Universitat de les Illes Balears

High School schedule making using Evolutionary Algorithm

Professor:
Sebastián Massanet Massanet

Authors:
Adrian Sauter
Laura Häge
Natalia Cardona M

Assignment 1
December 9, 2022

Computational Intelligence

1 Introduction

To this day, schedule making is still often done by hand resulting in an exhausting and time consuming task. Moreover, many constraints have to be considered, some of which are mandatory, so called *hard constraints*, and some which would be additionally nice to achieve but are not a deal breaker, so called *soft constraints*. Breaking the constraints is referred to as *conflicts*. In the case of High School time schedules, a hard constraint could be that no lecturer is assigned to more than one class at the same time slot. A soft constraint could be that classes do not have the same subject more than two time slots a day.

1.1 Evolutionary Algorithm

One automatic way to tackle schedule making is using evolutionary algorithms. Evolutionary algorithms are inspired by Darwin´s natural selection approach *'the survival of the fittest'*. In the schedule making case, the population contains of individuals representing different time schedules. The main catch is to define a fitness function tailored specifically for the problem through integrating the hard and soft constraints. The *fitness* of each individual time schedule is defined by the fitness function. In other words, the *fitness* of each individual describes how well it meets the constraints. Similar to the biological role model, the *'fittest'* individuals are more likely to pass their (gene) information to the next generation. Furthermore, to alter the gene pool of the next generation, evolutionary algorithm tools like Mutation and Crossover/Recombination can be defined as well.

2 Method

2.1 Constraints

For creating the fitness function, constraints have to be defined. To integrate the difference between hard and soft constraints a weighting scheme is used that represents the importance of each constraint. Hard constraints have to be fulfilled and therefore get assigned the same large weight. Soft constraints receive a much smaller weight and are further differentiated according to their importance with corresponding weights.

2.1.1 Hard Constraints

Three hard constraints were defined. (H1) is already made to be impossible by the chosen representation of any individual. (H2) is also made impossible by the way an individual is initialized. While the individual is initialized randomly, it is still made sure that the required number of hours is met for every class. (H3) and (H4) are likely to appear due to the random initialization of the time tables and through crossover and mutation operations.

(H1) A class is assigned more than one subject for the same time slot

(H2) A class needs to have the assigned numbers of hours per course

(H3) A lecturer is assigned to more than one class at the same time slot (punishment of 10000)

(H4) A lecturer is assigned to a class when the he/ she is unavailable (punishment of 10000)

2.1.2 Soft Constraints

will keep it like that until we have our soft constraints fixed

Furthermore, some soft constraints were defined. The assigned weight of each constraint can be seen in the respective parentheses. The first soft constraint (S1) was chosen because in High Schools there are underage students who need supervision every day of the week. (S2) was chosen to avoid unnecessary waiting slots. (S3) was chosen to meet common working regulations. It didn't seem plausible to add a soft constraint which would punish if a teacher has free time slots, because it could very well happen that the teacher is not available and therefore should have those free time slots.

(S1) A class does not have has a single course on a day (punishment of 50)

(S2) A class has a break of longer than 1 time slot in between lectures (punishment of 50)

(S3) A class has a the same subject more than once on the same day (punishment of 50)

2.2 Fitness Function

The Fitness Function encodes all constraints and checks for conflicts. The *fitness* of each individual therefore represents all conflicts with any of the soft and hard constraints. Individuals that do have conflicts with constraints have a negative *fitness* score since conflicts are handled as a punishment. The respective weights of each conflicted constraint are subtracted from the *fitness* score. A individual with a *fitness* score of 0 meets all hard and soft constraints and therefore would be a perfect output.

2.3 Mutation and Crossover

The chosen mutation method has two probabilities with which it can be altered. The *mutation_class_prob* is the probability of a class to be mutated in general. If a random probability is smaller than the *mutation_class_prob*, the algorithm enters the loop in which the second probability steps in. The second probability which is called *mutation_slot_prob*, is the probability with which the content of each time slot of the class is randomly swapped with another the content of another time slot within the class.
For the Crossover, two individuals are chosen randomly as parents and with probability *crossover_prob*, a crossover takes place: with a 50% probability, class$_i$ ($i \in (1, 2, ...|classes|)$) of parent1 is appended to the child, otherwise, it will be class$_i$ of parent2. That way, there's a 50% chance for each class of each parent to be represented in the child. This process is repeated until the size of the new generation is the same as the size of the previous generation. If crossover doesn't take place (i.e. with probability (1-*crossover_prob*)), the parent with the better fitness value is kept unaltered and added to the new population.
Important note: We found random crossover to be leading to not that efficient results and explain that by the large amount of variety within each individual and the problem complexity in general. Due to this high variety, it's not necessarily good to randomly recombine two parents and therefore, a *smart_crossover* variant was implemented, which takes into account the number of conflicts (a teacher being assigned to two classes at the same time) and appends the next class from the individual with the least amount of conflicts. However, the results with that 'smart' crossover method weren't significantly better.

2.4 Selection of the next Generation

Evolutionary Algorithms need a selection scheme that defines how the next generation is made of. In this work the *tournament selection* was chosen. This method randomly selects num_inds_for_tournament (parameter of the algorithm) and keeps the individual with the highest fitness value. This is repeated until the size of the list of selected individuals is equal to the size of the previous generation.

2.5 Keep Elite Individuals

Keeping the best individuals in each generation without mutating them or making them eligible for crossover can help preserve good individuals without destroying them. Therefore, the best *num_elite* individuals are kept in each generation. After saving them, the entire population (including the best individuals) are passed through the crossover and mutation process. That way, the best individuals also have the chance to be made even better, but since the unaltered versions are kept as well, the possible good solutions are not lost. After that, the new population is put together again, consisting of the elite individuals which were saved before and randomly sampled individuals of the population that was created after crossover and mutation. This population has the same size as the old population (i.e. (*num_individuals* - *num_elite*) individuals are randomly sampled and *num_elite individuals* are kept unchanged).

3 Implementation

The programming language of choice in this work is python, of which the version 3.9.4 was used.

3.1 Necessary packages

The following packages need to be installed in order to run the code:

- copy
- pandas
- random
- numpy
- matplotlib
- tabulate
- json

3.2 Input Data

As input for the initialization of the algorithm, the following aspects have to be defined:

- classes and their required subjects (including the corresponding teacher and the amount of hours for each subject)
- teachers and their availability for each time slot
- the time slots during which the subjects can be taught to the students

The input was chosen to be in form of JSON files. Within the JSON file, the whole input is an object of three lists: **classes**, **teachers** and **time slots**. The list of classes contains objects for each class, representing the class name and the subjects with the respective amount of hours. For each subject, the name is defined as well as the respective teacher and the hours per week. The list of teachers contains teacher objects with the teacher names and their availability. The availability of each teacher is defined as a list of sublists, where each sublist is of the length of the time slots per day and therefore represents a day of the work week. The availability is encoded binary, meaning 1 stands for *available* and 0 stands for *unavailable*. The list of time slots defines the exact hours when the time slots take place exactly.

3.3 Representation

For the initialization of the population, one has to define what a single individual looks like. The representation of each individual in this work is a nested list that contains all time tables for all classes defined prior. The first sublist contains the time table of the first class, the second sublist contains the time table for the second class and so on. Each 'class-sublist' then consists of further sublists, each of which represents a day. In the inner-most list, the scheduled subject for that day and time slot for that specific class are contained in a tuple.
Example:

$$[[[[(class1\_monday\_firstslot\_subject\_and\_teacher),$$
$$(class1\_monday\_secondslot\_subject\_and\_teacher), ...$$
$$(class1\_monday\_lastslot\_subject\_and\_teacher)],$$
$$[(class1\_tuesday\_firstslot\_subject\_and\_teacher), ...], ...,$$
$$[(class1\_friday\_lastslot\_subject\_and\_teacher), ...]],$$

$$[[(class2\_monday\_firstslot\_subject\_and\_teacher),...],...]],...,$$

$$[...,[classn\_friday\_lastlot\_subject\_and\_teacher]]]$$

The whole population is a list of individuals. Since the nested list representation is not the most intuitive to read and understand, this representation is only used within the algorithm. The output is in easily human-readable form of a timetable for both, classes and teachers.

3.4 Functions

At the beginning, the classes ***Class*** and ***Teacher*** were defined for initializing the respective instances of classes and teachers.

The class ***Class*** contains the class name, an empty time table created with ***create_empty_timetable*** and the subjects with respective hours. Another 'subject' is initialized here that represents the *breaks*, where no classes take place. The amount of breaks a class has is calculated by the function ***fill_break_hours*** that simply subtracts the total amount of time slots by the amount of scheduled lectures for each class.

The class ***Teacher*** initializes the name of each teacher, an empty time table (again with ***create_empty_timetable***) and their availability table. This table is made with the function ***create_availability_table***, that loops over the availability list of each teacher that was given as input and creates a data frame with a binary encoding (1 = teacher is available for that time slot, 0 = teacher is not available for that time slot) of the availability of the teacher).

To initialize an individual, the function ***create_random_individual*** takes in the list of all classes. For each Class a random subject is picked for each time slot. Since the amounts of hours of each subject (including the breaks) is stored in the Class objects, they are reduced as soon as the respective subject is picked. In this way, the correct number of each subject a week is already given. This takes away the hard constraint that a solution of the algorithm must meet the required amount of hours per subject per class.

The mutation function ***mutate*** switches the subjects of two random time slots of a random class with probability of *mutation_slot_probability* percent, but only if the class is chosen to be mutated (which is affected by the probability *mutation_class_probability.*

The ***crossover*** function is the most complex one. The exact choice of it will be discussed in Section 5. The Crossover method has two options. If the parameter *smart_crossover* is set to *False* in the ***main***-function, then the first case is chosen. If it is set to *True* the second case is chosen.

1. Two parents are randomly selected. With probability *crossover_probability*, a child is created by crossing over the two parents in the following way: with a 50% chance either a class of one parent or of the other parent is picked to create a new individual. This is repeated until the next generation is of the same size as the initial population. However, with probability (1-*crossover_probability*), no crossover takes place and the parent with the higher fitness value is appended to the new population.

2. A predefined number of parents (default value = 10) is randomly sampled from the from the population. First, the first class of the parent with the best fitness value is appended to the child. Then, while iterating through the number of classes, the respective best class is added to the new individual. The best class is determined by its number of conflicts with the parts of the child that are already created. The number of conflicts here are the number of doubled teacher assignments to the same time slot if this class would be appended to the child. Therefore, the method tries to keep the number of doubled teacher assignments as low as possible. This process is repeated until the new generation has the same size as the initial population.

For the selection of the individuals for the new generation, ***tournament_selection*** was chosen. A number of random parents are chosen (default value is 5), those are sorted by their fitness values. Out of them, the one with the best fitness value is selected to be part of the next generation. This is repeated until the next generation is of the same size as the initial population.

To evaluate the population, each individual gets evaluated with the function ***get_fitness_value*** that loops through the population and calls ***evaluate_individual*** for each individual. The function ***evaluate_individual*** contains the hard and soft constraints that have been defined. It checks how many times a teacher is scheduled for more than one class in a time slot (with ***check_double_teacher(teachers)***), how many times a teacher is scheduled for a time slot when their unavailable (with ***teacher_not_available(teachers)***), how many days a class has no lectures at all (with ***zero_classes(classes)***), how many times a class has two or more free time slots in between lectures (with ***free_timeslots(classes)***) and how many times a class has the same subject twice on the same day (with ***same_subject_same_day(classes)***). All those occurrences get punished since the represent conflicts.

The ***evolutionary_algorithm*** function executes the whole algorithm. Firstly, an initial population consisting of random individuals and of size *num_individuals* is created. Then, the initial population is shuffled (by mutation with probability of 1), since it is logically the case that the last time slots only contained 'breaks' because in most examples, the 'Break-subject' has the highest number of hours compared to any other single subject and due to the process of initializing an individual with the number of required hours being decremented by 1 when the subject is chosen. This process and the fact that the 'Break-subject' has the highest number of hours lead to the 'Break-subject' to likely be left in the end as the last possible option (meaning the last 'subject' with hours != 0) to be chosen for the individual. Similar to the ***crossover***, depending on the value of the parameter *keep_elite* there are two cases, the first one is executed when the value is *False*, the second case is executed if the value is *True*.

1. The ***crossover***, the ***mutation*** and the ***tournament_selection*** function are executed for the number of defined iterations.

2. In this case, the best few individuals (default value = 10) are 'put aside'. In this way it is avoided that good solutions are mixed up again. Then, the entire population goes through the crossover and mutation process. This includes the elite individuals as well, since there may be a chance to improve them through these methods. However, because we also kept the original elite versions of them, we can add them to the next population as well in case they got worse through crossover and mutation.

If a perfect individual (with a fitness score of 0) is found during the iterations, the algorithm gets stopped. Otherwise, the average fitness value and the best fitness value of the current population is printed out.

For a nice representation of the results three functions were defined: ***create_nice_class_timetable*** creates easily readable time tables for each class. The function ***create_nice_teacher_timetable*** creates timetables for each teachers which are color-coordinated. If a teacher is available at the given time slot the font will appear green, otherwise the font is red. The function ***plot_fitness*** plots the development of the fitness value during the iterations.

4 Results

The evolutionary algorithm proposed in this paper is investigated under different circumstances. As input, different JSON files were created. For better comparison, the same amount of time slots and classes were chosen for each example. Furthermore, the total amount of hours each class is taught a week was set to the same number as well. We are aware that due the random initialization the results of the algorithm vary from each run to run. This means, when rerunning the code, it is likely that the output will be slightly different each time. However, for illustrative reasons some example results are presented in the following.

4.1 Performance of the Algorithm

To test the power and performance of the algorithm in different scenarios, three examples of varying difficulty were chosen. The initialization choices for the algorithm were: **500 iterations**, **500 individuals** per generation, **Crossover Probability of 0.5**, the **Probability of class mutation is 0.5** and the **Probability of time slot mutation is 0.33**. For all examples the parameters of **keep_elite is set to True** and **smart_crossover is set to False**.

4.1.1 Easy Time Scheduling Example

The first example was chosen to be a 'non-overlapping, high availability' example, meaning that no two classes have the same teacher ('non-overlapping') and all teacher have a high availability rate when it comes to holding lectures. In total, there are three classes and six teachers. In this case, each teacher has a workload of six hours a week, since they only teach one class. As expected, the algorithm easily finds a perfect solution within 500 iterations. Depending on the random initialization, this can happen even within a few iterations. In this case, the hard constraint *H3* is already met: It is not possible for a teacher to be assigned to multiple classes at once. Therefore, the algorithm only has to optimize for *H4* and the soft constraints. In this way, the penalty is much smaller and a perfect result can be found quicker. The file that was used for this example is *easy.json* and one result of the algorithm is shown in 7.1. The average fitness values are plotted in figure 1. The first random initialized best individual had a fitness value around -10000. It is visible that the average fitness value increased drastically within the first iterations. Afterwards, an alternating behavior can be observed. However, after around 280 Iterations the algorithm found a perfect solution (fitness value of 0) and stopped. This means the found individual meets all hard and soft constraints.

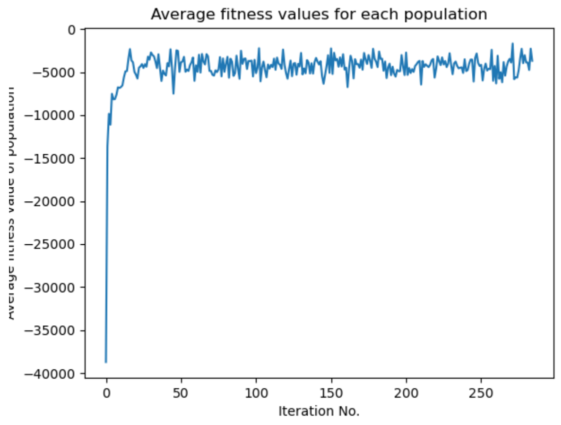

Figure 1: Average Fitness Values, Easy example

4.1.2 Advanced Time Scheduling Example

The next example that was chosen contains again six teachers and three classes. In this case however, multiple classes can have the same teachers. Here, the algorithm has to optimize for *H2* as well, since teachers can be assigned to multiple classes at one time slot. Furthermore, some teachers have a higher workload due to the assignments to multiple classes. This example requires significantly more iterations and computational power. This can be also seen in the best fitness value of the first random initialized example population of around -60.000. The average fitness value shows a continuous increase. However, for resources and comparison reasons we stopped the algorithm after 500 iterations. For 500 iterations the algorithm takes a few hours. During multiple runs we found a perfect solution. However, for transparency reasons we display the solution of the final 'test' run that did not find a perfect solution. The **final best fitness value** was **-10.200**. The file that was used for this example is *advanced.json*. One result is shown in figure 7.2.

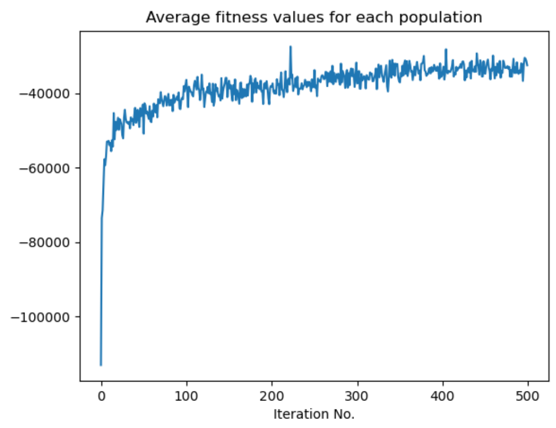

Figure 2: Average Fitness Values, Advanced example

4.1.3 Difficult Time Scheduling Example

For the most difficult example, three classes and three teachers are chosen. Again, multiple classes can have the same teachers. The fewer amount of teachers increases the probability of teachers being assigned to multiple classes at once and therefore, it is more likely that the hard constraint *H2* is violated. The workload of each teacher is now 12 hours a week. Furthermore, the teacher availabilities were reduced. However, this was done in a way that perfect solutions still are possible. Less teacher availability increases the chance of breaking hard constraint *H3*. The input file that was used here is *difficult.json*. In conclusion, this example offers more difficulties in finding a good or perfect solution. This can also be seen in the first best fitness value (around -100.000) of the initial population. The Algorithm shows an continuous increase of the fitness value, however for better comparison we stopped the algorithm after 500 iterations as well. **The final best fitness value** was **-20.300** One result of the algorithm is presented in 7.3. The development of the average fitness value during the iterations can be seen in figure 3.

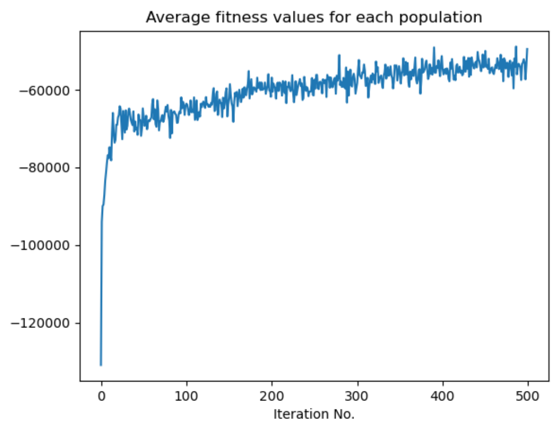

Figure 3: Average Fitness Values, Difficult example

4.2 Impact of Function Variations

During the process of testing the algorithm, two modifications were added to investigate their impact on the performance.

4.2.1 Impact of Smart Crossover

The first modification is a **smart Crossover** function instead of a random one. The idea behind this crossover approach is a guided crossover, meaning that a fixed number of parents are chosen and then, while iterating over all classes to create an individual, the class from that parent with the fewest number of conflicts with the already created child is appended to that child (see 3.4 for a more detailed explanation of the smart Crossover method). This should result in an individual with less conflicts (conflicts in the sense of a reduced number of double teacher assignments to the same time slot). The results however showed minor improvement to the randomized crossover method. The algorithm did not perform significantly better or faster with smart Crossover.

4.2.2 Impact of Keep Elite Individuals

The other modification of the algorithm is the **Keep Elite** Individual modification. As mentioned earlier, this method removes the best *num_elite* individuals out of the population. The Mutation and Crossover method are then applied to the entire population (including the best individuals). This ensures that the the best individuals have the chance to get even better. However, since we stored the original version of the best individuals, it's ensured that if they get worse, we still have their previous, better version stored. This has lead to a significant improvement of the algorithms performance and solutions. In fact, before applying this modification, the best fitness value of a population started to alternate at one point ending up in a local optimum. To compare how the algorithm performs without this modification, an example run of the *easy.json* file is executed. The results of the average fitness values are illustrated in Table 1 and Figure 4. It is clearly visible, that there is no continuous improvement in the best fitness value even after 300 iterations.

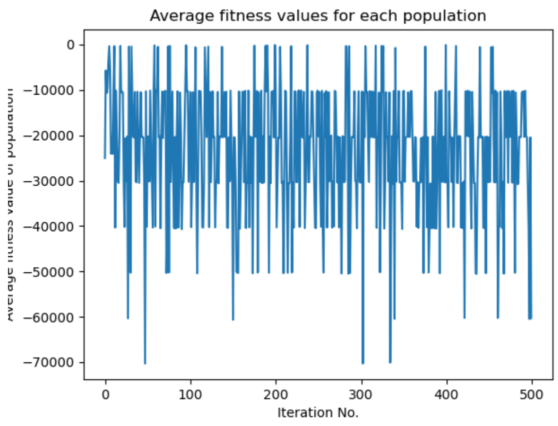

Figure 4: Average Fitness Values, Keep Elites turned off

5 Discussion

5.1 Online solutions

While we're aware that there are a lot of solutions for this algorithm online which can be adapted to work for this specific problem, we decided to build our own algorithm from scratch. That way, it's made sure that every single function and process of the algorithm is understood completely and can be explained by us.

5.2 Previous approaches

The version we submitted is the 4^{th} approach of the algorithm. First, we experimented with different representations, all of which where in the form

Iteration	Best Fitness Value of the current Population
1	-50
2	-100
3	-100
4	-150
5	-150
...	...
30	-20.350
31	-450
32	-10.250
...	...
100	-20.450
101	-10.450
102	-30.300
...	...
200	-100
201	-40.300
...	...
300	-30.450
301	-10.350
302	-70.300

Table 1: Average Fitness Values of a Run of the Easy Example

of binary codes which represented a subject and a teacher. The binary representation made the mutation more efficient because it offered the possibility of bit flips for mutating a single code. However, we found that by the representation we chose for the final version, there are only two hard constraints (H3) and (H4) to be taken care of by the algorithm. The other hard constraints were already made to be impossible due to the representation and initialization of individuals. Furthermore, in the binary case one had to check if the random bit flips created new teachers or subjects that have not been defined before.

5.3 Evaluating the Performance

The Evolutionary Algorithm proposed in this paper is able to solve easy time scheduling tasks perfectly and fast. In more advanced and difficult cases the algorithm takes significantly more computational power and iterations to provide a perfect or good solution. The results verify that the chosen soft and hard constraints in this work are goal-oriented and provide a basis for good solutions.

During the process, multiple aspects were recognized about the influence on the performance of the algorithm. The examples that were chosen to test the algorithm, showed some significant differences in difficulty. The main influences that were observed are listed here:

- From switching from the easy to the advanced example, it got clear that if teachers are assigned to multiple classes, the problem that the algorithm has to solves becomes a lot harder. In the easy example, no class shared the same teacher with another class. This was changed in the advanced example. It showed that the possibility of multi-class-assignments in one time slot of a teacher cause noticeable decrease in the performance.

- Furthermore, the number of subjects each teacher has to hold has a huge impact on the performance. This could be observed when comparing the advanced example to the difficult one. In both of those examples three classes were defined and all classes have a total amount of twelve hours a week. The difficult example however, only contains three teachers instead of six. Therefore the workload of each teacher is a lot higher and the chances of running into conflicts with their unavailability are higher as well.

- Additionally, increasing the unavailabilty rate of the teachers influenced the performance in all three examples negatively.

Those observations can be compressed into two formulas. The lower the workload of the teachers and the lower the class-sharing-rate, the better the performance of the algorithm.

5.4 Modification of the Algorithm

Due to ongoing issues with the original idea of an evolutionary algorithm, that simply contained basic recombination and selection operators, we decided to 'boost' the Algorithm. With keeping the elite individuals out of the recombination operators (as described in 4.2.2), the algorithm finally showed results that could not be achieved even with parameter-tuning. Before that modification the algorithm kept on getting stuck in local optima and did not show any form of improvement over large amount of iterations. We concluded, that the random recombination operators probably mix up the individuals too much. In that way already good ones were messed up again. We tried multiple variations of the Crossover and Mutation method but nothing showed a significant improvement. What was surprising on our side, is that the *smart Crossover* method did not help the performance noticeably (4.2.1).

6 Outlook

6.1 Temperature parameter

Possible improvements on the algorithm could include a temperature-parameter which would affect the crossover- and mutation-probabilities. This could be implemented by starting with high probabilities for finding possible good solutions and then reducing these probabilities by a bit in each generation in order to just apply small changes to good solutions in the hope of improving them without changing to much (which could make good individuals bad again).

6.2 More soft constraints

Additionally, more soft constraints could be added. One possible soft constraint could for example regard the teachers. A teacher shouldn't have more than one free time slot in between lectures IFF he/she is available during these time slots.

7 Apendix

Example results that we encountered while running the algorithm are illustrated below. However, please note that the algorithm prints out much nicer time schedules.

7.1 Results Easy Example

Time Schedule: Class 1

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	Math Teacher: A	Math Teacher: A	Break	Break
9:45-11:15	Break	Break	Physics Teacher: B	Break	Informatics Teacher: B
11:30-13:00	Physical Education Teacher: A	Chemistry Teacher: B	Break	Biology teacher: B	Break
14:00-15:30	English Teacher: A	Spanish Teacher: A	English Teacher: A	Break	Chemistry Teacher: B
15:45-17:15	Break	Break	Break	Break	Break
17:30-19:00	Break	Informatics Teacher: B	Break	Break	Break

Time Schedule: Class 2

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	Break	Break	Break	Break
9:45-11:15	Chemistry Teacher: D	Math Teacher: C	Break	Physics Teacher: D	Break
11:30-13:00	Informatics Teacher: D	Break	Break	Break	Break
14:00-15:30	P.E. Teacher: D	Spanish Teacher: C	English Teacher: C	Informatics Teacher: C	Spanish Teacher: C
15:45-17:15	Break	P.E. Teacher: D	Break	Break	Break
17:30-19:00	Math Teacher: C	Biology Teacher: C	Break	Break	Break

Time Schedule: Class 3

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	Math Teacher: E	Informatics Teacher: F	Break	Break
9:45-11:15	Spanish Teacher: E	Break	Break	Chemistry Teacher: F	P. E. Teacher: F
11:30-13:00	Math Teacher: E	English Teacher: E	Chemistry Teacher: F	Break	Break
14:00-15:30	Break	Break	Physics Teacher: F	Informatics Teacher: F	Break
15:45-17:15	Break	Biology Teacher: E	Biology Teacher: E	Break	Break
17:30-19:00	Break	Break	Break	Break	Break

Time Schedule: Teacher A

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	Class 1 Math	Class 1 Math	-	x
9:45-11:15	-	x	-	-	-
11:30-13:00	Class 1 Physical Education	-	-	-	-
14:00-15:30	-	Class 1 Spanish	Class 1 English	-	-
15:45-17:15	x	-	x	-	-
17:30-19:00	Class 1 English	-	-	-	-

Time Schedule: Teacher B

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	x	-	-	-	-
9:45-11:15	Class 1 Biology	x	Class 1 Physics	-	Class 1 Informatics
11:30-13:00	-	Class 1 Chemistry	-	Class 1 Biology	-
14:00-15:30	-	-	-	-	Class 1 Chemistry
15:45-17:15	-	-	-	x	-
17:30-19:00	-	Class 1 Informatics	-	-	-

Time Schedule: Teacher C

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	-	-	-	-
9:45-11:15	-	Class 2 Math	-	-	-
11:30-13:00	x	-	-	-	x
14:00-15:30	-	Class 2 Spanish	Class 2 English	-	-
15:45-17:15	-	-	x	-	Class 2 Spanish
17:30-19:00	Class 2 Math	Class 2 Biology	-	-	-

Time Schedule: Teacher D

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	-	-	-	-
9:45-11:15	Class 2 Chemistry	-	x	Class 2 Physics	-
11:30-13:00	Class 2 Informatics	x	-	-	-
14:00-15:30	Class 2 P.E.	-	-	Class 2 Informatics	-
15:45-17:15	x	Class 2 P.E.	-	-	-
17:30-19:00	-	-	-	x	-

Time Schedule: Teacher E

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	Class 3 Math	x	-	-
9:45-11:15	Class 3 Spanish	-	-	-	x
11:30-13:00	Class 3 Math	Class 3 English	-	-	-
14:00-15:30	-	-	-	-	-
15:45-17:15	-	Class 3 Biology	Class 3 Biology	-	-
17:30-19:00	-	-	-		-

Time Schedule: Teacher F

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	-	Class 3 Informatics	-	-
9:45-11:15	-	-	-	Class 3 Chemistry	Class 3 P. E
11:30-13:00	-	-	Class 3 Chemistry	x	-
14:00-15:30	-	-	Class 3 Physic	Class 3 Informatics	-
15:45-17:15	-	-	-	-	-
17:30-19:00	-	x	-	-	-

7.2 Results Advanced Example

Time Schedule: Class Cougars

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	English Chomsky	Break	Biology Mr. White	Break
9:45-11:15	P.E. Mr. Johnson	Break	English Mr. Chomsky	Break	Acting Ms. Johnson
11:30-13:00	Math Mr. Euler	Break	Break	Break	Break
14:00-15:30	IT Mr. Euler	Break	Break	Break	Politics Ms. Freud
15:45-17:15	Break	Chemistry Mr. White	Break	History Ms. Freud	Break
17:30-19:00	Physics Mr. White	Spanish Mr. Chomsky	Math Mr. Euler	Music Ms. Freud	Spanish Mr. Chomsky

Time Schedule: Class Dolphins

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	French Ms. Chomsky	Break	Break	Economy Mr. Musk
9:45-11:15	Break	Chemistry Mr. White	Chinese Mr. Musk	Break	Break
11:30-13:00	P.E Mr. Johnson	Spanish Ms. Chomsky	Break	Break	Break
14:00-15:30	Politics Ms. Freud	Break	Math Mr. Euler	Spanish Ms. Chomsky	Break
15:45-17:15	Break	Break	Art Ms. Freud	Break	Break
17:30-19:00	Math Mr. Euler	Statistics Mr. Euler	Break	Psychology Ms. Freud	Break

Time Schedule: Class Eagles

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	Break	Break	Break	Break
9:45-11:15	Break	Chinese Mr. Musk	Break	Break	Break
11:30-13:00	Spanish Ms. Chomsky	P.E. Ms. Johnson	Biology Ms. Chomsky	Acting Mr. Johnson	Math Mr. Euler
14:00-15:30	Spanish Ms. Chomsky	History Ms. Freud	Break	Break	Chinese Mr. Musk
15:45-17:15	Math Mr. Euler	Music Ms. Freud	Break	Break	Latin Mr. Musk
17:30-19:00	Break	Astronomy Mr. Musk	Break	Break	Break

Time Schedule: Mr. Euler

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	-	x	-	-
9:45-11:15	-	-	-	-	-
11:30-13:00	Class Cougars Math	-	-	-	Class Eagles Math
14:00-15:30	Class Cougars IT	-	Class Dolphins Math	-	-
15:45-17:15	Class Eagles Math	-	-	-	x
17:30-19:00	Class Dolphins Math	Class Dolphins Statistics	Class Cougars Math	-	-

Time Schedule: Ms. Chomsky

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	x	Class Cougars English Class Dolphins French	-	x	-
9:45-11:15	-	-	Class Cougars English	-	-
11:30-13:00	Class Eagles Spanish	Class Dolphins Spanish	Class Eagles Biology	-	-
14:00-15:30	Class Eagles Spanish	-	-	Class Dolphins Spanish	-
15:45-17:15	-	-	-	x	-
17:30-19:00	-	Class Cougars Spanish	x	x	Class Cougars Spanish

Time Schedule: Mr. White

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	x	-	Class Cougars Biology	-
9:45-11:15	-	Class Dolphins Chemistry	-	x	-
11:30-13:00	-	-	-	-	-
14:00-15:30	-	-	-	-	-
15:45-17:15	-	Class Cougars Chemistry	-	-	-
17:30-19:00	Class Cougars Physics	-	x	-	x

Time Schedule: Ms. Freud

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	x	-	x	-	-
9:45-11:15	x	-	-	-	-
11:30-13:00	-	-	-	-	-
14:00-15:30	Class Dolphins Politics	Class Eagles History	-	-	Class Cougars Politics
15:45-17:15	-	Class Eagles Music	Class Dolphins Art	-	x
17:30-19:00	-	x	-	Class Cougars Music Class Dolphins Psychology	x

Time Schedule: Mr. Johnson

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	x	-	x	-	-
9:45-11:15	Class Cougars P. E.	-	-	-	Class Cougars Acting
11:30-13:00	Class Dolphins P. E.	Class Eagles P. E.	-	Class Eagles Acting	-
14:00-15:30	-	-	-	-	-
15:45-17:15	-	-	-	-	-
17:30-19:00	x	-	-	-	x

Time Schedule: Mr. Musk

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	x	-	-	x	Class Dolphins Economy
9:45-11:15	x	Class Eagles Chinese	Class Dolphins Chinese	x	-
11:30-13:00	-	-	-	-	-
14:00-15:30	-	-	x	-	Class Eagles Chinese
15:45-17:15	-	-	-	-	Class Eagles Latin
17:30-19:00	-	Class Eagles Astronomy	x	-	x

7.3 Results Difficult Example

Time Schedule: Class 1

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	P. E. Mr. Propper	Break	Break	Break	Break
9:45-11:15	Spanish Ms. Smith	Break	English Ms. Smith	Break	Math Ms. Alice
11:30-13:00	Break	Math Ms. Alice	Spanish Ms. Smith	Biology Ms. Alice	Math Ms. Alice
14:00-15:30	Break	Math Ms. Alice	Break	Break	Break
15:45-17:15	Break	English Ms. Smith	Break	Break	Break
17:30-19:00	P. E. Mr. Propper	Break	Biology Ms. Alice	Break	Break

Time Schedule: Class 2

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Break	P. E. Mr. Propper	Break	Math Ms. Smith	Break
9:45-11:15	Break	Spanish Mr. Propper	Math Ms. Smith	Biology Ms. Alice	Break
11:30-13:00	English Ms. Smith	Spanish Mr. Propper	Break	English Ms. Smith	P.E Mr. Propper
14:00-15:30	Break	Break	Break	Break	Break
15:45-17:15	Break	Break	Break	Break	Spanish Mr. Propper
17:30-19:00	Break	Break	Biology Ms. Alice	Spanish Mr. Propper	Break

Time Schedule: Class 3

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Spanish Mr. Propper	Break	Break	P.E. Mr. Propper	Break
9:45-11:15	Break	Break	Break	Break	Break
11:30-13:00	Break	Math Ms. Alice	Break	Break	Break
14:00-15:30	English Ms. Smith	Break	Math Ms. Alice	P.E. Mr. Propper	Break
15:45-17:15	English Ms. Smith	Break	English Ms. Smith	Biology Ms. Alice	Biology Ms. Alice
17:30-19:00	English Ms. Smith	Spanish Mr. Propper	Break	Break	Break

Time Schedule: Teacher Mr. Propper

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	Class 1 P.E. Class 3 Spanish	Class 2 P.E.	-	Class 3 P.E.	-
9:45-11:15	x	Class 2 Spanish	-	-	-
11:30-13:00	-	Class 2 Spanish	-	-	Class 2 P.E.
14:00-15:30	-	-	-	Class 3 P.E.	-
15:45-17:15	-	x	-	x	Class 2 Spanish
17:30-19:00	x	x Class 3 Spanish		x Class 2 Spanish	x

Time Schedule: Teacher Ms. Smith

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	x	-	-	-
9:45-11:15	Class 2 Spanish	-	Class 1 English	-	-
11:30-13:00	Class 2 English	-	Class 1 Spanish	Class 2 English	-
14:00-15:30	Class 3 English	-	-	x	-
15:45-17:15	Class 3 English	Class 1 English	Class 3 English	x	-
17:30-19:00	Class 3 English	x	x	x	x

Time Schedule: Teacher Ms. Alice

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00-9:30	-	x	-	-	x
9:45-11:15	-	-	-	Class 2 Biology	Class 1 Math
11:30-13:00	x	Class 1 Math Class 3 Math	-	Class 2 Biology	Class 1 Math
14:00-15:30	-	Class 1 Math	Class 3 Math	-	-
15:45-17:15	-	-	-	Class 3 Biology	Class 3 Biology
17:30-19:00	-	-	Class 1 Biology Class 3 Biology	x	x