Using a Genetic Algorithm to Solve Class Scheduling

Optimization Methods for Engineers - Project Presentation

Thomas Meyer-Lehnert (21-949-292)

30.05.2024

Overview

Problem Description - Motivation

Schools and universities have to create a timetable for their classes.

This timetable has requirements, e.g.:

- It has to include all classes to be taught
- It must not have one professor teaching two classes at the same time
- It should not have overlaps of classes for students
- < … >

-> Find the 'best' timetable according to some evalutation criteria

- Generally NP-hard[1][2][3]
- Simplified version of the 'University Course Timetabling Problem'[4]

[1] Javier Arias-Osori and Andrés Mora-Esquivel

[2] Anmar Abuhamdah and Masri Ayob

[3] NP-hard: In simple terms, the problem is not solvable in polynomial time.

[4] Anmar Abuhamdah and Masri Ayob

Problem Description - Overview

We want find timetables that that assign each timeslot (e.g. Monday 10-12) and room to a class (or no class). We then want to optimize this to find a timetable that is ‘best’, so for example minimize class overlaps for students.

The model will be simplfied: Classes, Rooms, Timeslots are integers, Students and Professors sets of integers (the classes they take/teach).

Each class just has one lesson, each room has infinite capacity.

One could define a lot more constaints, but to showcase the genetic algorithmic approach, this is sufficient. More mandatory constraints and optimization criteria could easily be added.

Problem Description - Formal

- We have a set of Constraints
  $\mathbb{C} = \{C = \text{Classes}, S = \text{Students}, P = \text{Professors}, R = \text{Rooms}, T = \text{Timeslots}\}$
- $C \subset \mathbb{N}, R \subset \mathbb{N}, T \subset \mathbb{N}$
- Each 'student' $s \in S$ & each 'professor' $p \in P$ is a set of classes $C_s \subseteq C$ (resp. C_p)

A Timetable $Q \in \mathbb{Q}$ is a bijective function $T \times R \to C$ that satisfies:

- All classes are held: $\forall c \in C \exists t \in T, r \in R : Q(t, r) = c$
- No professor teaches two classes at the same time:
  $\forall p \in P, t \in T : |\{c \in C_p : Q(t, r) = c\}| \leq 1$

As a bijective function, Q has an inverse $Q^{-1} : C \to T \times R$

We want to find a timetable Q that minimizes some evaluation function $f : \mathbb{Q} \to \mathbb{R}^+$

Problem Description - Evaluation Function

We define the evaluation function $f(Q)$ as follows:

$$f(Q) = \sum_{s \in S}$$

$$(|\{c \in C_s \mid \exists c' \in C_s \text{ such that c' is held at the same time as c}\}| - 1)$$

$$+(|\{t \in T \mid Q(\_, t-1) \in C_s \land Q(\_, t+1) \in C_s \land Q(\_, t) \notin C_s\}|)$$

This function counts the number of overlaps of classes for students and the number of 'gaps' in the timetable. The second criterium was chosen to make the algorithm prefer 'blocks' of classes over fragmented schedules.

A more sophisticated evaluation function could include lunch breaks, prefer morning classes, prefer one full day over many sparsely filled days, etc.

Genetic Algorithm - Motivation

-> Why use a genetic algorithm?

This problem has no obvious way of ‘improving’ upon a previous solution, and it is very difficult to determine when local minima are reached. Thus, an algorithm with a lot of randomness is beneficial, as it can explore the solution space more effectively.

Especially the mixing step of a genetic algorithm is good for this task, as it is a way to combine the good parts of two solutions while leaving out the bad parts.

Other approaches (such as Taboo Search (not explained here)) are possible, however many papers about this problem also use genetic algorithms.

Genetic Algorithm - Overview

In general, a genetic algorithm can be used to optimize a problem in the following way:

1. Represent a possible solution to the Problem as a ‘Genome’
2. Create a population of Genomes
3. Evaluate each Genome using an evaluation function
4. Select the best Genomes from the population
5. Create a new population by ‘breeding’ (combining/crossing) the best genomes and mutate them with a small probability
6. Repeat from step 3 until a stopping criterion is met

Genetic Algorithm - Application

In our case, a genome is a timetable Q. We represent it as a table of size $|T| \times |R|$ with entries in C. The special value of 0 means ‘no class’.

Using the steps outline before, we can iteratively find timetables minimizing the evaluation function f.

We now need to define a combining function (from now on called `cross`) and a mutation function `mutate`.

Genetic Algorithm - Combining Function `cross`

`cross` takes two timetables Q_l, Q_r and creates a new timetable Q^*.

We define it as follows:

1. Begin with a timetable filled with zeros for no class.
2. For each timeslot t and room r, we choose $Q^*(t, r) = Q_{l(t,r)}$ with probability 0.5, otherwise $Q_{r(t,r)}$. If this would create a duplicate assignment (class taught in two slots), we skip it.

Now we repair possible constraint violations:

3. For each class, if it is not held, we add it to a random free timeslot and room

Genetic Algorithm - Mutation Function `mutate`

`mutate` takes a timetable Q and returns a new timetable Q^*.

We define it as follows:

For each timeslot and room:

1. Mutate with probability ρ
2. Remove this class from the slot
3. Place it into a random other free slot

Genetic Algorithm - Selection & Iteration

We select the best timetables from the population using the evaluation function f. We sort the population by f and take the best $x\%$ of timetables.

Then we create a new population by making pairs and crossing them, then mutating the ‘children’ with probability ρ.[5]

We repeat this for the desired number of generations to obtain our final population. The result will then be the timetable from this population with the lowest evaluation score.

[5] e.g. If our selection split x is 10%, each pair will create 20 children to create a new generation of equal size to the previous one.

Genetic Algorithm - What about the Professor Constraint?

We did not fix the possible violation of the constraint that no professor teaches two classes at the same time.

We could handle this similarly to the other constraints, but I found that in this case, a different approach leads to better convergence behavior:

Instead of trying to fix non-compliant timetables, we just add a strong penalty to the evaluation function.

This way, the population will gradually evolve to not violate this constraint, as it will always be a large disadvantage.

Genetic Algorithm - Implementation

I implemented the described algorithm in Rust[6]. It takes the number of courses, rooms and timeslots ($|C| \leq |T| * |R|$!), and definitions for each student and professor. It can also generate mock data for testing.

For a more detailed description of the input format and how to use the program, consult the `README.md` file.

[6] See appendix for relevant code

Results

For demonstration purposes, I created three test sets:

> Note: Each Course is taught by one professor, so the number of professors is determined by `Cs/Prof`

> Note 2: Due to the random nature of the generated data, results will be much less optimal than in a real life scenario.

> Note 3: Due to the nature of `eval`, each test set has a minimum penalty, that's why we won't see scores lower that. More important is the convergence behavior.

Test Set	Students	Courses	Timeslots	Rooms	Cs/Stud	Cs/Prof
Small	100	16	10	3	3	2
Medium	256	50	20	5	8	5
Realistic	5000	100	40	10	20	10

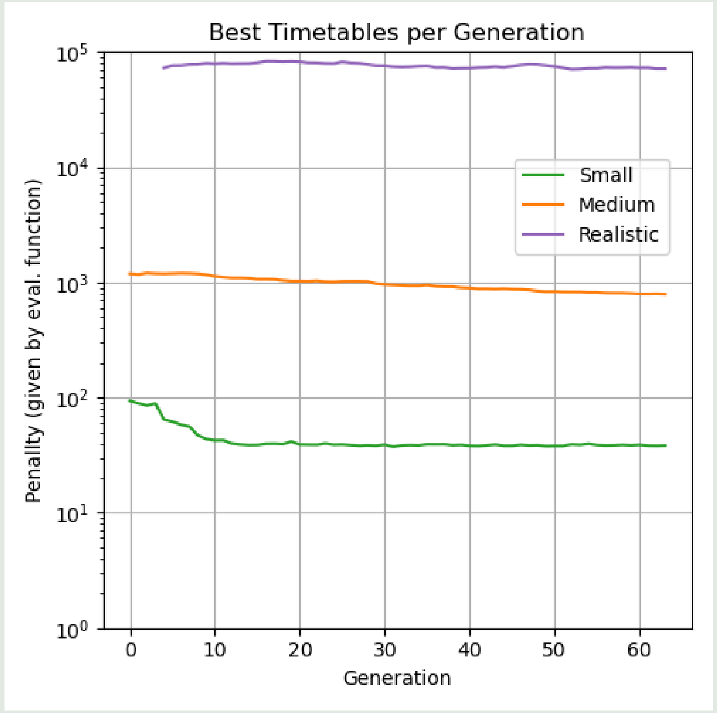
Charts Explanation

I'll be showing charts like the one on the right.

They show the evaluation function score of the best timetable in each generation for all three examples, on a logarithmic scale.

If there are parts of the lines missing, this means that in the respective generation, no valid timetable was found yet (all of them are in violation with the professor constraint).

Except for the parameter to be varied, the parameters for execution are:
`population size: 1024, selection split: 5%, mutation chance: 2%`[7]



[7]Selection split: Take the x% best timetables out of each generation to create the next one with.

Results - Varying Population Size

The small example isn't much affected, however the other examples show worse results with larger populations. This could be caused by less strict selection leaking bad traits into new generations.

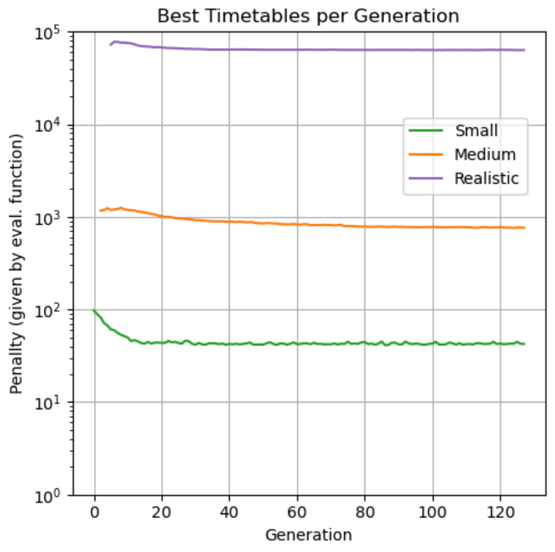


Figure 1: $|\text{Popul.}| = 128$

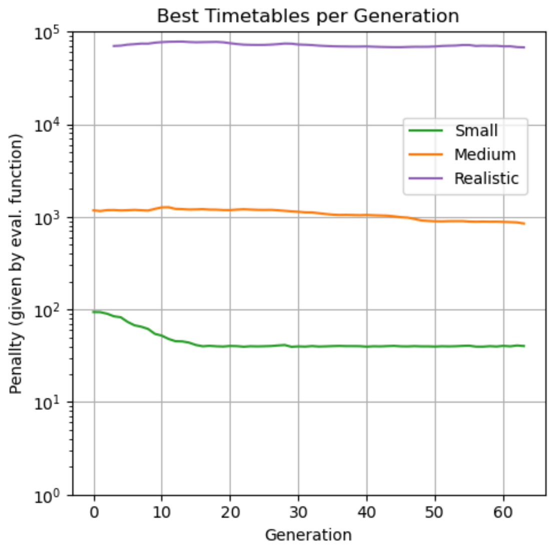


Figure 2: $|\text{Popul.}| = 1024$

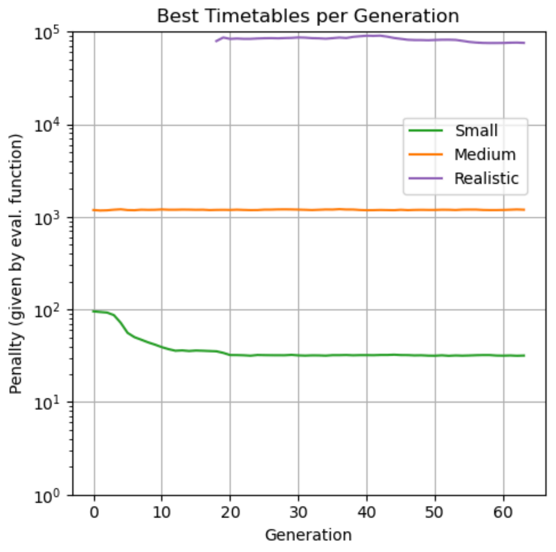


Figure 3: $|\text{Popul.}| = 4096$

Results - Varying Selection Split

The selection split has a big impact. A small split forces out bad timetables quickly, leading to faster convergence. Higher splits can even completely break the ability to find valid timetables (see the missing purple line in Figure 6).

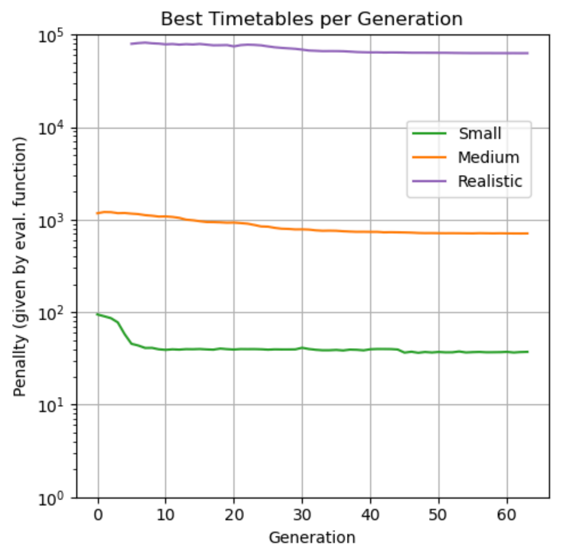


Figure 4: Split = 2%

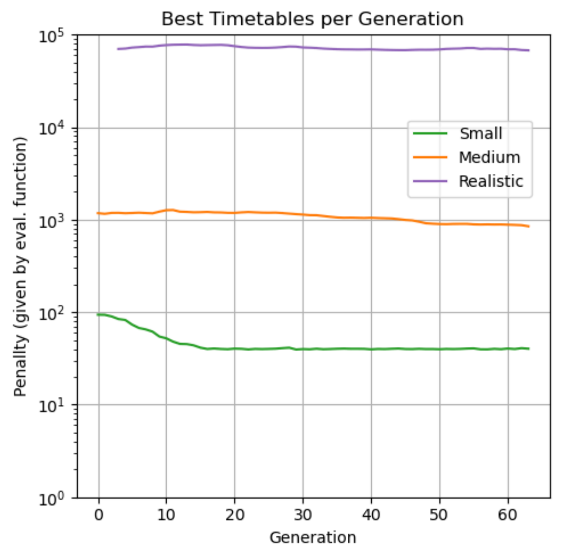


Figure 5: Split = 5%

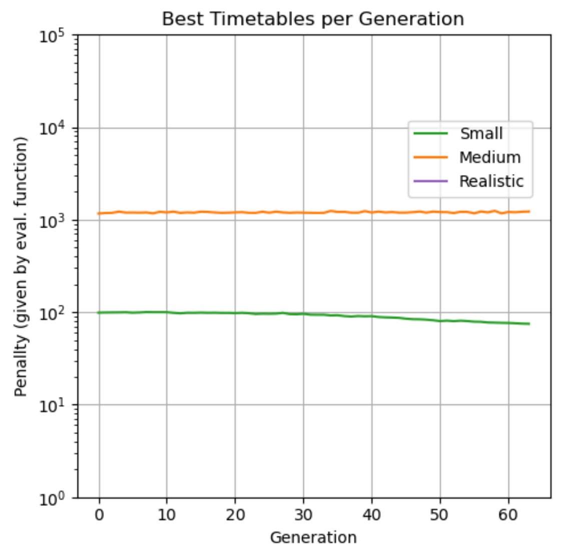


Figure 6: Split = 20%

Results = Varying Mutation Chance

Varying the mutation chance ρ (explained in the section about the Genetic Algorithm) also has a big impact. Low mutation rates lead to more smooth convergence, while high mutation rates can lead to faster convergence and even better results, although for the realistic example to high mutations show no convergence.

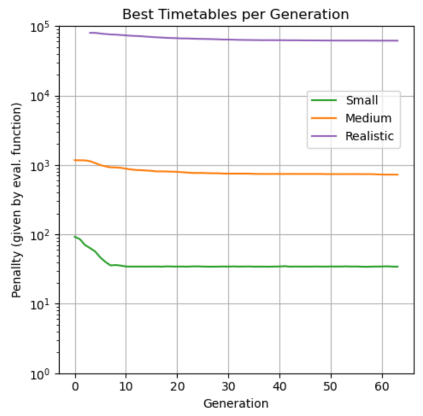


Figure 7: $\rho = 0.25\%$

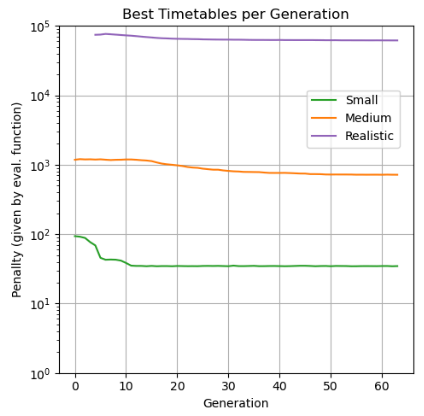


Figure 8: $\rho = 0.5\%$

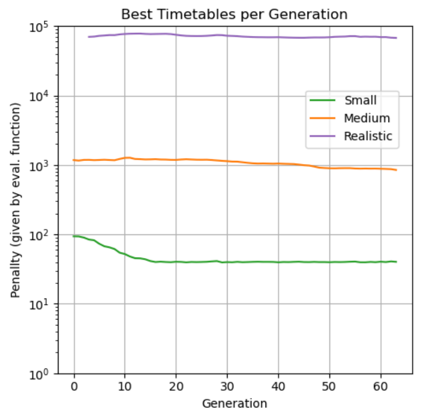


Figure 9: $\rho = 2\%$

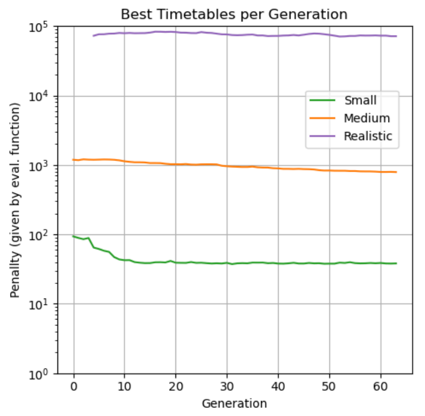


Figure 10: $\rho = 5\%$

Lessons Learned

- The Meta-parameters greatly influence effectiveness
- Same goes for problem modeling
- Not always obvious why a change leads to better results
- Implementation details can be tricky: Bugs, oversights, ..
- Pro: Very well parallelizable: almost linear speedup with more cpu cores! (Evaluations of genomes is completely independent, and make up most of the computation)

-> Genetic Algorithms are a powerful tool, but require a lot of experimentation and tuning

Future work, Ideas and Things to Improve

- The quality measure of larger examples is non-obvious. Thus, the effectiveness of this algorithm is not well measurable at the moment.
- A lot of per-dataset tweaking required: Some automatic hyperparameter adjustment could improve results.
- Real world data might bring different results and new insights

Conclusion

Genetic Algorithms are a good tool to approach optimization problems that benefit from stochastic search. For small to medium sized problems, they can be very fast and effective. For larger problems, a lot of fine tuning is needed to achieve good results. However, by taking inspiration from nature, simple functions for mutation and combining yield impressive results.

Find the code at https://github.com/vypxl/genetic_schedule_solver

Sources

[1] Javier Arias-Osori and Andrés Mora-Esquivel, "A solution to the university course timetabling problem using a hybrid method based on genetic algorithms." [Online]. Available: https://www.redalyc.org/journal/496/49668129006/html/

[2] Anmar Abuhamdah and Masri Ayob, "Adaptive randomized descent algorithm for solving course timetabling problems." [Online]. Available: https://academicjournals.org/journal/IJPS/article-full-text-pdf/0F9AE6934683