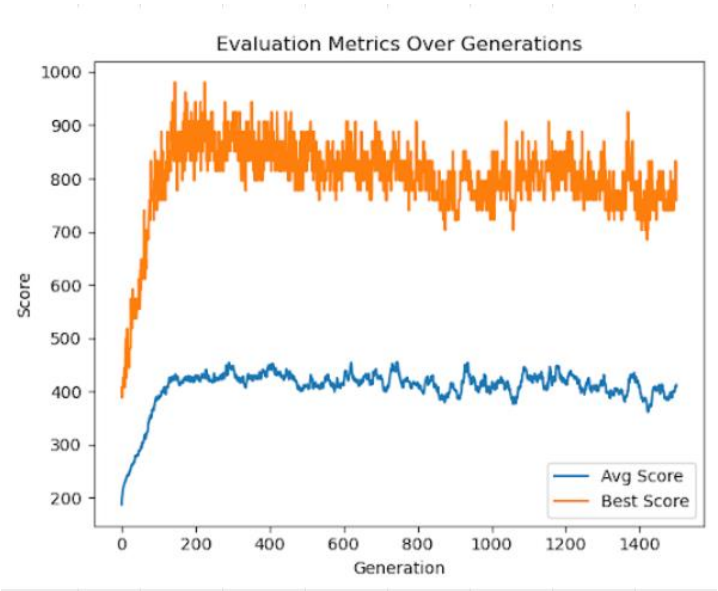
A2 Variable Mutation Rate based on Fitness improvements
Best Fitness Achieved: 90.7%
Population: 2.000
Generations: 1.500
Starting Mutation Rate: 0.001
Mutation Rate Range: [0.001, 0.1]
Adjustment Factor: 1.2
Improvement Threshold: 0.2
Generation Threshold: 10
Seed: 2358873219

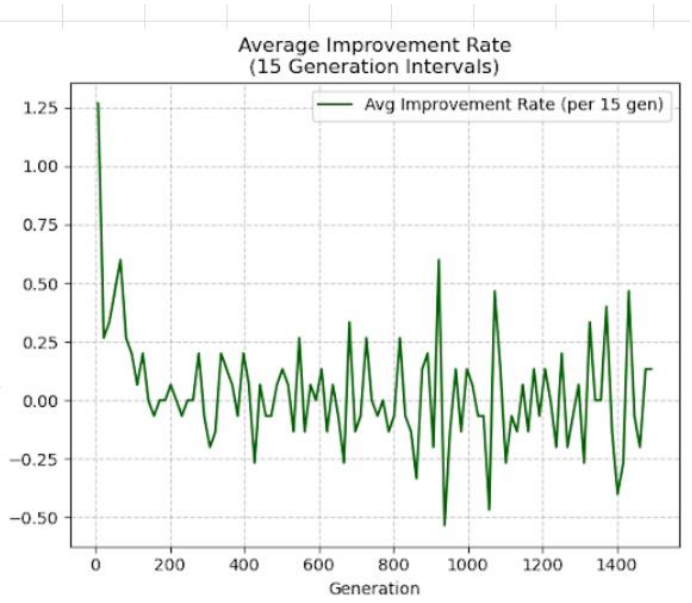




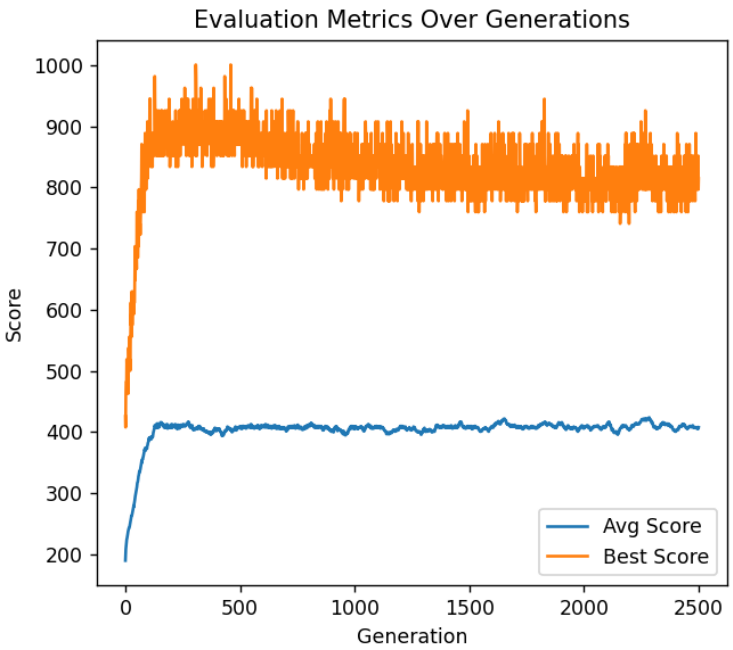


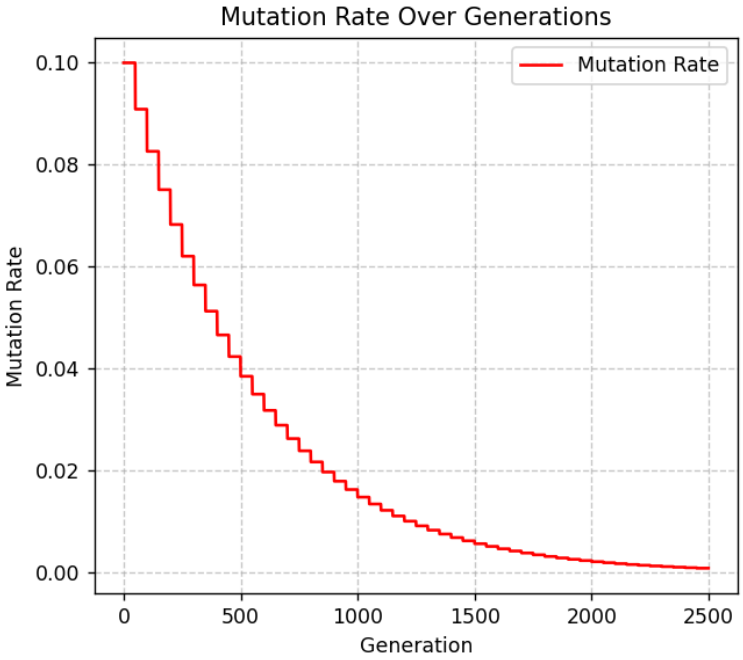
A3. Constant Mutation Rate, applied only to chromosomes with below average fitness.
Best Fitness Achieved: 98.1%
Population: 2.000
Generations: 1.500
Constant Mutation Rate: 0.1
Seed: 2358873219





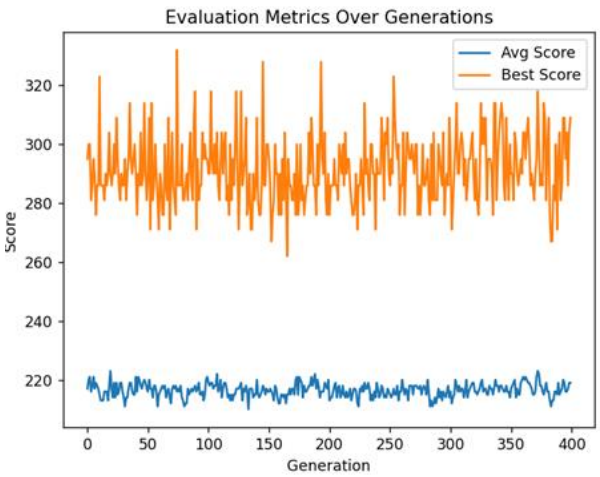
A4. Variable mutation rate every 50 generations
Best Fitness Achieved: 100%
Population: 20.000
Generations: 2.500
Starting Mutation Rate: 0.2
Mutation Decrease: 20%
Seed: 1426592087



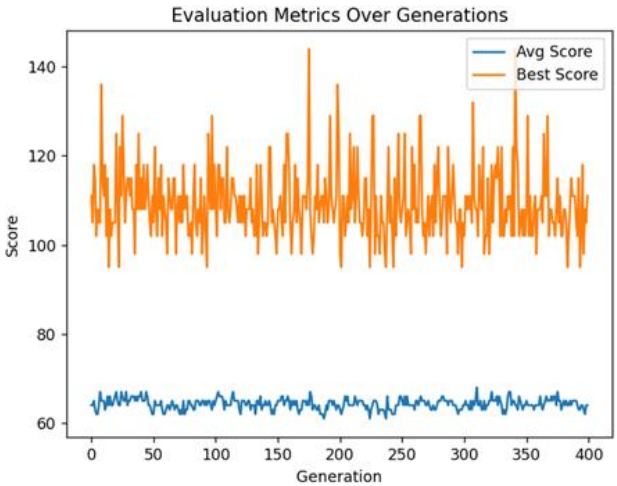


A5. Comparing constant mutation with Linear, Squared and Cubic score
** using lessons.json and teachers.json dataset*
Population: 200
Generations:400
Mutation Rate: 0.2
Seed: 1
The higher the exponent, the stronger the increase; however, beyond the 5th power, we observed only minor gains in relative to the computational cost.

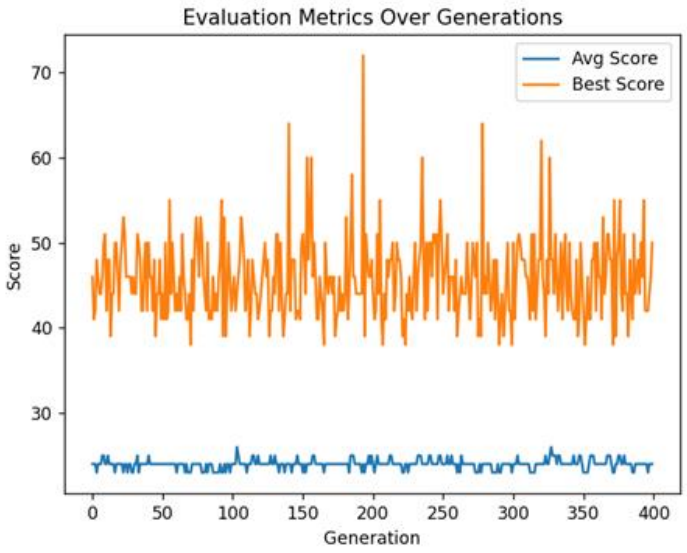
40% for *Linear* (100% is 800)

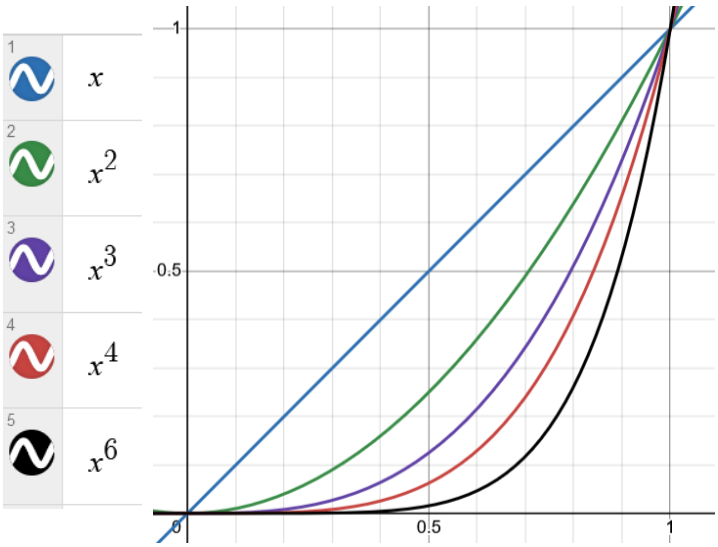


42% for S*quared*



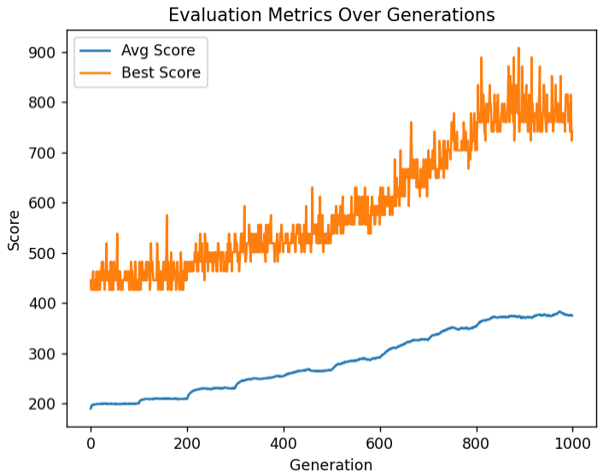
45% for *Cubic*





* Since the results of the scoring functions range within [0,1], increasing the exponent penalizes lower values more heavily.

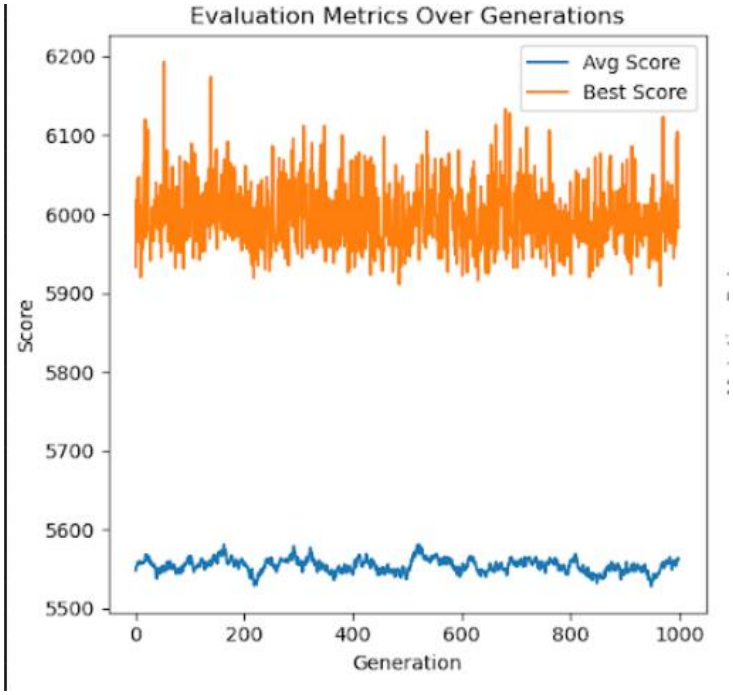
A6. Variable mutation rate with step
** using lessons.json and teachers.json dataset*
Population: 25000
Generations: 1000
Starting Mutation Rate: 0.1
Seed: 413143562

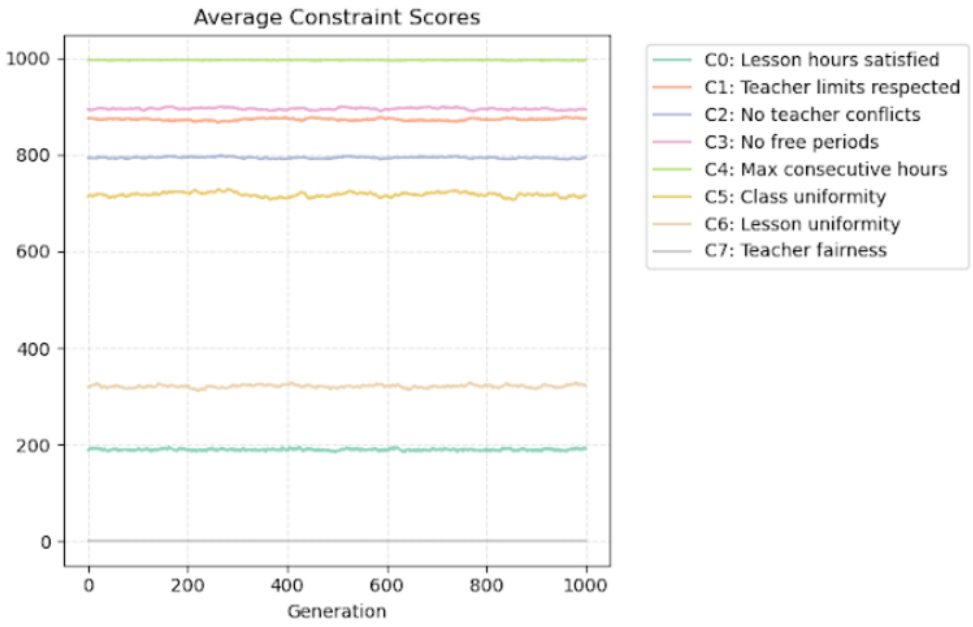



```
case 100:
    MUTATION_PROB = 0.05L;
    break;
case 200:
    MUTATION_PROB = 0.02L;
    break;
case 300:
    MUTATION_PROB = 0.01L;
    break;
case 400:
    MUTATION_PROB = 0.007L;
    break;
case 500:
    MUTATION_PROB = 0.004L;
    break;
case 600:
    MUTATION_PROB = 0.002L;
    break;
case 700:
    MUTATION_PROB = 0.001L;
    break;
case 800:
    MUTATION_PROB = 0.0005L;
    break;
```

B. Experiments using every fitness function
** using simpleTeachers.json and simpleLessons.json dataset*

B1. Constant mutation
Best Fitness Achieved: 77.5%
Population: 2.000
Generations: 1.000
Mutation Rate: 0.05





B2. Variable mutation every 50 generations with Linear scoring
Best Fitness Achieved: 70.4%
Population: 20.000
Generations: 5.000
Starting Mutation Rate: 0.1
Minimum Mutation Rate: 0.0001
ADJUSTMENT_FACTOR: 1.2