Hacettepe University Scheduling

Twilight Group

Timetabling Problem

Utilizing Genetic Algorithms to optimize university schedules

https://github.com/Sebastiao199/TwilightGroup

Authors:

André Filipe Silva	20230972
Guilherme Sá	20230520
Ilyass Jannah	20230598
Sebastião Oliveira	20220558
Sebastião Rosalino	20230372

NOVA IMS

Academic Orientation: Leonardo Vanneschi and Berfin Sakallioglu

2023/2024

Contents

Division of Labor

André Silva - charles.py, selection_algorithms.py, Problem Introduction, Selection Methods, Conclusion

Guilherme Sá - utils.py, Problem Introduction

Ilyass Jannah - fitness.py, data.py, Fitness Sharing, Appendix

Sebastião Oliveira - crossovers.py, mutations.py, Crossover Operators, Mutation Operators

Sebastião Rosalino - optimization_problem.py, experiments.py, Results and Discussion, Conclusion

---

1 Problem Introduction

This project aims to optimize the schedules of Hacettepe University, ensuring that each Practical Turn (from now on simply referred to as Turn) has the best possible schedule. We utilize Genetic Algorithms to generate to most optimal schedules possible, and we discuss our approach and results throughout our report.

Before we start, we realize our nomenclature can be confusing. As such, we created a Glossary with longer explanations for each term used. It can be found on [Appendix A - Glossary].

For the **Individual's Representation**, we have a proposal of weekly schedule for all Turns attending the university. The representation is a three-dimensional array, showing which (if any) subject each Turn is attending on a specific block. Each inner list represents a day of the week for a Turn, and it has 8 values, corresponding to the 8 blocks in the schedule for each day. Each Turn is enrolled in 4 subjects, no more no less (this is a **hard constraint**). It is noteworthy to mention that all students in the same Turn have the same schedule. This was the most intuitive and realistic representation we could find to depict our scheduling problem. You can find a more visual representation on [Appendix B - Individual's Representation visual example].

Regarding our **Fitness Function**, we increase the fitness value when one of our soft constraints is violated. This means that a lower fitness is better, making this a **Minimization Optimization Problem**. Since we define as the weekly minimum blocks per subject to be 8, we penalize the fitness if a Turn has less than 8 blocks of each assigned subject. We also penalize overlaps as we assume that each subject can only be taught by one professor. If, for example, Turn 1 and Turn 3 both have Maths scheduled on the first block of Mondays, one of the Turns will not have the subject due to lack of teaching staff. To ensure students have adequate leisure time, or at least a lunch break, a penalty will be imposed on individuals that do not include at least one break block per day. Furthermore, we want our students to have lunch at proper lunch time. Note, however, that this is only a **soft constraint** - meaning that it is possible that our best individual does not have any breaks during the day. Faculty is demanding! Finally, when individuals do have a break, we want them to have it in what we call "middle blocks" (blocks in the 3rd or 4th index position, out of 8 possible) - you can consider it as proper lunch time. According to their importance, different penalty points were assigned to each criteria. Here follows the hierarchy of penalties for every criteria:

Penalty name	**Penalty definition**
Minimum Blocks per Subject = 8	+= max(0,[8 - Actual Nr Blocks Per Subject) * 5])
Overlap	+= 3
Having no 'Break' in a day	+= 4
Break not in "middle blocks"	+= 2

Table 1: Fitness penalties

Our fitness function started in a more simple fashion, with less penalties. As we were experimenting, we noticed that we were converging to a global optimum rather quickly. As a result, we added more and more layers of soft constraints, until we arrived at the final fitness function.

Our **Data** was synthetically generated using an algorithm we created ourselves (found in 'data.py'). We created 10 Turns, with 4 subjects each, 5 days of classes per week, and 8 blocks of classes per days. The University offers 30 unique subjects.

Our fitness can only be 0 (that would be a **Global Optimum**) or a positive integer.

2 Fitness Sharing

In our case, we did not experience premature convergence (the primary issue addressed by fitness sharing). However, we decided to apply fitness sharing to our Genetic Algorithm to see if it could enhance our results.

The implementation for our timetable problem was as follows:

1. Calculate the initial fitness values for all individuals in the population.
2. Compute the Hamming distance between all pairs of individuals (weekly schedules for all turns). This distance is the number of differing blocks between two individuals. The function iterates through turns, days, and blocks, incrementing a counter when blocks differ. The final counter value represents the Hamming distance. We then calculate the Hamming distance matrix for the population, which returns a list of lists, where each sublist contains the Hamming distances between one individual and the others.
3. Calculate the total length of the individuals, defined as the total number of blocks (5 days * 8 blocks per day = 40 blocks).
4. Calculate the inverted normalized distance, which is $1 - \frac{\text{Hamming distance}}{\text{length of individuals}}$.
5. Compute the new fitness scores using fitness sharing. Since this is a minimization problem, rare individuals will have their fitness values decreased, and common individuals will have their fitness values increased. For each individual, sum the inverted normalized distances and update the fitness score as follows:

$$\text{New Fitness Value} = \text{Old Fitness Value} \times (\text{Sum of inverted distances for the individual})$$

This process aims to balance fitness scores, thereby improving the algorithm's exploration capabilities by enhancing diversity in the population.

3 Selection Methods

For the selection phase of the Genetic Algorithm, we tested three different selection methods: Fitness Proportional Selection, Ranking Selection, and Tournament Selection.

Starting with **Fitness Proportional Selection**, as this is a minimization problem, individuals with a lower fitness value must have a higher probability of being selected. Thus, we take the inverse of the fitness ($\frac{1}{fitness}$), in this way attributing a higher probability of being selected to the lowest fitness individuals.

Continuing to **Ranking Selection**, this method simply loops through each individual's fitness value and ranks them, top to bottom, from highest to lowest fitness. A probability of selection is given to each of the ranked individuals: the higher the rank, the higher the probability of being selected.

Finally, with **Tournament Selection**, a pool of 3 random individuals is sampled, with repetition, from the population for each tournament, and in each tournament, the best individual (lowest fitness value) is selected as the winner. A tournament size of 3 provides moderate selection pressure, which helps to avoid premature convergence while still effectively driving the population towards better solutions. This pressure ensures that stronger individuals are more likely to be selected, promoting convergence, but also allowing for a sufficient chance of selecting weaker individuals, maintaining genetic diversity and preventing the algorithm from getting stuck in local optima.

It is worth noting that the **Ranking Selection** method seems particularly useful in this optimization problem: by using ranks rather than raw fitness values, it mitigates the effects of outliers, helping to

maintain diversity within the population by giving 'worse' individuals a better chance of being selected. This stochastic method also helps avoid premature convergence on sub-optimal solutions and ensures a broader search across generations.

Exploring different methods is important for achieving a balance between population diversity and selection pressure.

4 Crossover Operators

We implemented 4 different crossover methods based on two different crossover architectures - **Single Point Crossover and Uniform Crossover**. Each one of these architectures has two different implementations due to our Individual's Representation being 3-dimensional.

Going into detail, we will start by the **Single Point Day Crossover**. In this method, we split the parents at the day level (e.g. we split on Tuesday), and the days are swapped between parents at that point to generate the offspring.

The **Single Point Block Crossover** follows the same logic, but inside each day: for each day of the week, a random crossover point is selected and the split is done at that point to generate the offspring. It is important to note that the crossover point does not have to be the same for each day of the week - we implemented this additional randomness to add diversity to the offspring.

Regarding the **Uniform Crossover**, in essence, either offspring has the same chance to inherit a specific allele from the parent. We apply this in two ways, which are exactly as for the Single Point Crossover. We call these methods **Uniform Day Crossover**, where the swapping is done at the day level and **Uniform Block Crossover**, where the swapping is done at the block level (i.e. inside each day).

Lastly, we implemented checks in our code to guarantee that the crossovers were possible - we made sure the parents had the same number of school days, and made sure that in each day they had the same number of subjects. If these criteria were not met, the crossover would not happen.

5 Mutation Operators

We implement 3 different mutation methods: **Swap Mutation**, **Inversion Mutation**, and **Scramble Mutation**. All three of our methods work at the block level, so they are intra-day mutations, but we apply them for all days of the week.

The implementations of Swap Mutation and Inversion Mutation closely resemble the class approach, only slightly modified for our problem.

The **Scramble Mutation** however was implemented from the ground up. This mutation randomly scrambles the order of blocks within each day for all Turns. As such, the mutated individuals will have a totally shuffled schedule.

For examples of our crossovers and mutations please check [Appendix C - Crossovers and Mutations Examples].

6 Results and Discussion

We studied all possible combination of parameters (Selection Algorithms, Crossovers and Mutations) to find out the best solution to our problem. With 3 selection methods, 4 crossover operators, 3 mutation

operators, and with or without fitness sharing, that gives us a total of 72 possible combinations. We always keep elitism on (N=1). The choice for N=1 is because, when optimization problems have multiple local optima (which is our case, and we know this because we ran the experiments extensively) it is advisable to use singular elitism in order to avoid staying stuck in said local optima. Since our problem is complex and computationally expensive to run, we decided that it would be prudent to always keep the best individual ever found, so that we do not waste the progress made by our GA in its evolution. We kept the probabilities of crossover and mutation fixed at 90% and 20%, respectively. We let our population evolve for 500 generations. To reach statistical significance, we ran 30 trials of each combination, resulting in a total of 2160 different *experiments*. For the plots, we use the average best fitness across all trials and the 95% Confidence Intervals.

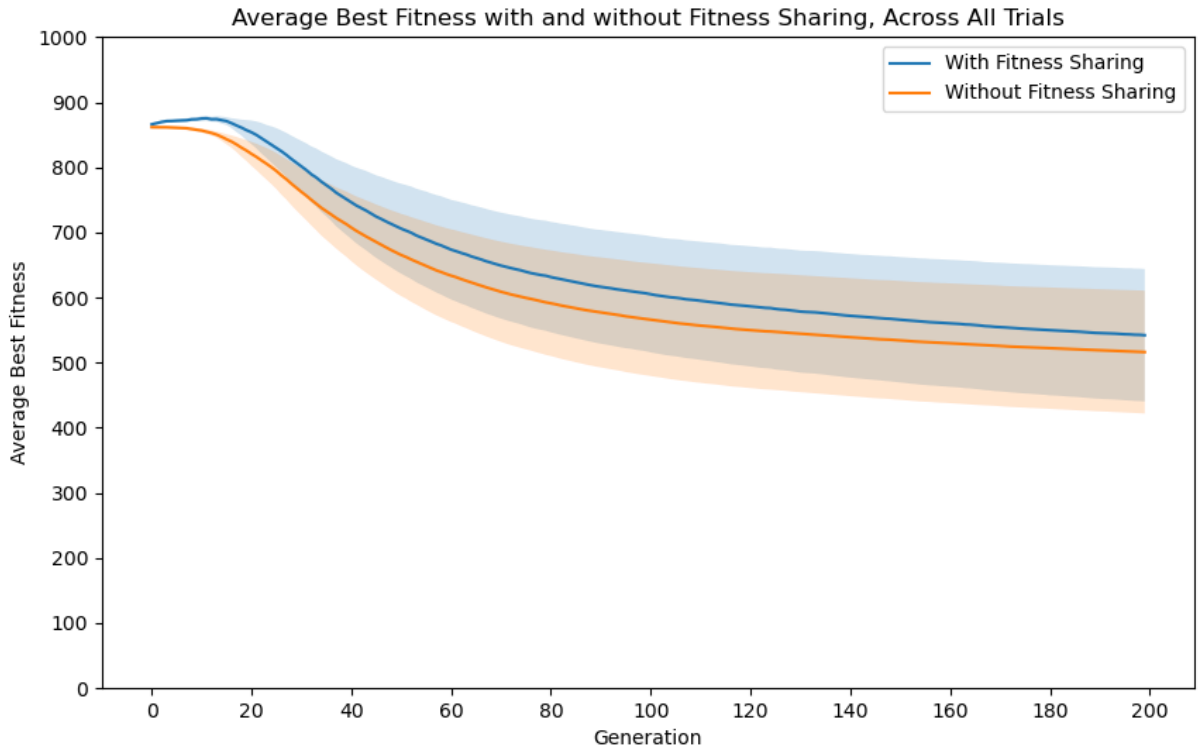

Figure 1: Comparison between Fitness Sharing and No Fitness Sharing

Fitness sharing did not result in any statistically significant differences. The only difference is that computing experiments with it enabled take exponentially more computational resources than without it. As such, when we were confident of its lack of importance, we ran only experiments without fitness sharing.

To determine the best combination, we took the average of the best fitness values for all 500 generations, across the 30 trials ran, and then calculated the minimum value out of those averages. We found that the winning combination was composed by Tournament Selection, Uniform Block Crossover and Block Scramble Mutation. Our best individual had a fitness score of 56, and you can find its representation in an easily readable format on our Github by clicking here.

In this page and the following ones, you can find the plots that show the behavior of the different parameters.

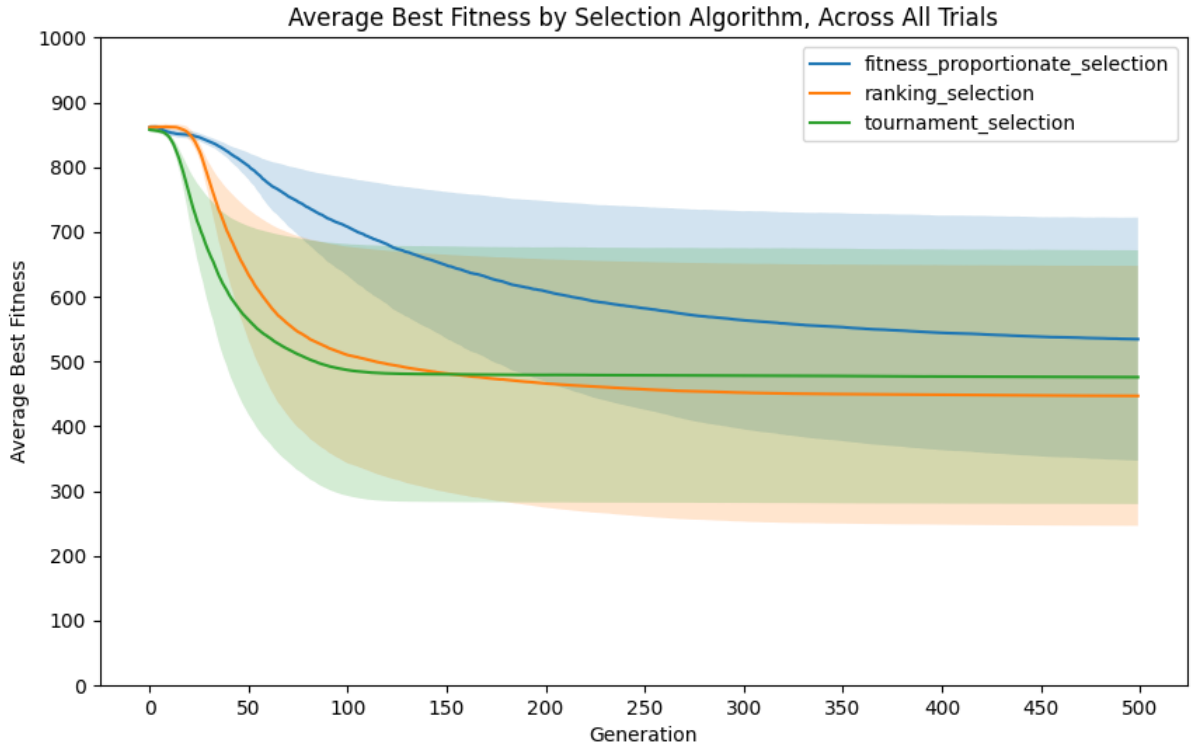

Figure 2: Fitness evolution by selection method

By looking at just the average value (the plotted line), we would conclude that our best Selection Method, overall, is Ranking Selection. Tournament selection comes very close to it, and Fitness Proportionate Selection lags behind those methods a little. However, the 95% confidence intervals overlap between the three methods, which does not let us accurately make these statements, as they show no statistically significant differences between them, except at the very start, when tournament selection is a stronger initializer.

Looking at Figure 3, it is quite clear that the "block" Crossovers work much better than the "day" crossovers. Changing blocks within a day, either by uniform crossover or single point crossover is much better for fitness than applying transformations to days. This makes sense given that each day has 8 blocks and there are only 5 days, so there are more possibilities for different crossovers in blocks than in days.

As for Mutations, in Figure 4, Swap and Scramble are so close together that it is almost indistinguishable which one produces better scores. Inversion does not come far off, however. Once again, this is just looking at the average value. The 95% confidence intervals give us a notion that all three mutation types perform approximately in the same manner.

All in all, we conclude that only different Crossover operators have a significant impact on the convergence of our Genetic Algorithm. Different Mutation operators do not show difference in their convergence pace, and lead to approximately the same results.

Analysing our best individual, some details stand out. First, our algorithm managed to properly make it so that when a 'Break' block was introduced in a day, it was always in the "appropriate" time slots (3rd or 4th index position - Block 4 or Block 5), as we set it to be. However, not all days had at least one 'Break' block. Although this is not optimal in a real-world scenario, as we would want students to always have a lunch break, we did not force it as a hard constraint on our problem's design. By not forcing it as a hard constraint, this result follows logic.

In addition, we also did not force a hard constraint of a maximum number of 'Break' blocks per day. This was intended however, as we did not mind that a Turn had a day without classes, as long as the schedule was appropriate. This stands out immediately on Turn 1's Monday schedule, which is a free day.

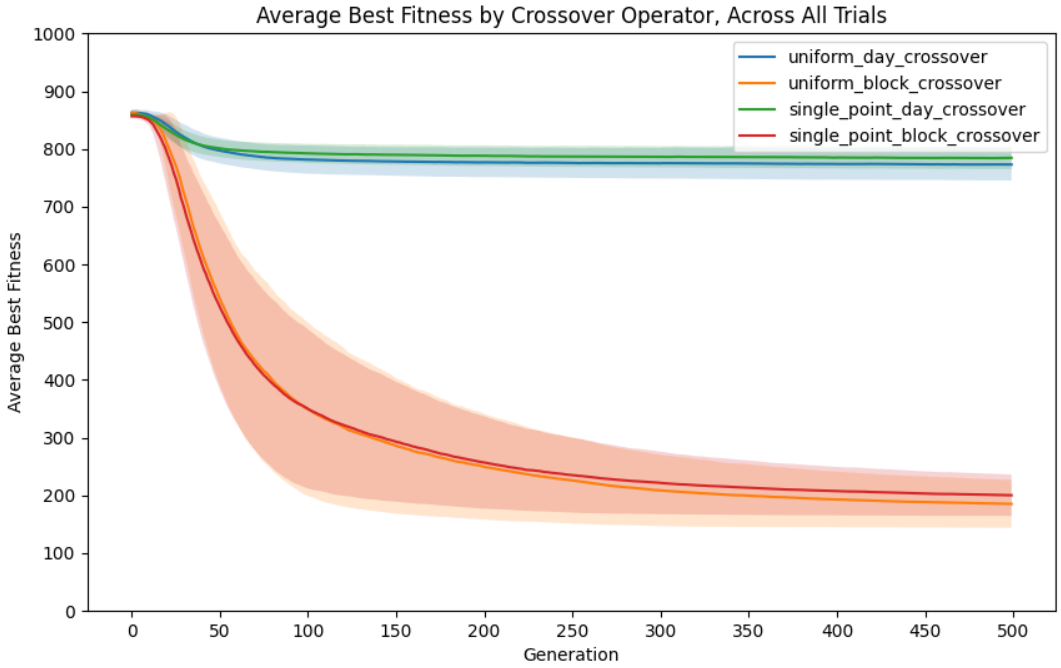

Figure 3: Fitness evolution by crossover method

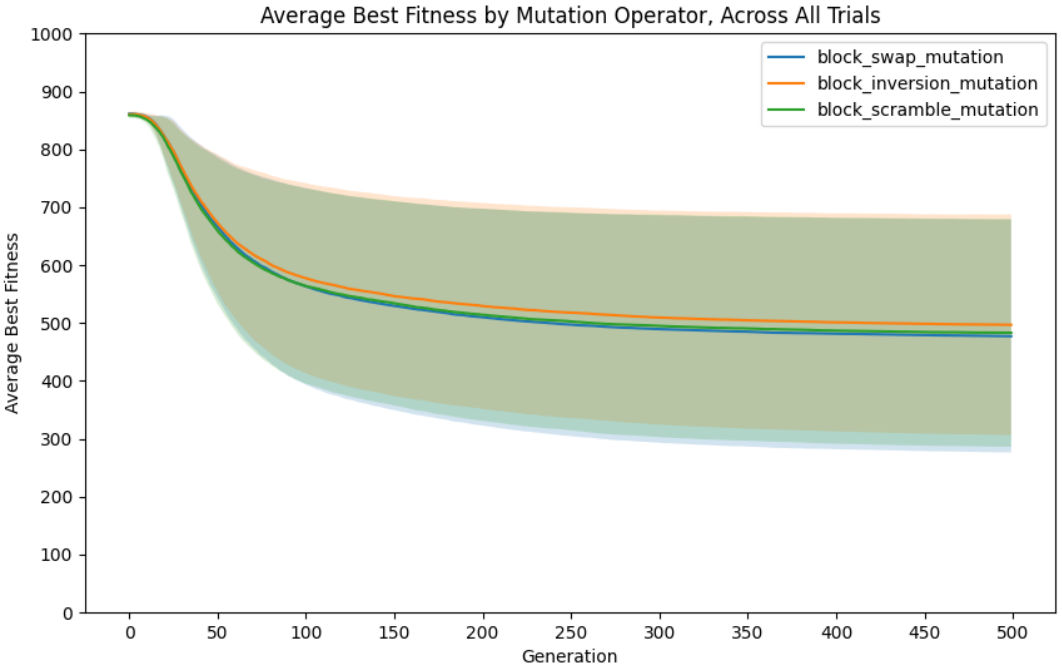

Figure 4: Fitness evolution by mutation method

The algorithm was able to ensure that every single Turn had at least the minimum number of Blocks per Subject (8) defined by us.

There are no overlaps in our best Individual. This means our penalties were designed well enough so as to avoid the algorithm needing to converge to a best individual with overlaps.

Using the best combination found, the algorithm was tested with more generations. Below you can find the results on Figure 5.

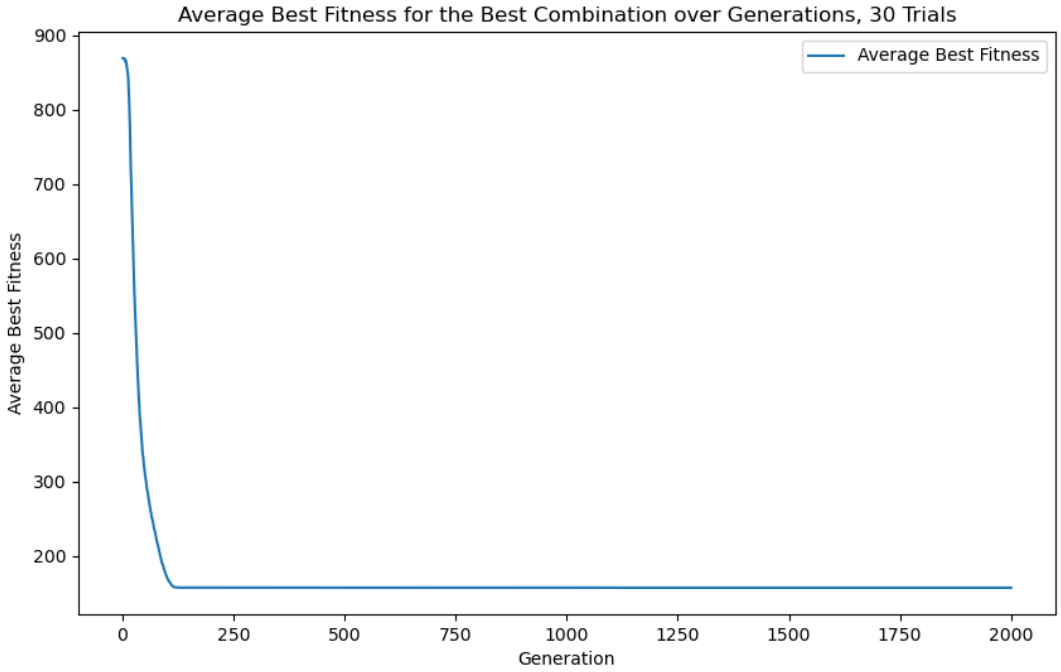

Figure 5: Fitness by Generation of Best Combination Algorithm

7 Conclusion

In our work, we developed a quite extensive search for the best timetabling possible, applying the knowledge of Genetic Algorithms learned in the course. The champion approach is the combination of Tournament Selection, Uniform Block Crossover, Block Scramble Mutation, and no fitness sharing.

We had 72 different possible combinations, and we ran each combination for 30 trials for statistical significance, resulting in 2160 runs. This was quite computationally intensive, and limiting in the sense that if we wanted to try new approaches, we would need to run everything from the start, which takes more than 48h to run. Limitations to our work come from this time restriction. We could have tried different tournament sizes, different combinations of crossover and mutation probabilities, or we could have introduced other soft/hard constraints. In addition, the implementation of partially mapped and/or cycle crossover could further improve our results, but once again would increase the computational cost in such an exponential way that it would be unfeasible for us to test in time.

All factors considered, we believe we achieved good results. But we recognize there is a lot of room to improve and further work that could be done, as mentioned above.

Appendix

Appendix A - Glossary

The following Glossary will not be presented in alphabetical order, but in the best order for all of the terms to be understood.

Subject - A Subject is our designation for different courses: Maths, Programming, Literature, etc. The university offers 30 different subjects.

Block - A "Block" represents a time slot in the schedule. Blocks are indivisible - Maths from 9AM to 10AM would be considered a Block. Turkish Literature from 2PM to 3PM would also be considered a Block. Having no classes in a Block can also happen, that is when a Block is labelled as "Break".

Turn - A Practical Turn / Turn is a class. Meaning, it is a group of people that all attend the same subjects in the exact same schedule. For our purposes, we do not need to go to the individuality of the person, so we just define it at the level of the Turn. A Turn is also indivisible.

Appendix B - Individual's Representation visual example

Individual Example - 3 Turns, 5 Days of school per week, 4 Blocks per day, 2 Subjects per Turn.

```
Turn 1                                                  Turn 2                                                  Turn 3
[[['Subject_2', 'Subject_2', 'Subject_1', 'Break'],     [['Break', 'Subject_2', 'Break', 'Subject_1'],          [['Subject_1', 'Subject_2', 'Break', 'Subject_2'],
  ['Subject_2', 'Subject_2', 'Subject_1', 'Subject_2'],   ['Subject_1', 'Subject_2', 'Subject_1', 'Break'],     ['Subject_1', 'Break', 'Break', 'Subject_1'],
  ['Subject_1', 'Subject_2', 'Subject_2', 'Subject_2'],   ['Break', 'Break', 'Break', 'Subject_2'],           ['Subject_1', 'Subject_1', 'Subject_2', 'Subject_2'],
  ['Subject_2', 'Subject_1', 'Subject_1', 'Break'],       ['Break', 'Break', 'Subject_2', 'Subject_2'],       ['Subject_1', 'Subject_2', 'Subject_1', 'Subject_1'],
  ['Subject_1', 'Break', 'Subject_1', 'Subject_2']],      ['Break', 'Subject_2', 'Subject_2', 'Subject_2']],  ['Subject_2', 'Subject_2', 'Subject_1', 'Subject_2']]]
```

Appendix C - Crossovers and Mutations Examples

We decided to include simple examples for our crossovers and mutations to provide a better understanding of how they work, as they differ from class implementation. All crossovers/mutations come from the same set of parents, so we only define them once. In the mutations, we only show one offspring as all the mutation examples are about one offspring.

Here, we show an example of a Turn with just 2 days of school, and only 3 Blocks of courses per day.

Parent 1

[[['Break', 'Subject_2', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]

Parent 2

[[['Subject_2', 'Break', 'Subject_1'], ['Break', 'Subject_2', 'Subject_2']]]

Uniform Day Crossover Example
Offspring 1
[[['Subject_2', 'Break', 'Subject_1'], ['Break', 'Subject_2', 'Subject_2']]]
Offspring 2
[[['Break', 'Subject_2', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]

Uniform Block Crossover Example
Offspring 1
[[['Subject_2', 'Break', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]
Offspring 2
[[['Break', 'Subject_2', 'Subject_1'], ['Break', 'Subject_2', 'Subject_2']]]

Single Point Day Crossover Example
Offspring 1
[[['Break', 'Subject_2', 'Subject_2'], ['Break', 'Subject_2', 'Subject_2']]]
Offspring 2
[[['Subject_2', 'Break', 'Subject_1'], ['Break', 'Subject_2', 'Break']]]

Single Point Block Crossover Example
Offspring 1
[[['Break', 'Break', 'Subject_1'], ['Break', 'Subject_2', 'Subject_2']]]
Offspring 2
[[['Subject_2', 'Subject_2', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]

Block Swap Mutation Example
Offspring
[[['Break', 'Subject_2', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]
Mutated Offspring
[[['Subject_2', 'Break', 'Subject_2'], ['Subject_2', 'Break', 'Break']]]

Block Inversion Mutation Example
Mutated Offspring
[[['Subject_2', 'Break', 'Subject_2'], ['Subject_2', 'Break', 'Break']]]

Block Scramble Mutation Example
Mutated Offspring
[[['Subject_2', 'Break', 'Subject_2'], ['Break', 'Subject_2', 'Break']]]