Utrecht University
Advanced Planning and Decision Making

Master Thesis

Timetabling at Utrecht University

Author:
H.C. Kampman

Supervisors:
dr. J.A. Hoogeveen
dr. ir. J.M. van den Akker

August 30, 2013

Abstract

In this thesis we examined the possibilities to automate the roomscheduling of the lectures given at Utrecht University. The amount of lectures that has to be given at Utrecht University in combination with the complexity of the problem (NP-Complete) makes this problem hard to solve with the current system, which uses a greedy algorithm. We successfully implemented a local search to solve the original problem including extra constraints that were desired by the schedulers at Utrecht University. To decrease the running time and at the same time fulfilling the regularity constraint we applied a 2-phase approach: first a *stamp* is created, which consists of the lectures that should be repeated every week. This stamp is then applied for every week to ensure the regularity of the lectures. In the second part of the approach the incidental lectures are scheduled to create a complete timetable. In this thesis we examined several algorithms for generating the initial timetable, as well as a Branch and Bound heuristic in the local search, accompanied by other problem-specific operators. This resulted in an algorithm that is far more superior than a greedy algorithm such as the algorithm of Syllabus Plus.

1 Introduction

In this section we first give the motivation of this research, followed by a problem description. Next we describe how the current system tackles this problem, with the use of Syllabus Plus[1]. After that we give a summary of the research done in this field. We conclude the introduction with an overview of the rest of this thesis.

1.1 Motivation

In a previous research we looked into the timetabling of exams at Utrecht University. This problem was really challenging due to the practical constraints and the size of the problem. All the exams had to be scheduled in two weeks, where the room- and time-assignment was essential. In this case there were only a few large rooms and relatively many small rooms, which made the choice of determining which exams should be held in which room very difficult. During this research we had the chance to look under the hood of the timetabling system used at Utrecht University. Seeing the effect of bringing structure to this particular problem we realized the system as a whole could use more structure. Therefore we decided to show what the effect would be when there is more structure in the input data, giving a local search handles to operate. The introduction of a local search might lead to better timetables.

As students we had come across timetabling issues first-hand. Some subsequent lectures where scheduled in rooms that were geographically far apart from each other, which made it impractical to get on time for the next lecture. This was mostly the case when a lecture was scheduled in a room of a different faculty. This brings another practical problem: students had a hard time *locating* the room, since they were not familiar with the building (since it belongs to a different faculty). Another problematic tendency is that lectures were more often scheduled at unpopular points of time, like early in the morning or after 17.00. This is bothersome for the students as well as the teachers, which does not contribute to the quality of the lectures.

Internally, Utrecht University has several points for improvement. The main point of improvement concerns the room occupation; this costs Utrecht University a lot of money and it is a considerable potential for cost savings. As students of Utrecht University we like to contribute by doing research in this area. It would be great if this paper would lead to an improvement of the current system used by Utrecht University.

1.2 Problem Description

Every student at Utrecht University chooses which courses he wants to follow for every timeperiod; the students are completely free in which courses they pick. To make sure that the lectures of the chosen courses do not overlap, Utrecht University introduced *timeslots*. A timeslot defines in which time-window a course may give their lectures. For instance: a course with timeslot A is allowed to schedule their lectures on Monday morning, Tuesday evening and Wednesday morning. The timeslot-model provides the student some guidance to pick non-overlapping courses.

As an addition to the current situation the schedulers at Utrecht University wanted to be able to define time-dependencies between lectures of the same course. For instance: the teacher

[1]Syllabus Plus is developed by Scientia: http://www.scientia.com/uk/

wants to first give a global introduction, which all the students should attend. After that the students should split up and do a practical assignment. The practical assignment has to be worked out by the students, which takes approximately one or two days, so multiple seminars should be scheduled about two to three days after the practical assignment. In this situation a chain of lectures is described with time-dependencies between each lecture; the starting time of the first lecture influences the possible starting times of the last lecture. Therefore each lecture should be scheduled carefully. This extra feature constraints the time-windows defined by the timeslots even further.

Next to the determination of the starting times of a lecture a *room* has to be assigned to each lecture. Not every room of Utrecht University is a suitable room for each lecture; rooms have different equipment available, like blackboards, beamers, computers and so on. Next to these clearly defined requirements there are also less tangible requirements like the fact that students do not want to go to unfamiliar places. Therefore *roomsets* are introduced, which define in which rooms a lecture can be given.

Schedulers at Utrecht University have the possibility to put lectures in external rooms. Typically these lectures have a very high amount of participants. However, external rooms are expensive and the usage of these rooms should therefore be minimized.

A course has a certain structure of lectures; for instance, every week there are two normal lectures and two practical lectures. One of the demands from the students and teachers is that all of these regular lectures should have the same starting time and should be in the same room every time the lecture is given.

At a large educational organization as Utrecht University there is a high chance of unforeseen incidents, such as the maintenance of a building or other sudden events that prevent a room from being used. These events can mix up the carefully constructed timetable completely, since the lectures that were scheduled in these rooms now have to be scheduled in other rooms which could lead to one giant puzzle. To prevent the timetable from being transformed into a difficult jigsaw an empty-rooms-threshold can be set. This threshold indicates how many rooms have to be empty at any given time; this creates a buffer for emergency situations.

Under these constraints the lectures at Utrecht University have to be scheduled. Clearly, this is impractical to do by hand.

1.3 Current system

The current system uses Syllabus Plus to schedule the lectures. The assignment of rooms and starting times for every lecture is done by an internal black-box algorithm. The running time of Syllabus' algorithm is only a couple of seconds, which leads to the conclusion that some kind of greedy search is inside the black box. Each lecture receives a bunch of rooms to tell Syllabus what the room-options are for each lecture, based on (niggling) demands from teachers like the distinction of blackboards from whiteboards. Utrecht University also supports the idea that the distinction between rooms based on such small aspects is objectionable; the institution wants to have more uniform rooms so they can permute the roomassignement of a lecture more easily. When Syllabus cannot find a feasible solution it returns the lectures which could not be scheduled. The scheduler then increases the roomset for these lectures by dropping several constraints argued by the teachers and Syllabus then re-tries to find a feasible solution. This is repeated until a feasible solution is found. This process is very time consuming and labor-intense, and should therefore be improved.

1.4 Literature

Timetabling in general causes many headaches for the people who have to create and maintain the timetables, independent from whether the timetables are for hospital staff, train or bus drivers, firefighters or schools. Since these challenging problems are so common and applicable in real life a lot of research has been done in this field. In this literature study a collection of papers regarding the timetabling of schools is looked into. The papers are categorized in three different groups:

- Exam timetabling
- Student-centered timetabling
- Curriculum-based timetabling

Although the timetabling of exams is different from the normal lecture timetabling (student centered or curriculum-based) they do have some very similar properties; for instance, in both timetables there might be clashes between meetings/exams and each meeting/exam has to fit in the room.

1.4.1 Exam timetabling

One of the main problems with the timetabling of examinations is the difficulty in evenly spreading the exams for every single student. It is already hard to make a feasible solution where each student does not have overlapping examinations, but the soft constraint of having a decent individual timetable for every student makes the problem very difficult.

Big improvements can be made by clustering exams that can be held at the same time. This creates a better overview for the scheduler so that he can find a valid solution more easily. Johnson (1990) developed heuristics that made the life of the scheduler easier by introducing heuristics that determine which exams can be held at the same time.

Mansour et al. (2011) developed a scatter search approach to create feasible timetables in which the students had good timetables: not more than two exams on the same day and the time between exams should be as long as possible. Scatter search is a variant of the more commonly used genetic search; however, scatter search is less randomized, which makes it converge faster. This approach resulted in timetables that were significantly better than the timetables produced by extensive manual timetabling.

A faster converge rate is the result of less wider exploration of the search-space. Burke et al. (1996) implemented a memetic algorithm which focuses on a hill-climbing algorithm. When stuck in a solution multiple simple mutation operators are used upon the optimized solution to escape from the local optimum. The mutated solution is then again optimized by applying hill-climbing operators. This approach resulted in a fast algorithm, which spent most of its time on hill-climbing because the mutation operators were kept simple. However, the algorithm proved to be less effective on highly constrained instances.

Weng and Asmuni (2013) implemented a Particle Swarm Optimization algorithm. The Artificial Bee Colony (ABC) was extended and tested against the more enhanced Global Best Concept - Artificial Bee Colony (GBABC). The strength of an ABC implementation is that it has a very high exploration rate. This however contradicts the ability to exploit potential good solutions. Therefore this algorithm is enhanced by a local search to improve its exploitation ability, resulting in the GBABC algorithm. Both these algorithms were tested on real

life data, where the GBABC algorithm outperformed the ABC algorithm.

Mumford (2008) used a different approach. The main focus of this paper is on the calculation of the minimum amount of timeslots needed to accommodate all the exams. The duration of exams are all the same and it is known which student has which exams. The spread of the exams should be as evenly as possible for every student. They developed a genetic local search to optimize the timetable by maximizing the multi-objective objective function. Amongst others they use the Kempe chain exchange operator: find two exams that conflict with each other, create a connected subgraph and apply relabeling in this subgraph. This, however, does not always lead to improvements. Results from this approach compare reasonably well with results from other recent studies. However, this approach really stands out from the other algorithms for the more heavily constrained problems.

1.4.2 Student-centered timetabling

Another school-timetabling variant is the Student-centered timetabling. In this problem students pick classes which they want to follow. In most cases a teacher can teach different classes, so the scheduler has to determine the following aspects for each class:

- Which teacher will give the class
- Which students will be part of the class
- In which room will the class be held
- At what time will the class be held

Just as the exam timetabling problem, the student-centered timetabling problem takes the individual timetables of the students and teachers into account. This results in more constraints than the Curriculum-based timetabling problem, which makes it harder to find a feasible solution.

Three different metaheuristics are tested in Colorni et al. (1997): Simulated Annealing, Genetic Algorithm and Tabu Search. All three metaheuristics were tested with a real life example: an Italian High School. According to the objective function the Tabu Search produced the best timetables, followed by the timetables of the Genetic Algorithm and those created by the Simulated Annealing algorithm. However, the end-users preferred the use of the Genetic Algorithm since this algorithm produces multiple solutions so that the end-user can easily pick the best suitable timetable. This indicates that it is hard to transform the hard and soft constraints into an appropriate objective function. The main focus within these algorithms is on the ability to recover an infeasible solution to a feasible solution. Potential infeasibilities are also penalized in the objective function to reduce the chance that a solution has to be recovered.

Also for the student-centered timetabling problem a Particle Swarm Optimization algorithm is developed. Socha et al. (2002) mimicked an ant colony to explore the solution-space. This approach led to good timetables, which is remarkable since no problem-specific operators were implemented.

For a high school in Greece a system is developed based on an ILP model, as is described in Birbas et al. (1997). This model is plugged into CPLEX where the Mixed Integer Programming solver is used. Many of the hard constraints are converted to soft constraints where infeasible solutions get a high penalty. This implementation resulted in *optimal* timetables, where most algorithms in similar studies produce sub-optimal timetables.

In most schools students in the last stage of their study are allowed to pick a couple of lectures that they want to follow. Based on the choices of the students a timetable will be made, with as few as possible clashes. Barham and Westwood (1978) decided to create groups of lectures that can be held concurrently by taking the option with the most participants and the option with the second most participants that can be held *at the same time*, and combine them into one group. This process is repeated to fill the groups. Barham and Westwood (1978) introduced two strategies which create timetables based on different objective function; the scheduler can then pick the most appropriate solution to work with.

The method described in Wood and Whitaker (1998) uses a phased ILP algorithm. It is customary for the particular secondary school in New Zealand that students are not completely free to choose their classes; they have to pick one class out of each stack, where a stack consists of multiple classes. Since each student only chooses one class out of each stack, the scheduler can safely schedule all the classes of a stack at the same time without introducing problems for the students. This stack-system is very similar to the timeslot model of Utrecht University. In the paper of Wood and Whitaker (1998) a model is introduced where students can first pick the classes they want to follow. After this the classes are put into stacks so that there won't be any clashes. This is done with an implementation of Simulated Annealing. Next, the stacks are used to create a timetable. The creation of this timetable is a relative small problem so a backtracking algorithm could be used here. As a result, this method produces a timetable within hours, where it took manually timetabling several weeks to create a timetable.

1.4.3 Curriculum-based timetabling

As the problem grows in size the schedulers are unable to take the individual timetables of the students into account. Therefore a third category of school timetabling problems is introduced: the Curriculum-based timetabling problem. The main difficulty in this category of timetabling problems mostly lies in the size of the problem. Where secondary and high schools only have a few hundred students the universities often have thousands of students, resulting in a massive search-space. The so-called student-clashes (a student has two classes at the same time) are avoided by using a similar system as described in Wood and Whitaker (1998) to reduce the search-space. In this research area significantly more research has been done. Cooper and Kingston (n.d.) proved that the complexity of these kind of timetabling construction problems is NP-Complete.

Duong and Dien (2008) defined a swapping method where the starting times of two meetings are being swapped. The objective function is defined by the preferences of the teachers, so the swapping of two meetings might result in a better timetable. As an extension of this method they also introduced an operator which swaps three meetings. However, this operator is very destructive and should not be used too often. These operators are similar to 2-opt and 3-opt, two well known operators commonly used in the traveling salesman problem. The combination of these swap-operators with the Tabu-search result in better timetables in less computation time.

Several simple operators were used in the Simulated Annealing described in Kalender et al. (2012). At each iteration one of the operators is picked randomly, but the chance of each operator to be picked was made adaptive. This leads to a higher usage of successful operators, which in turn leads to significantly better timetables.

At the UPAEP University in Mexico meetings have time windows in which they can be

scheduled. These windows are defined by the availability of the teachers. Besides the time windows there are also other demands that are very constraining. This results in a timetabling problem that is heavily constrained. Sánchez-Partida et al. (2013) modeled the problem as an ILP problem and solved it efficiently with the Lingo 13 solver.

Doulaty et al. (2013) implemented a genetic algorithm to create timetables for the Computer Science and Mathematical Science departments. This was a three-days job when it was done by hand, but they managed to generate a timetable automatically in 15 minutes. Although the timetabling problem at Utrecht University is much bigger, this 15 minutes-runtime is a good guideline for the duration of the algorithm.

Müller (2005) had as their main goal to produce timetables which are fully accepted by the end-users. Therefore all the constraints have to be implemented, including many small constraints that are easily overlooked when you are not a scheduler. The timetables also have to be *maintained* by the users, so when an adjustment has to be applied the algorithm has to come up with an adjusted timetable which includes the adjustment and has barely any changes in comparison with the original timetable. To find the appropriate minimal adjustment they convert the problem in a minimum perturbation problem. The total problem was divided into sub-problems in order to reduce the size of the problems:

- Lectures that can only be held in large rooms
- Averagely sized rooms
- Computerlab lectures

This approach has been successfully applied at Purdue University.

The objective function Pongcharoen et al. (2007) covers many different aspects, including student movement, fragmentation in the timetables of both students and teachers (the less gaps in an individual timetable the better), and time preferences of teachers. In this research a Genetic Algorithm is compared with Simulated Annealing. The Genetic Algorithm will have difficulties finding a good solution when an infeasible solution receives a high penalty or gets discarded; this way complete populations might be useless when the problem is highly constrained. However, in this research infeasible solutions are getting *repaired* instead of discarded. The repairing function consists of moving lectures around. The use of this repairing function should be minimized since it costs time which could also be spend on searching for a better solution. Therefore they successfully introduced a clash-detector to detect potential clashes to prevent the clashes.

Lukáč (2013) implemented the timetabling problem of the University of Masaryk in UniTime[2]. They added the feature to deal with the lunch breaks of teachers and students. They also added alternative time-windows for the teachers so that the search-space is larger, which was needed since this situation was heavily constrained. Schedulers of the University of Masaryk had a hard time to find a timetable that covered all these constraints. By upgrading UniTime to handle these constraints the researchers were able to produce feasible solutions using this software package.

maw Chen and fang Shih (2013) implemented a Particle Swarm Optimization algorithm to solve a (fictive and relatively small) timetabling situation. To reduce premature convergence a local search mechanism was introduced, which proved to be vital: the algorithm performed worse without the local search.

[2]Opensource timetabling software: http://www.unitime.org/

El-Jazzar (2012) used a three-phase approach to create timetables:

1. Find a feasible solution with Simulated Annealing according to the hard constraints.
2. Use the feasible solution to generate a population of solutions for the Scatter Search.
3. Tune the optimized timetables with a small local search algorithm.

The focus of this research was on the second stage of the approach. This resulted in better timetables than the winner of the International Timetabling Competition in 2007.

1.5 Structure

In section 2 we analyze the given data and describe how we shaped it in usable data to give more insight in the structure of the data. After that we describe the model that follows from the data analysis; this is described in section 3. The concrete implementation of this mathematical model is described in section 4. Then several experiments are run with the implementation to see the effects of the different aspects of the algorithm. This paper concludes by giving a summary of the findings.

2 Data purification

A lot of information is needed for a university to be able to facilitate the courses for students. The goal of this section of the paper is to extract structure from this enormous amount of data. This is done by purifying the data: filtering off unneeded information to get a better overview of the problem at hand. Utrecht University uses *Osiris*[3] to cover all the administrative aspects and to store all the data at a central point. Amongst others, this system stores information about students and employees of Utrecht University, but also course information. A course has a *course code*; these codes are in most cases a conjunction of faculty name and course name. The course code is a unique identifier of a certain course; every time the course is given the same course code should be used. A course can be given multiple times, spread over different years, or even multiple times in the same academic year. Every time a course is given the course is *instantiated*, giving the course a starting period and the academic year it is given in. The combination of course code, starting period and academic year is a unique identifier for an instantiated course, see figure 1 for an example. From now on an instantiated course will simply be called a course, since only the instantiated courses are relevant for roomscheduling.

	code	startingperiod	year
1	OGMV05015	1	2011
2	OGMV05015	2	2011
3	OGMV05015	3	2011
4	OGMV05015	4	2011

Figure 1: A real life example of a course being instantiated 4 times over a year.

Utrecht University has divided the academic year into four periods. The possible starting periods for a course are as follows:

- Period 1: week 36 until 45
- Period 2: week 46 until 5
- Period 3: week 6 until 16
- Period 4: week 17 until 27

Next to these four periods courses can also have "Year" as a starting period. These courses are always projects; they typically do not need rooms for meetings, because these projects are mostly individual.

Besides final examweeks, there are also *re-examweeks*. The weeknumber of the period in which these re-exams take place is determined per department. Just like the normal examweeks, the re-examweeks of a department usually do not have lectures (of the same department) scheduled in these weeks.

All in all there are only a few dates on which there are absolutely no lectures scheduled. Unfortunately, this means that the algorithm cannot exclude certain weeks to restrain the search space.

The data we acquired from the Osiris database contained different appearances for the denotation of a period. One would assume only period 1, 2, 3, 4 and Year would be used in the

[3]The Student Administration System Osiris: http://www.osiris4u.nl/

data, but unfortunately this is not the case. In table 1 the mapping is shown between the different appearances in the data of Osiris and the used periodnames in the algorithm.

Period appearance	Periodname
1	1
BLOK1-S	1
SEM1	1
GS1	1
2	2
BLOK2-S	2
GS2	2
3	3
SEM2	3
BLOK3-S	3
BLOK3	3
4	4
BLOK4-S	4

Table 1: Mapping between period appearances in the data and the periods used in the algorithm.

2.1 Timeslots

Utrecht University developed the *timeslot model*. This model structures courses by grouping the courses into (roughly) four different groups: every course gets a timeslot (A, B, C or D)[4]. The timeslot determines the times at which a lecture of a course can be scheduled. For instance: the course INFOIMP has timeslot D, which means that the lectures will only take place on Wednesday afternoon, Friday morning or Friday afternoon, according to figure 2. The advantage of this timeslot model from a student point of view is that it is easier to see

timeslots	ma	di	wo	do	vr
09.00-12.45	A1+2	B1+2	A4+5	C6+7	D4+5
13.15-17.00	C1+2	C4+5	D1+2	B3+4	D6+7
17.00-18.45	C3	A3	D3	B5	

Figure 2: The timeslot model used by Utrecht University.

which courses overlap: a student can follow another course, next to INFOIMP, if its timeslot is not D. It is guaranteed that the lectures of the different courses will not overlap. The introduction of the timeslot-model saves a lot of puzzling for the student to figure out which courses he can take without any clashes in his timetable. This also helps in the determination of which teacher will teach which course; teachers are only human (even though it may seem otherwise) and cannot be at two places at the same time. One of the cornerstones of Utrecht University is the versatility of its courses. This has as advantage that, for instance, a mathematician can follow a couple of psychology courses. When he follows a certain amount of psychology courses he will get an extra annotation on his degree. The timeslot-model also

[4]The special timeslot E will be explained later.

helps in accommodating this feature by structuring the timespans of the lectures of a course across the whole university.

In the Osiris database the timeslot in which a course should take place can be notated in various ways. Just as there were numerous ways to define the same period (table 1) there are also multiple ways to define a timeslot. The mapping table of the timeslots is stated in table 2.

Timeslot appearance	Timeslot
A	A
B	B
C	C
D	D
E	E
-	A, B, C, D, E
UCU-T1	A
UCU-T2	A
UCU-B2	B

Table 2: Mapping between timeslot appearances in the data and the periods used in the algorithm.

The E-timeslot is a special timeslot, which is introduced for part-time students. Lectures with this timeslot are scheduled in the evening, since most of these part-time students have jobs to attend to.

When a course does not have a timeslot, indicated by '-', it is assumed that the timeslot in which the course should be scheduled in is not relevant. This might be the case when a course is so extensive that it is not possible for a student to do another course next to it. Another explanation might be that the course is a project and therefore does not need strictly scheduled lectures. In both cases the course is not restricted to a certain timeslot and can therefore be freely scheduled.

The appearances starting with 'UCU' originate from the University College Utrecht, which is a special department of Utrecht University. The timeslots used at the University College can roughly be mapped to the normal timeslots used at Utrecht University.

A course can have multiple timeslots, increasing the time-window in which a meeting can be scheduled in. To keep the timeslot calculations fast and simple, we have used bit values to represent the timeslots; every bit value represents a different combination of timeslots. This way the timeslot calculations were on bit-level and are therefore very fast.

2.2 Lecture types

As is customary in the educational world, the students at Utrecht University get different types of lectures so that they get familiar with the subjects as effectively as possible; different methods of presentation lead to better insights in the subjects. There is often a big gap between theory and practice, which is also captured by the different lecture types. Another reason to accommodate different types of lectures is because some topics are practically impossible to present to the students with normal lectures, for instance autopsy for medical students.

Lecture types in Osiris database	aggregated lecture type	appearances
BASISCOL	hoorcollege	164
COACHGROEP	werkgroep	228
COMP PRACT	practicum	298
DEMO	werkcollege	3
DISC COL	seminar	12
DISC GRP	werkgroep	10
GASTCOL	hoorcollege	16
GROEP OPDR	werkgroep	132
GUEST LECT	hoorcollege	3
HOORCOL	hoorcollege	2.961
HOORCOL-1	hoorcollege	157
HOORCOL-2	hoorcollege	47
HOORCOL-3	hoorcollege	12
HOORCOL-4	hoorcollege	15
HW COL	hoorcollege	1.520
HW COL-1	hoorcollege	61
HW COL-2	hoorcollege	57
INDIV	werkgroep	50
INDIV OPDR	werkgroep	6
INST	seminar	205
INSTBK-V1	seminar	190
INSTBK-V2	seminar	2
INSTBK-V3	seminar	109
INSTBK-V4	seminar	161
INSTBK-V5	seminar	19
INSTBK-V6	seminar	7
INSTR COL	hoorcollege	86
INTERVISIE	seminar	14
LECTURE	hoorcollege	49
MASTERCLASS	seminar	9
NVT	seminar	1
O & O	seminar	1
ONDERZOEK	werkgroep	5
PRACT	practicum	1.384
PRACT-1	practicum	219
PRESENT	seminar	42
PROJECT	werkgroep	136
PROJECTGR	werkgroep	18
RESP COL	hoorcollege	9
SEMINAR	seminar	504
SEMINAR-1	seminar	1
SEMINAR-2	seminar	8
STAGE	werkgroep	8
STAGEBIJEENK	werkgroep	21
TALENPR	practicum	91
TALENPR-2	practicum	44
THEMA	werkgroep	295
TUTORIAL	practicum	250
VIDEO/FILM	seminar	33
WERKCOL	werkcollege	4.909
WERKCOL-1	werkcollege	297
WERKCOL-2	werkcollege	124
WERKCOL-3	werkcollege	39
WERKCOL-4	werkcollege	7
WERKGROEP	werkgroep	4.427
WG EXTENS	werkgroep	6
WG INTENS	werkgroep	86
WORKSHOP	werkcollege	40

Table 3: Mapping of the different appearances of lecturetypes in the Osiris database to the aggregated lecture types used in the algorithm. The number of appearances is denoted in the last column. 4.407 Meetings have a type other than 'WERKCOL', 'WERKGROEP', 'HOORCOL', 'HW COL' and 'PRACT'.

There are various ways defined in the Osiris Database to indicate what type of lecture a meeting has. As can be seen in table 3 there are 58 different types of lectures. There are a few types that are used often ('WERKCOL', 'WERKGROEP', 'HOORCOL', 'HW COL' and 'PRACT'), but most of the types are used significantly less often. Since the lecture types of these less frequent types are used as a type for a *substantial* part of the meetings, we cannot simply omit these meetings. Therefore, again, a mapping has to be defined to map each lecture type in the Osiris database to an aggregated lecture type. Due to this mapping function the algorithm only has to focus on six different types of lectures. The lecture types originate from different faculties of Utrecht University. To have a perfect mapping would cost a lot of time and lies beyond the scope of this project. However, additions and changes can easily be applied.

The result of the mapping function on the available data from Osiris can be seen in table 4. The following six types of lectures are defined:

- Werkgroep (translation: workgroup)
- Werkcollege (translation: practical lecture)
- Hoorcollege (translation: normal lecture)
- Practicum (translation: practicum)
- Computerpracticum (translation: computer practicum)
- Seminar (translation: seminar)

Aggregated lecture type	Number of appearances
werkgroep	5.428
werkcollege	5.419
hoorcollege	5.157
practicum	1.988
seminar	1.318
computerpracticum	298

Table 4: The number of appearances of the aggregated meeting types.

There is no clear difference between a workgroup and a practical lecture, as each faculty has its own interpretation. In both lecture types the students have to do homework assignments in class to get a better understanding of the subjects, for instance making exercises. The students are assigned to different groups, so that each group is relative small and each student receives enough attention from the lecture leader. It is often the case that multiple groups have their lectures at the same time (only the rooms differ).

A normal lecture is a classic lecture in which the students listen to the teacher's explanation. In contrast to workgroups and practical lectures these lectures have mostly a one-way interaction from the teacher to the students, with the exception of a sporadic question from the students. Because of this one-way interaction the maximum capacity of a normal lecture can be a lot higher than that of a workgroup or practical lecture. This has as a direct consequence that the room should be bigger to accommodate all the students.

During a practicum the students will be working on a practical assignment. The definition of a practicum depends on the faculty of the course; most courses (like Chemistry and Medicine) have specific needs for a practical assignment. An exception is Computer Science: a Computer Science practicum needs a room with computers, where other courses would define such a

Faculty	Description
BETA	Beta courses
DGK	'Diergeneeskunde', veterinary Medicine. These courses often need special rooms.
GEO	'Geowetenschappen', geosciences
GNK	'Geneeskunde', Medicine. Just as Veterinary Medicine, Medicine needs special rooms for their education.
GW	'Geesteswetenschappen', humanities
IVLOS	Teacher training
OC-UMC	Department of University Medical Center Utrecht
REBO	'Recht, Economie, Bestuur en Organisatie', law, economy, management and organisation
SW	'Sociale wetenschappen', Social studies
UC	University College
UU	Used for non faculty specific meetings

Table 5: Different faculties of Utrecht University.

practicum as a 'computer practicum'. The discrimination between an ordinary practicum and computer practicum is made because the rooms needed for these types of practica differ a lot.

The last lecture type is the seminar. These are miscellaneous lectures which cannot be put in any other lecture type. It ranges from workshops to discussion groups. Mostly the number of participants is relatively low.

2.3 Roomsets

Utrecht University has numerous rooms at his disposal, scattered across Utrecht (see figure 3 for a small map). Utrecht University divided the rooms in various sets, taking the lecture type into account. If there are no restrictions on the room in which a lecture type can be scheduled (e.g. a workgroup can be scheduled in rooms X, Y or Z) a practical problem will arise: people do not like to go to unfamiliar places. To accommodate this practical problem the *faculty* of a meeting also has to be taken into account, since students of the same faculty often use the same buildings for their education. In table 5 the abbreviations used in the Osiris database are listed.

Now that when the faculties are known, a *roomset* can be defined:

Roomset A *roomset* is a set of rooms defined by the combination of lecture type and faculty.

Since there are 6 lecture types and 11 different faculties there are 66 roomsets. A room can be in multiple roomsets. A roomset can also be empty.

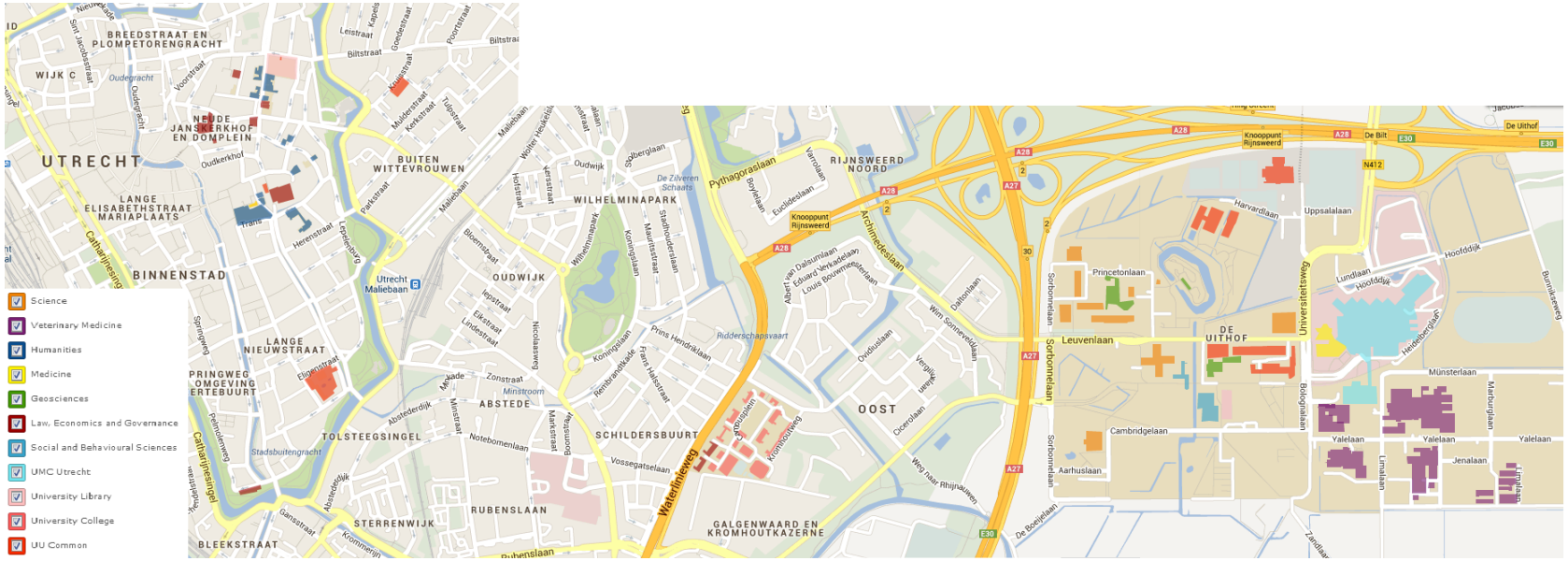

Figure 3: A part of the map of Utrecht. The different colors indicate the faculty to which it belongs.

Because of defining roomsets in this way, every faculty + lecture type combination has a roomset, which consists of rooms that the students of that faculty are familiar with. The assignment of a roomset for every faculty + lecture type pair narrows down the search process tremendously, since the number of possible rooms where a lecture could be scheduled in is greatly reduced. Note that the roomset is for every lecture of the same faculty and lecture type the same, regardless of the number of participants of the lecture; checking if the number of participants of a lecture does not exceed the size of the room is done in a later stage.

The timetable of academic year 2011-2012 is used to determine the roomsets: the mappings of previous sections are applied and the scheduled room of every meeting in the timetable is added to the appropriate roomset (lecture type + faculty combination). By using this real life data the roomsets become less error prone than when the roomsets are composed by us. This also makes sure that every meeting has a non-empty roomset. If the roomsets are composed by employees of Utrecht University the roomsets will be the best reflection of the real life roomsets; this procedure is however very time-consuming and therefore omitted. The roomsets currently used by Syllabus Plus cannot be used, because these roomsets are defined differently, namely by equipment-constraints and distance-constraints. Also, the roomsets of Syllabus Plus are not as fixed as our roomsets; the sets of Syllabus are expanded regularly by dropping constraints to find a valid timetable. The roomsets constructed by our definition are more stable and should provide a realistic representation.

In table 6 the roomsets are listed together with the number of rooms in it. Since the roomsets are based on the actual meetings we have to schedule, we can conclude that multiple combinations of faculty + lecture type do not occur in the data. For instance: the faculty Geosciences only has 3 out of 6 lecture types.

2.4 Miscellaneous

In this section we will discuss miscellaneous findings from the data analysis and perform a checkup of assumptions for consistency. This data analysis consists of the examination of the meetings table, composed by applying the various mappings on an Osiris table. Every meeting in this table (18.450 rows) should be scheduled in the timetable. There are however some irregularities / errors in the data. Some meetings were too ambiguous because of the errors, so they were omitted; other errors could be fixed:

- A meeting has a *period* and a *starting period* field. A course can last longer than one period: the starting period represents the first period of the course, while the normal period field represents the actual period of a meeting. A portion of the meetings where $period \neq startingperiod$ has a *starting period* which is greater than *period*. This would mean that a meeting of a course is on an *earlier* date than the beginning of the starting period, which is not consistent; these meetings were therefore dropped.

- There are two ways to determine the number of participants of a course. The first method involves the table which keeps track of the courses that each student follows. With this table the exact number of participants can be calculated. This information is however not available at the time that the meetings are planned, so this information cannot be used.

  In the course information table of the Osiris database there is a field called *maximum participant*. This should be the ultimate maximum number of participants that could follow a course at the same time. This is in 11.868 of 18.450 entries not the case: the number of participants of these meetings exceeds the maximum number of participants.

Faculty	Lecture type	Number of rooms
BETA	computerpracticum	8
BETA	hoorcollege	52
BETA	practicum	16
BETA	seminar	17
BETA	werkcollege	40
BETA	werkgroep	11
GEO	computerpracticum	6
GEO	hoorcollege	11
GEO	werkcollege	13
GW	computerpracticum	8
GW	hoorcollege	79
GW	practicum	37
GW	seminar	33
GW	werkcollege	78
GW	werkgroep	19
IVLOS	seminar	46
IVLOS	werkcollege	2
IVLOS	werkgroep	19
REBO	computerpracticum	1
REBO	hoorcollege	39
REBO	practicum	18
REBO	seminar	13
REBO	werkcollege	35
REBO	werkgroep	49
SW	computerpracticum	11
SW	hoorcollege	51
SW	practicum	62
SW	seminar	37
SW	werkcollege	59
SW	werkgroep	70

Table 6: The non-empty roomsets with the number of rooms of that roomset.

An explanation of this could be that the calculation of the number of participants does not handle the reincarnations properly: a course can get reincarnated multiple times in a year, meaning that a course can, for example, be given four times in a year (see figure 1 for a concrete example). It seems that the number of participants of all these incarnated courses are added together. This is easily fixed by dividing the amount of participants by the amount of reincarnations. There are however not many courses that get reincarnated.

There are 2.219 rows where the maximum number of participants was undefined; in these cases the number of participants is calculated by an aggregation-function applied on the student-enrollment table. In practice, the student-enrollment is not known at the moment the timetable has to be created, so using this aggregation-function should not be used too often since it provides information for the algorithm that is not known for the schedulers (when the timetable has to be created).

- The number of rows of the meetings-table is not equal to the number of meetings that has to be scheduled. This has to do with two fields: *week_from* and *week_till*. Week_from determines the first week in which the meeting has to be scheduled; week_till determines the last week in which the meeting has to be scheduled. This way one could make a regular meeting for a whole period with only one entry. There are however 8.182 rows

Difference in weeks	Number of appearances
0	8.182
1	1.621
2	2.040
3	1.801
4	1.466
5	699
6	307
7	793
8	1.016
9	507
>9	18

Table 7: The number of appearances of the number of weeks. The amount of entries with 0 difference in weeks is enormous; this could be reduced by aggregating the entries that are incorrect.

where $week\_from = week\_till$. These entries might refer to the incidental lectures. However, many lectures use one entry for every week. So if a lecture should be scheduled from week 1 till 8 there are 8 entries, each with $week\_from = week\_till$. This could easily be compressed to one entry with: $week\_from = 1$ and $week\_till = 8$.

In table 7 an overview is given of the differences in weeks ($week\_till - week\_from$). Note that this does not prevent the algorithm from working properly; it merely shows that the input data is not flawless.

- There are almost 10.063 rows that have week_from or week_till greater than 12. This would indicate that the week_from and week_till fields are actual weeknumbers. However, a portion of the rows do not meet this interpretation.
  - There are 935 rows with their week between 1 and 5, but do not have period 2 as their period.
  - There are 2.564 rows with their week between 6 and 12, but do not have period 3 as their period.
  Note that week 1 until 5 belong to period 2 and week 6 until 12 belong to period 3. This means that either the weeks ór the period is filled in incorrectly. To overcome this problem it is assumed that the number of weeks is filled in correctly in ambiguous cases.

- Each row in the table has a field called *frequency*. This determines how often a lecture should be scheduled per week. However, this field is *always* '1W' (once in a week), despite the fact that most of the courses have *at least* two normal lectures in a week. If it would be used properly it would result in a more compact and more easily understandable table. Again, this is merely a point of improvement to keep the database manageable.

- Since we use data from a previous year, the starting time and ending time is known. However, these are only used to calculate the duration of a meeting; they are not used, for instance, to produce an initial timetable.

- A course can have multiple groups, so that the different groups can have a workgroup at the same time. The groups are numbered and stored in the Osiris database. However, it is not uncommon that there are gaps in the counting. This means that the highest groupnumber may not be equal to the number of groups. For instance, a course can have groups with the following groupnumbers: 1, 2, 4 and 20; there are only 4 different groups instead of 20. This is an irregularity, but does not influence the algorithm.

3 Model

The first step in solving this timetabling problem of Utrecht University is translating the problem to a mathematical model, which is done in this section. Next, the objective function to determine the quality of a timetable is explained.

3.1 Problem description

The model for this problem is not straightforward because of some additional constraints, but in the end every meeting m of meetings M should be scheduled exactly once in a suitable room r at an appropriate time t subject to the following constraints:

$$\forall m \in M : r \in m_R :$$

Here m_R is the roomset of meeting m, defined by the lecture type m_{type} and faculty $m_{faculty}$ of m.

$$\forall m \in M : r_{capacity} \geq m_{participants} :$$

This ensures that there is enough space in room r for all the students $m_{participants}$.

$$\forall m \in M : t_{quarters} \subseteq\in m_T :$$

Here $t_{quarters} := \{t_i : t_{start} \leq i < t_{start} + m_{duration}\}$, defining the set of quarters of an hour that meeting m occupies. m_T is the total set of available quarters of an hour according to the timeslot. Note that *quarters of an hour* could easily be replaced by any other measurement of time. Here, quarters of an hour is chosen as the time unit; this will be explained in further detail in section 4.2.

$$\forall m \in M : t_{week} = m_{week} :$$

Here m_{week} is the week in which the meeting should be scheduled; this should be equal to the week-component of time t.

Also, there can only be maximal one meeting scheduled in room r at time t.

This description is the base of the problem. However, there are additional features that complicate the problem:

- *Regularity*: Having regular lectures throughout a period. This will be explained in section 3.1.1.
- *Dependency*: A lecture can only be scheduled *after* another lecture. Dependency between lectures can only be defined on lectures that are part of the same course. A more detailed definition can be found in section 3.1.2.

The data has to have more structure to accommodate these features; therefore an aggregation-function will be applied, which assigns a meeting m to a *Global Lecture*.

Global Lecture A Global Lecture is a collective term for a collection of meetings. These meetings only differ in m_{week}, while all having the same values for the following key-properties (amongst other less crucial properties):

- Incarnated course
- Lecture type (as mentioned in section 2.2)

- Group number
- Number of participants

A Global Lecture has a constructed property $GlobalLecture_{weeks}$ which defines in which week the Global Lecture is held. This set of weeknumbers is constructed from the collection of meetings $GlobalLecture_{meetings}$ which are the meetings that are mapped to the Global Lecture.

Course	Lecture type	Weeks	Participants	Groupnumber	Remarks
S & T	Normal lecture	1-8	80	1	First weekly normal lecture for all participants
S & T	Normal lecture	1-8	80	1	Second weekly normal lecture for all participants
S & T	Practical Lecture	1-8	40	1	First weekly practical lecture for group 1
S & T	Practical Lecture	1-8	40	1	Second weekly practical lecture for group 1
S & T	Practical Lecture	1-8	40	2	First weekly practical lecture for group 2
S & T	Practical Lecture	1-8	40	2	Second weekly practical lecture for group 2
S & T	Seminar	9	40	1	Seminar for group 1 in the final week
S & T	Seminar	9	40	2	Seminar for group 2 in the final week

Table 8: The table of the meetings of the course Scheduling & Timetabling.

Example In this concrete example we look into the lecture-structure of the course *Scheduling & Timetabling*; the meetings for this course are listed in table 8.

Each student will have the following lectures:

- For the first eight weeks: two normal lectures per week
- For the first eight weeks: two practical lectures per week.
- For the last week (week 9): a seminar.

The normal lectures are for all the participants of the course, so two Global Lectures are sufficient to express the normal lectures of this course. However, for the practical lectures the group is split in half to make sure everyone can ask their questions and receive enough attention from the teacher. Therefore the number of Global Lectures for the practical lectures is doubled, since there are two groups for each practical lecture. The seminar of the course is also divided into two groups, so this lecture needs two Global Lectures to express the situation appropriately as well. An overview of the Global Lectures of this example is shown in figure 4.

Aggregating the meetings into a Global Lecture will add an intermediate level in the model, providing handles for solving the problem. The introduction of the Global Lectures allows us to define the *regularity*- and *dependency*-features.

3.1.1 Regularity

As is common for educational timetables, lectures are related and connected to other lectures: it is highly preferable by teachers and students that regular lectures are repeated every week, so that the lectures are at the same time and in the same room for every week. This makes it easier for the teachers and students to plan other activities next to the lectures; now they do not have to check the timetable to see when and where their class is, making a timetable a lot more convenient.

The introduction of Global Lectures allows us to discriminate lectures that are held on a regular base from lectures that are only held occasionally:

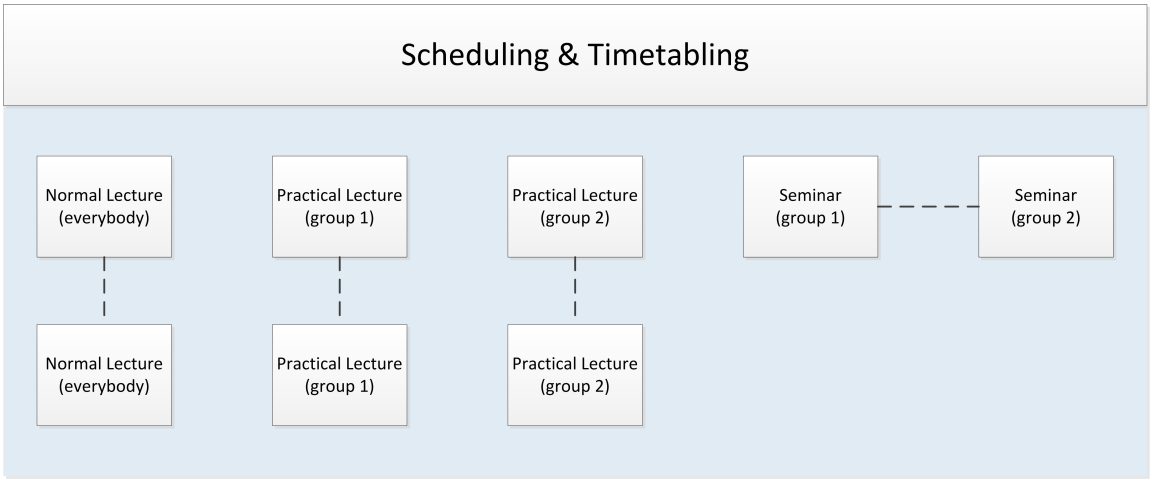

Figure 4: The Global Lecture structure of the course Scheduling & Timetabling. The course consists of two weekly normal lectures and two weekly practical lectures (for two different groups). In the last week of the course there will be a seminar for both groups. This results in a total of eight Global Lectures. The dashed lines between the Global Lectures indicate correlation between the lectures: two connected Global Lectures should not be given at the same time. Extra dependencies between Global Lectures can be defined to further specify precedence-rules, including time-windows. These dependencies will be further explained in section 3.1.2.

- *Regular Lectures*: $L_{regular}$ This is a set of Global Lectures where $GlobalLecture_{weeks}$ is relatively large, meaning that the Global Lecture is held in many weeks. In the example of Scheduling & Timetabling (figure 4) the normal lectures and practical lectures are typically regular lectures; these lectures are held weekly.
- *Incidental Lectures*: $L_{incidental}$ This is a set of Global Lectures where $GlobalLecture_{weeks}$ is relatively small, meaning that the Global Lecture is held in only a few weeks. In the example of figure 4 the seminars are typical incidental lectures; these are only held in the last week.

Note that $L_{regular} \cup L_{incidental}$ is the total set of Global Lectures, and $L_{regular} \cap L_{incidental} = \emptyset$ (disjoint).

Now that the Global Lectures are split into two distinct sets, we can define the regularity-constraint: every meeting of the same *regular* Global Lecture should be held at the same time in the week (for instance: monday 11:00) and should be given in the same room (for instance: BBL-007).

By introducing this constraint to the base problem we now have a way to model the desired regularity in a timetable.

3.1.2 Dependency

One of the disadvantages of the current system is the lack of *order* of lectures. It is not possible to indicate that lecture A has to be held before lecture B. Most courses prefer to start the week with a normal lecture instead of a practical session, which means that the normal lecture has to be held *before* a practical session. Some courses want to make a whole sequence of lectures, for instance:

1. *Normal college* to introduce the student to the subject matter
2. *Practical session* where the students have to do field work or a practical assignment
3. *Computer practicum* to examine the results of the practical session
4. *Seminar* to discuss or present the findings of the day

These lectures should follow up each other seamlessly. To model this feature a global approach like 'just schedule lecture B somewhere after lecture A' will not do the trick; a more complex model has to be applied here.

Dependent Global Lecture B can be dependent on Global Lecture A, meaning that B has to be scheduled *after* A. Besides a pointer to the previous lecture, the following two properties should also be defined to completely create a dependency:

- *Minimum time*, defining the minimum amount of time between the start of lecture A and the start of lecture B.
- *Maximum time*, defining the maximum amount of time between the start of lecture A and the start of lecture B.

The example of the course 'Scheduling & Timetabling' is extended with dependencies in figure 5.

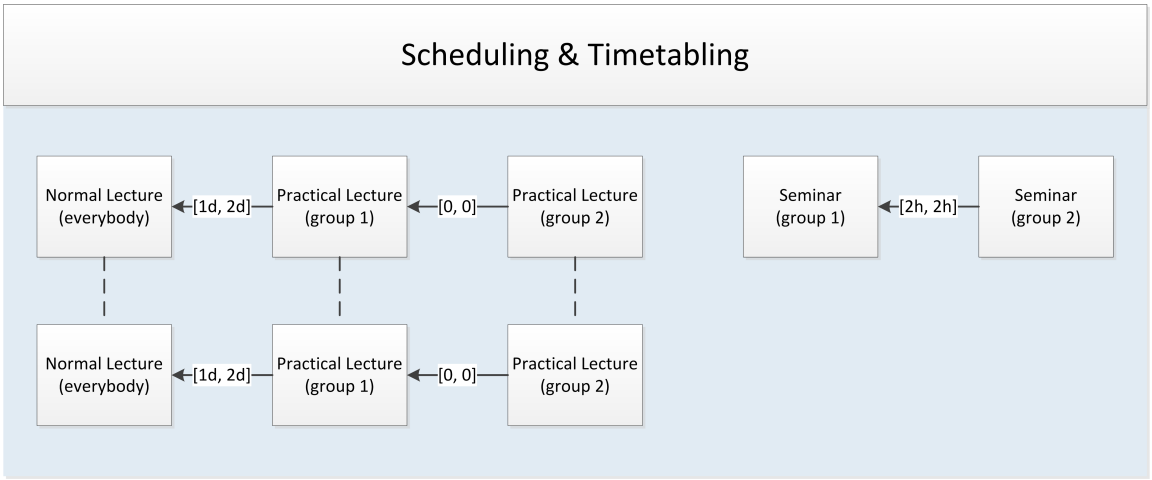

Figure 5: The example of figure 4 is extended by adding dependencies between Global Lectures. An arrow from lecture A to lecture B indicates that lecture A has to be scheduled after lecture B; the time window is denoted between brackets. Explanation of the dependencies: a practical lecture has to be scheduled between 1 day and 2 days *after* a normal lecture. The practical lectures of both groups should be held parallel to each other, so the time window between both groups of the practical lectures is set to [0, 0]. The second seminar should be scheduled 2 hours after the start of the first seminar. Since the duration of a seminar is two hours, this time window indicates that the two seminars should follow each other seamlessly. Note that the time windows are expressed in quarters of an hour in the actual implementation.

Setting up the dependencies in combination with the *time window* [Minimum, Maximum] in this manner provides us the following advantages:

- The algorithm can be given more freedom in scheduling lecture B by creating a large time window. This gives the algorithm more options to choose from, which ultimately leads to a better timetable.

- When needed, lectures can be chained seamlessly after each other and thereby allowing the scheduler to model the previously mentioned chain of lectures.
- Some lectures have too many participants to fit in any room of Utrecht University. A solution to this problem is to divide the total group in multiple smaller groups. The teacher will do his lecture live in one room, while the other rooms follow him through a camera and beamer arrangement. All the groups should start at the same time, which is possible by creating dependencies with time window [0,0]. This enforces the algorithm to schedule all the groups at the same moment.

For consistency, a *regular* Global Lecture can only be dependent on another *regular* Global Lecture. Letting a *regular* Global Lecture depend on an *incidental* Global Lecture would be making no sense, since this would imply that the *regular* Global Lecture can only be scheduled in the week of the *incidental* Global Lecture. However, an *incidental* Global Lecture *can* be dependent on a *regular* Global Lecture.

In this thesis the dependencies are hard constraints; the addition of the minimum and maximum time should provide enough flexibility to handle this constraint properly. However, one could choose to relax the dependency constraint by penalizing the violation of the time-window: the more time between the starting time of the lecture and one of the boundaries of the time-window, the higher the penalty.

3.2 Objective function

In the search-process timetables are compared to each other to determine which timetable is better. Many optimization problems have tangible objectives like maximizing profit or minimizing the amount of space. Since timetables do not have a direct tangible objective, it is harder to define an objective function for timetabling problems. The objective function needs to express soft preferences that are difficult to express in a scoring function. The different parts of the objective function is explained in this section piece by piece.

The objective function created for this timetabling problem mainly focuses on the preferences of the students and teachers. The greatest desire of the students and teachers is that lectures follow each other, so that they do not have to wait for their next lecture. For instance: having a lecture at 9.00-11.00 whilst the next lecture is at 15.00-17.00 is not preferable, since the student has nothing to do from 11.00 till 15.00. Most of the students have a traveling time of 45+ minutes so it is not efficient to go back home, forcing them to stay in Utrecht.

Theoretically this could be solved perfectly, since it is known for every student which lectures he follows. This can then be included in the objective function, which results in an optimal timetable for all the students. This approach however has two downsides:

- A study program consists of a variety of courses, where some of the courses are mandatory and some have to be chosen, but are still mandatory (for instance: a student has to pick two out of five given courses). Next to the mandatory courses a student has to complete a fairly high amount of courses which he can choose (almost) freely. The result of this system is that almost every student that graduates has a unique curriculum. This makes it impossible to form groups, which prevents the search process from speeding up. An objective function that involves all the 30.000+ students would cost a lot of time, which is not wanted in a local search.
- There is also a more practical problem: students have to enroll for a course. The deadline of this enrollment varies for each department between the start of each academic year

and the start of the course. However, the timetable should be known at the start of the academic year. Since not all the enrollments are known at this moment of time it is impossible to make an optimal timetable for all the students, including those who haven't enrolled any courses. One could solve this problem by calculating estimate values. There are however courses that are influenced heavily by worldly events; for instance: a history course regarding the Middle East greatly increases in popularity when important events occur in this region. This causes a lot of noise in the calculated estimates, so it still leaves many uncertainties. Besides uncertainty of the number of participants of a lecture, there is also uncertainty in the combinations of enrolled courses of a single student: the specific combination of courses of each student is needed in order to create a timetable that has as few gaps as possible. The calculations of these estimates are beyond the scope of this project.

3.2.1 Quarter Penalty

Because of these downsides a simpler approach is used. This approach is based on the popularity of certain time-periods. Rewarding the algorithm to schedule lectures as close to the middle of the day as possible reduces the chance on gaps in the timetable in a simple and effective way. To model this reward system a function is introduced that maps every point of time to a value, indicating the reward to schedule a meeting at this point of time. This is modeled by the Total Quarter Penalty $TQP(T)$ of timetable T:

$$TQP(T) = \sum_{m \in M} \sum_{t=m_{start}}^{m_{end}} QuarterPenalty(t) :$$

Here $QuarterPenalty(t)$ is a function that maps a point of time t to a reward/penalty value.

This approach has another nice side effect: most people do not like to get up early. This preference (or necessity for some students) can easily be modeled with the $QuarterPenalty$ function. An example of a $QuarterPenalty$ implementation can be found in figure 6.

3.2.2 Empty Room Penalty

Using only the Quarter penalty as an objective function will result in a timetable where certain time-periods have a high concentration of scheduled meetings (in our case that would be around noon).This is not without any risk. Rooms (or even worse: buildings) can be temporarily unavailable due to numerous reasons ranging from maintenance to the fall of a crane on a building (which actually happened in Utrecht in 2007). The meetings that were scheduled in these rooms should all be rescheduled to another room. This will prove to be difficult on the busy hours (around noon) since most of the rooms are used because of the high rewards these hours give. In the worst case the meeting should be scheduled to another room *and* another point of time. This is highly unwanted since it will require a lot of communication to have the teacher and students at the right spot at the right time. To prevent these kind of situations a *buffer* is introduced. This buffer indicates the amount of rooms that should be empty at any given time. The second part of the objective function is extended with the Buffer Penalty $BP(T)$ to include this aspect:

$$BP(T) = \sum_{t=0}^{end} EmptyRoomPenalty \times BufferPenalty(t) :$$

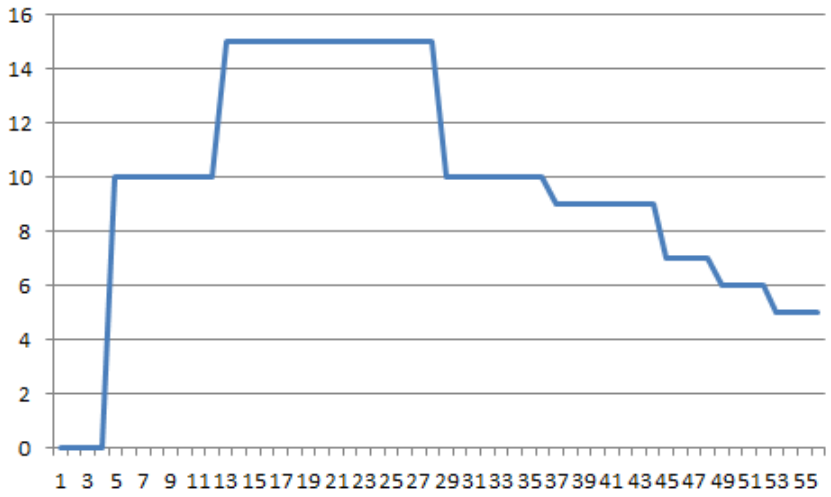

Figure 6: An example of a $QuarterPenalty(t)$ function. This function gives a relative high penalty for scheduling early. It also gives a lower reward for relatively late points of time. In this example a day is divided into quarters of an hour, starting at 8:00 and ending at 22:00. To illustrate: quarter 0 = [8:00-8:15], quarter 1 = [8:15-8:30], etc. Quarters 12 until and including 27 stand for 11:00-15:00.

Here *end* is the end of the timetable and $EmptyRoomPenalty$ is a negative constant to keep all the different penalties in proportion. $BufferPenalty(t)$ determines the penalty when there are less empty rooms available than set by the threshold:

$$BufferPenalty(t) = \begin{cases} (threshold - \#EmptyRooms(t))^2 & \text{if \#EmptyRooms(t)} < \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

Here $\#EmptyRooms(t)$ is the amount of rooms that are empty at time t; this number is based on the *category* of a room. Each room is put in a certain category, depending (in our implementation) on the capacity of the room, see table 9 for an example.

Category	Room capacity
1	0-20
2	20-50
3	50-90
4	90-120
5	120+

Table 9: The room categories used in the implementation of this thesis.

One could choose to create more categories, so the rooms can be split up in greater detail. However, the empty rooms constraint is introduced as a buffer for *emergencies*; when an emergency occurs, one should not be too picky on the room. Having a lot of different room categories also creates a lot of extra memory-usage, since the algorithm has to keep track of all the empty rooms for every category for every quarter of an hour of the week.

The squaring of the difference of the amount of empty rooms with the threshold is done to further penalize the situation where the amount of empty rooms differs greatly from the threshold, which is an unwanted situation.

When a room becomes empty this room will be added to the appropriate EmptyRooms-list (according to its category) for the analogous quarters of an hour. For example: when a room of category 3 becomes empty from 9:00 - 10:00 (quarters 4 until and including 7) as a consequence of the swapping of a lecture, then this room will be added to the following lists:

- EmptyLists(3, 4)
- EmptyLists(3, 5)
- EmptyLists(3, 6)
- EmptyLists(3, 7)

Here EmptyLists(X,Y) keeps track of all the empty rooms. It needs two parameters in order to return the appropriate list of empty rooms: the category and the quarter of an hour.

The consequence of this approach is that some room rescheduling has to be done when an emergency occurs. When a lecture has to be rescheduled to a different room because its original room has become inaccessible, then it might be the case that there is no empty room that is empty during the *complete* duration of the lecture. However, there might be a room available for the first half of the lecture, and another for the second half of the lecture. To overcome this problem some lectures have to be reassigned to different rooms to clear up one room for duration of the roomless lecture, see figure 7 for an example. We do realize that this is not good enough for a fully operational version of this software. However, it is sufficient for a proof of concept.

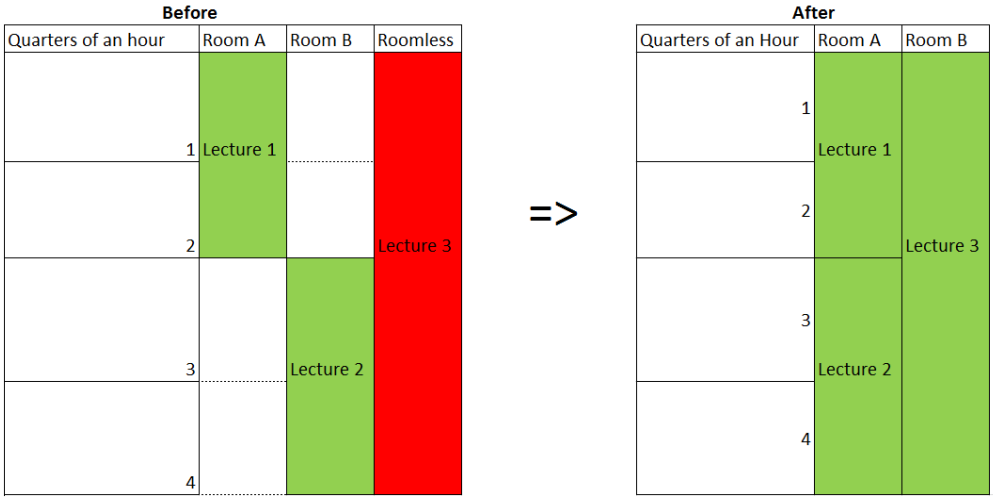

Figure 7: In this simplified example (the size of the buffer is only one room) lecture 3 has to be rescheduled since its original room has become inaccessible. To solve this problem one of the options is to reschedule lecture 2 to room A so that room B becomes available for lecture 3. After the rescheduling the timetable is feasible again.

3.2.3 Room Penalty

Utrecht University has several courses that attract an enormous amount of students (for instance: law or social courses). Utrecht University does not have appropriate rooms to accommodate such a large amount of students. Therefore *external* rooms can be rented, which cost money. These external rooms are very big and the number of participants of a

meeting only sporadically exceeds 200 persons, so Utrecht University sees no reason to build bigger rooms. Since the external rooms cost money and are not close to any other room of Utrecht University scheduling a lecture in an external room will give a penalty, which should also be included in the objective function. This is done by the Total Room Penalty $TRP(T)$:

$$TRP(T) = \sum_{m \in M} \sum_{t=m_{start}}^{m_{end}} RoomPenalty(m_{room}, t) :$$

Here m_{room} is the room where meeting m is scheduled in. The penalty of room r at time t is defined by $RoomPenalty(r, t)$:

$$RoomPenalty(r, t) = \begin{cases} cost(r, t) & \text{if } r_{external} \\ 0 & \text{otherwise} \end{cases}$$

Here $cost(r, t)$ defines the cost of renting external room r at time t. This way variations of renting prices can be included in the algorithm. Note that $cost(r, t)$ should be a negative value. The property $r_{external}$ is *true* if room r is external; *false* otherwise.

3.2.4 Unscheduled Penalty

The last aspect of the objective function is the ability to keep meetings unscheduled. This will give the search algorithm more freedom in finding a feasible solution. However, these unscheduled lectures should eventually be scheduled. The Total Unscheduled Penalty $TUP(T)$ is introduced to stimulate the algorithm in finding solutions with as few unscheduled lectures as possible.

$$TUP(T) = \sum_{m \in M} UnscheduledPenalty(m) :$$

Here the penalty for leaving a meeting unscheduled $UnscheduledPenalty(m)$ is defined as follows:

$$UnscheduledPenalty(m) = \begin{cases} UnscheduledPenalty & \text{if } m_{unscheduled} \\ 0 & \text{otherwise} \end{cases}$$

Here $UnscheduledPenalty$ is a penalty set by the user. The property $m_{unscheduled}$ is *true* in case meeting m is still unscheduled; *false* otherwise.

The total objective function for timetable T can now be defined as follows:

$$f(T) = TQP(T) + BP(T) + TRP(T) + TUP(T)$$

Since extra penalty is given for undesired aspects of the timetable, a higher objective function naturally results in a better timetable, which in turn makes this problem a *maximization* problem. Of course, this problem can easily be rewritten to a minimization problem by flipping the penalty costs. However, having a negative penalty seems a bit more intuitive than positive penalties.

The objective function now covers multiple desires from several parties. The concrete penalties like the chart of figure 6 still need to be filled in. When the algorithm is used in practice all these penalties should be aligned with the different parties to make sure that no party gets overshadowed by the demands of the other parties. Another possibility is to create multiple setups which have different penalty-values; these will all produce different timetables. The scheduler can then pick the best suited timetable.

4 Implementation

This section discusses the concrete implementation of the algorithm. First the global approach for finding an optimal timetable is explained. After that the datastructures will be looked into. To conclude this section the different local search operators are introduced.

4.1 Global Approach

In this section a small overview of the process is given. The individual parts of the process will be examined in the subsections.

As will be explained in section 4.1.1 a stamp is basically a small timetable consisting of only the regular lectures. This small timetable will then be plugged into a timetable that will cover *all* the weeks and all the lectures (the incidental lectures are also inserted at this stage); this will be further known as the *complete timetable*. For both the small and the complete timetable the same workflow is used. This is possible because at its core they are both timetables; they only differ in length and lectures (regular vs incidental).

First an initial timetable is generated by greedily inserting the lectures (regular lectures for the stamp; incidental lectures for the complete timetable). This greedy method can be adjusted by setting a lecture sorter and/or a room sorter. The lecture sorter sorts all the to-be-scheduled lectures; this influences the order in which the lectures are scheduled into the timetable. This way lectures that are difficult to schedule can get priority over the easy lectures in the greedy process.

When a lecture is being scheduled multiple rooms are examined to find a suitable spot. The order in which these rooms are examined can be influenced by the room sorter.

After the initialization of a timetable this timetable will be optimized. This is done by the local search meta-heuristic 'Simulated Annealing'. In the local search many small adjustments are made to the timetable to improve its quality. This proves out to greatly improve the timetables. There are several ways to find these small adjustments; therefore multiple operators are introduced. By having a variety of operators at his disposal the local search has many different ways to adjust the timetable.

In summary, the total process for the creation of a full timetable is as follows:

1. Create a stamp:
   (a) Initialize a stamp by inserting only the *regular* lectures with the greedy algorithm
   (b) Optimize the stamp using Simulated Annealing
2. Create the complete timetable:
   (a) Import the stamp into the complete timetable so that all the regular lectures are scheduled
   (b) Initialize the complete timetable by inserting the *incidental* lectures with the greedy algorithm
   (c) Optimize the complete timetable using Simulated Annealing

4.1.1 Two-phase approach

The implementation is based on the model described in section 3, thereby making a distinction between *regular* and *incidental* lectures. This distinction turns out to be essential for the algorithm to work efficiently, since it allows us to create a two-phased approach:

1. Schedule all the regular Global Lectures in a *stamp*. The definition of a stamp can be found below.
2. The stamp will be applied for every week in the timetable, scheduling meetings for every week in $GlobalLecture_{weeks}$. Next, the incidental lectures are scheduled in this complete timetable created by applying the stamp.

Stamp A *stamp* is the blueprint of a timetable that is used to schedule the regular lectures into the complete timetable. A stamp has the same size as one week of a timetable; all the regular lectures are scheduled in this small timetable. The stamp can then be inserted into a complete timetable (one that covers all the weeks) by scheduling every lecture in the stamp on the same spot in the complete timetable for every week that the lecture has to be repeated.

An illustration of this idea can be found in figure 8. This example is based on a single room and four weeks; needless to say, this is a simplified example. Note that it is not always the case that a regular lecture is repeated for *every* week; not every week of the timetable is present in $GlobalLecture_{weeks}$. By firstly creating a stamp and then applying the stamp on the complete timetable instead of scheduling all the lectures separately gives us two advantages:

- Since the stamp is repeated every week this makes sure that the regular lectures are always on the same time in the same room, which was one of the student's and teacher's wishes. The lectures imported in the total timetable by applying the stamp can be handled as solid items, meaning that the second phase may not reschedule these lectures. This makes sure that the regular lectures *stay* regular.

  When the first phase is skipped and all the lectures are scheduled at the same time additional checks have to be made to ensure the regularity of the regular lectures. Besides being error prone this also takes time which could be spent better on applying search operators.

- Splitting the enormous problem into two smaller subproblems makes optimizing the timetable faster, and thereby giving the algorithm more time to optimize. By firstly optimizing the stamp-solution the total timetable will have a head start. Splitting the problem into two subproblems has as a downside that the outcome might be suboptimal. However, the gain in speed outweighs the suboptimality of the produced timetable.

4.1.2 Initial timetable

As is common in solving local search problems, it all starts with a generated initial solution, for both the stamp and the total timetable. This is done in a greedy way by searching for a *spot* in the timetable for every lecture that has to be scheduled.

Spot A spot is a place in the timetable. It is defined by a point of time $spot_{time}$ and a location $spot_{room}$. Together with $spot_{week}$ this combination is unique, since searching for a spot for lecture l only spots with week equal to l_{week} are taken into account.

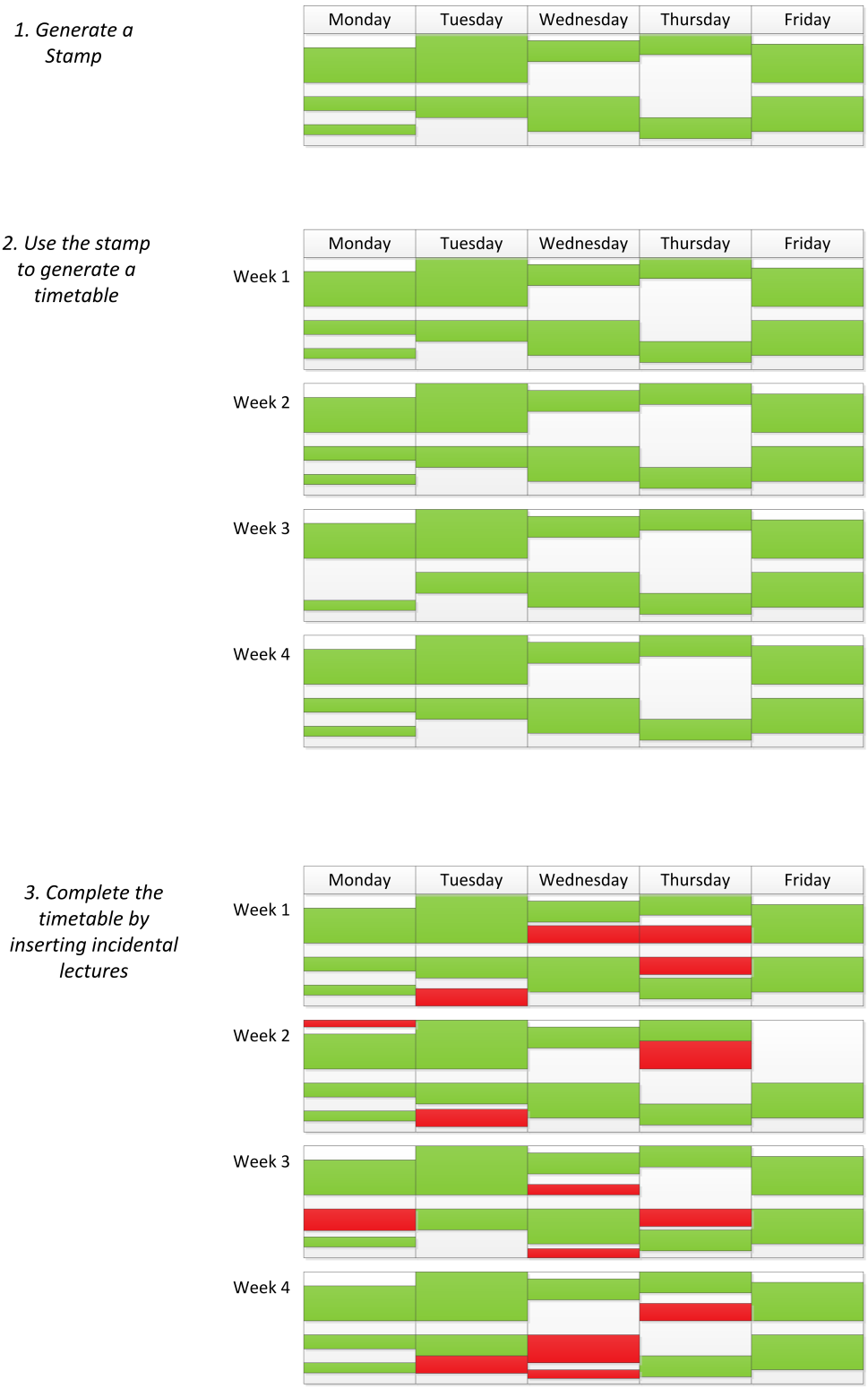

Figure 8: The generation of a timetable with optimization-parts in step 1 and 3. The green items are regular lectures; the red items are incidental lectures. Note that this timetable is for only one room and only four weeks; a real timetable would consist of multiple of such room-timetables.

To generate an initial timetable the algorithm searches for a spot in the timetable for each lecture $l \in L$, according to the method described in algorithm 1. Here, each room in the roomset of lecture l is tested on its capacity. When the room is large enough to accommodate the lecture the algorithm searches for a valid starting quarter.

Algorithm 1 Schedule lecture l in the timetable

```
1: procedure GreedySchedule(Lecture l)
2:     for Every room r ∈ l_R do
3:         if l_participants ≤ r_capacity then
4:             for t ∈ l_T do
5:                 if IsValidStartingQuarter(t, r, l) = true then
6:                     Schedule(l, t, r)
7:                     return true
8:                 end if
9:             end for
10:        end if
11:    end for
12:    return false
13: end procedure
```

The $IsValidStartingTime(t)$ checks whether or not the given point of time i is a valid starting quarter by checking the availability of each point of time j in room r, where $i \leq j < i+l_{duration}$. When a spot is found, the algorithm *immediately* schedules the lecture in this spot to speed up the process. This greedy search is very short-sighted since it schedules one lecture at the time, without feedback from any other lectures.

The algorithm described in algorithm 1 searches in every room r in l_R for a suitable spot for every lecture. Since the greedy algorithm immediately schedules this lecture when a spot is found, the order of the rooms in l_R influences the initial timetable, and thereby influences the total outcome. Therefore two sorting methods are introduced to generate different initial timetables.

4.1.3 Room sorter

To sort a list of rooms, the algorithm has to know how to compare two rooms. This is done by implementing a compare function: $Comparer(Room\ r_1, Room\ r_2)$.

4.1.3.1 RoomSize Comparer

The first and most straightforward sorting method we implemented was the *RoomSize Comparer*; the pseudo code of the compare method is found in algorithm 2. This method compares two rooms, based on the capacity of the rooms. The justification of this heuristic is that a lecture should be scheduled in a room that is as small as possible (as long as it fits), leaving the bigger rooms available for the bigger lectures. This utilizes the available space most.

4.1.3.2 Popularity Comparer

Preliminary results indicated that a portion of the lectures remain unscheduled. As a reaction on this phenomenon the *Popularity Comparer* is introduced. This compare-function calculates

Algorithm 2 Returns the room that should come up first in the sorted list

```
1: procedure RoomSizeComparer(Room r1, Room r2)
2:     if r1_capacity < r2_capacity then
3:         return r1
4:     else
5:         return r2
6:     end if
7: end procedure
```

a score for each room which is an estimate of the popularity of room r: the more lectures that have r in their roomset R, the higher $r_{popularity}$. Note that only the rooms that are big enough are taken into account for the calculation of the popularity score; rooms that are too small are excluded.

Popularity score The popularity score is a score for a room r, constructed by counting how often r is included in a lecture's roomset l_R: $r_{popularity} = \sum_{l:r \in l_R} \frac{1}{l_{size(R)}}$, where $size(R)$ of roomset R is calculated with the capacity of the room and the number of participants of a lecture taken into account.

An example of the calculations of this score can be found in figure 9.

Overview of rooms in the roomset of the lectures:

Lecture 1	**Lecture 2**	**Lecture 3**	**Lecture 4**	**Lecture 5**
Roomset:	Roomset:	Roomset:	Roomset:	Roomset:
A	D	A	B	A
B	E	B	C	D
C		C		
		D		

Calculations table:

Contributions:	Room A	Room B	Room C	Room D	Room E
Lecture 1	1/3	1/3	1/3		
Lecture 2				1/2	1/2
Lecture 3	0	1/3	1/3	1/3	
Lecture 4		0	1		
Lecture 5	1/2			1/2	
					+
	5/6	2/3	1 2/3	1 1/3	1/2

Figure 9: An example of the calculation of the popularity scores of the rooms. Room A is too small for lecture 3; so is room B for lecture 4. It turns out that room C is the most popular room, quickly followed by room D.

For simplicity we leave out the time-dimension, resulting in a situation where each room can only have one lecture assigned. When the algorithm tries to schedule the lectures (starting with lecture 1) using the lexicographical ordering of the rooms in the roomset as stated in the example, the following assignment will be found:

1. {A,B,C} → A
2. {D,E} → D
3. {B,C,D} → B
4. {C} → C
5. {A,D} → ?

Lecture 5 cannot be scheduled, since both room A and room D are already taken. When the roomsets are sorted by the popularity scores (try to assign the lecture to the least popular room) the algorithm will find the following assignment:

1. {B, A, C} → B
2. {E, D} → E
3. {E, B, D} → D
4. {C} → C
5. {A, D} → A

This approach schedules all the lectures and thereby producing a better timetable.

4.1.4 Lecture Comparer

The greedy algorithm not only depends on the order of the rooms; it also depends on the order of the lectures in the total lectures list. Therefore two sorting methods are introduced to sort the lectures list: *HardToSchedule* and *SizeComparer*. These lecture-sorting methods are in line with respectively Popularity comparer and RoomSize comparer.

4.1.4.1 HardToSchedule

As an extension of the popularity score of a room, the HardToSchedule sorting method is introduced to sort the lectures, based on a score-function similar to the method mentioned in section 4.1.3.2.

HardToSchedule Each lecture l receives a score based on the popularity of the rooms in l_R: $l_{HardToSchedule} = \sum_{r \in l_R} r_{popularity}/size(R)$

The HardToSchedule-scores of figure 9 can be found in figure 10. Note that the HardToSchedule-score of lecture l is the average of the popularity-scores of the rooms in l_R. The more trouble the algorithm is expected to have on scheduling lecture l, the higher $l_{HardToSchedule}$. The algorithm should start with scheduling the hardest lectures, since these lectures have the highest competition on finding a spot in a room of its roomset.

	Room A	Room B	Room C	Room D	Room E	∑	HardToSchedule
Lecture 1	5/6	2/3	1 2/3			3 1/6	**1.056**
Lecture 2				1 1/3	1/2	1 5/6	**0.917**
Lecture 3		2/3		1 1/3	1/2	2 1/2	**0.833**
Lecture 4			1 2/3			1 2/3	**1.667**
Lecture 5	5/6			1 1/3		2 1/6	**1.083**
					+		

Figure 10: An example of the calculation of the HardToSchedule-scores of the lectures. For each lecture the popularity-scores of each room is summed together. This sum is then divided by the number of rooms, to calculate an average popularity.

4.1.4.2 SizeComparer

A more obvious way to sort the lectures is by sorting the lectures on their number of participants. The idea behind this sorting method is that lectures that have a lot of participants are mostly harder to schedule than lectures with less participants, since these lectures have less rooms at their disposal. The main advantage of this lecture compare-function over the HardToSchedule compare-function, is that this sorting method does not need to do some hocus pocus-calculations before it can be used.

4.1.5 Local search

The initial timetables are generated by algorithm 1, using one of the room and lecture-compare methods. *Simulated Annealing* (Kirkpatrick et al. (1983)) was chosen as meta-heuristic to further improve this initial timetable, mainly because of the intuitive way to get out of a local optimum.

The name comes from the metallurgy, where 'annealing' is a technique where the metal is heated and then cooled. The cooling part is strictly controlled to increase the size of the crystals of the metal, thereby decreasing the number of defects. In computer science this meta-heuristic is applied quite often, where it is implemented as follows: at the beginning of the search process the temperature (set by the user) is very high; after each iteration this temperature is decreased. The temperature determines the chance to continue with a timetable T with score $S(T_{i+1})$ that is worse than the previous solution with fitness $S(T_i)$ in a maximization problem as follows: $f(x) = \begin{cases} 1 & S(T_{i+1}) \geq S(T_i) \\ e^{\frac{S(T_i)-S(T_{i+1})}{Temperature}} & otherwise \end{cases}$

When $f(x)$ is larger than a uniform random number the solution will be accepted, meaning the local search will continue with timetable T_{i+1}. The acceptance of a worse solution than the previous solution allows the local search to take a few steps back to reconsider its decisions. This allows the algorithm to get out of a local optimum and explore a different part of the search-space.

The temperature has a direct influence on the acceptance of a solution. By decreasing the temperature after every iteration the chance of accepting a solution worse than the current solution is decreased. This makes sure that the algorithm *explores* a lot in the early stage of the search process, but *exploits* more towards the ending of the search process.

To determine the temperature and cooling-down rate some user input is needed:

- The chance that a timetable should be accepted that has X-points lower score than the previous score at the beginning of the search process: $chance_{initial}$.
- The chance that a timetable should be accepted that has X-points lower score than the previous score at the end of the search process: $chance_{finish}$.
- X has to be determined.
- Number of iterations: $\#iterations$.

These parameters can then be used in the following equations to calculate the initial temperature $T_{initial}$ and the cooling-down rate CDR:

- $e^{\frac{X}{T_{initial}}} = chance_{initial}$
- $e^{\frac{X}{T_{finish}}} = chance_{finish}$
- $T_{initial} \times CDR^{\#iterations} = T_{finish}$

Solving these equations results in educated guesses for a trial-and-error process to find a suitable initial temperature and an appropriate cooling-down rate.

4.1.5.1 Partial Calculation of the Objective Function

An iteration of local search modifies the solution by only a small bit. Therefore it is inefficient to recalculate the complete score of the modified solution; this can also be done by recalculating only the part that is changed. It is vital for the local search that the recalculation of the objective function is fast, especially when the solutions are relatively large (as is the case in most timetabling problems). The partial calculation of the objective function has to take two parts into account:

- Lecture score
- Empty rooms score

The lecture score is the score a lecture receives when it gets scheduled. This includes the score it receives for scheduling at a certain time ($QuarterPenalty(t)$) and the penalty for scheduling in an external room ($RoomPenalty(m_{room}, t)$). These penalties are described in section 3.2. Since these scores are not influenced by other lectures, this calculation was fairly easy to implement.

However, the empty rooms score proved to be much harder to implement. When the local search has found a different spot for a lecture, it does not automatically apply this change; it firsts calculates the improvement of the objective function *if* it moves to that new spot. So at the time of the recalculation, the lecture is still at the original spot. The problem of this method of recalculating is that the original spot and the next spot might overlap. For instance: a lecture is scheduled in room A from 10.00 until 12.00 and the local search wants to move it to room A from 11.00 until 13.00. In this scenario the list of empty rooms of type A increases by one from 10.00 until 11.00, stays of equal size from 11.00 until 12.00, and drops one in size from 12.00 until 13.00. The overlap of spots made the calculation of the empty rooms penalty a tricky one to implement. We overcame this by *detecting* the three different parts (increment-, equal- and decrement-part); each part had to be dealt with accordingly.

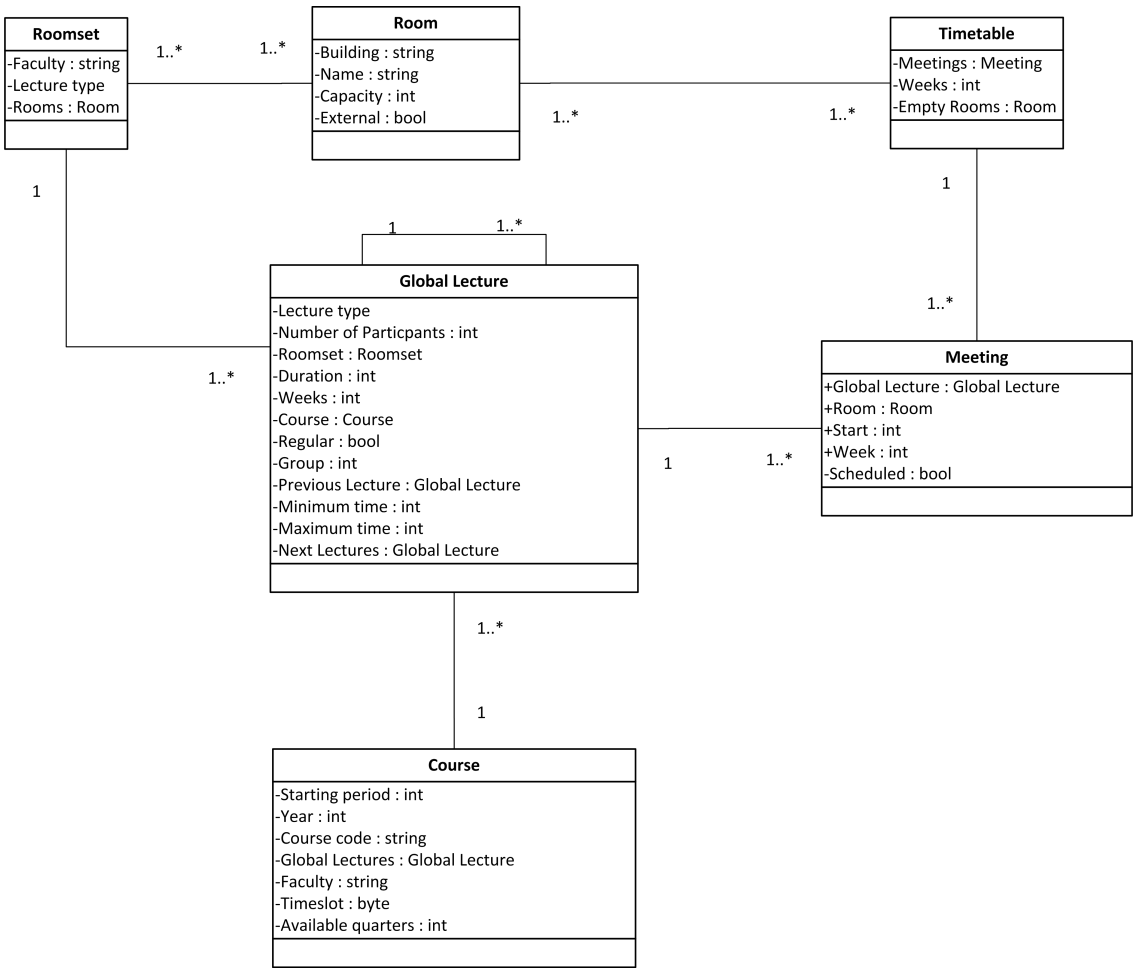

Figure 11: The class diagram of the developed software in UML[5]. The classes are stated with its key properties, as well as the relationships between the classes.

4.2 Datastructures

To get more feeling for the problem a diagram is shown in figure 11, in which the six major components are listed:

- *Course*: The courses are directly imported from the Osiris database. As stated in section 2, the primary key is the combination of Starting period, Year and Course code. Available quarters is a list of quarters in which a meeting of this course may be scheduled. This list is based on the Timeslot property of the course.
- *Global Lecture*: As stated in section 3.1.1 the Global Lectures are based on meetings that have the same key properties. Therefore this object is used to store the shared properties of those lectures. The meetings of a Global Lecture can be scheduled in the rooms defined by the Roomset of the lecture. However, it is not assured that all the rooms in the roomset are big enough for the lecture, since the roomsets are defined on a higher level. The dependency constraint as explained in section 3.1.2 is realized by the properties Previous Lecture, Minimum time and Maximum time, where Minimum and Maximum time denote the time window in which the previous lecture has to be

[5]Unified Modeling Language

scheduled. The list of Global Lectures that has to be scheduled *after* a lecture is stored in Next Lectures; this property is introduced mainly to keep the navigation between lectures fast.

- *Meeting*: A meeting is the object that is actually scheduled in a timetable, so it has to store concrete information like the starting time and room in which it will be scheduled.
- *Roomset*: In section 2 is stated that a roomset is defined by two components: Faculty and Lecture type. Next to these key-properties also a list of rooms that belong to this roomset will be stored. Note that a roomset might be empty.
- *Room*: A room is identified by its Name and its Building. Another important property is the Capacity of the room, denoting how many students can be in the room at the same time.
- *Timetable*: A timetable consists of a collection of meetings. Of course, all these meetings should be scheduled: $\forall m \in M : m_{Scheduled}$. To implement the buffer-constraint as defined in section 3.2 multiple lists of Empty Rooms are introduced to quickly determine which rooms are empty for any given point of time.

This structure is used in the implementation of the algorithm. In the concrete timetable the time is divided in *quarters of an hour*, because a portion of the lectures are not precisely one or two hours. Besides, the current system also uses a timetable based on quarters of an hour, which would make the possible transition from the current system to the system based on the algorithm discussed in this paper easier. In the structure used in this implementation, one week has 280 quarters of an hour that can be used in a timetable: 5 days from 8:00 till 22:00, see figure 12 for an example of the timetable of a single room.

4.3 Operators

An initial timetable can be generated by using a set of room and lecture comparing functions. This initial timetable will then be plugged into the local search method, where the timetable will be improved using different operators. An important function is the method that returns the X-best spots of scheduling a lecture, using the definition of spot as declared in section 4.1.2. This function is described in algorithm 3. In this algorithm every room in the roomset is tested on the capacity-constraint. When the room is big enough the algorithm checks all the valid starting quarters and adds the spot (time + room combination) to the list of possible spots. This list is sorted to make sure the best spots are in front of the list of possible spots and the best X spots are returned.

The following two operators are the base of the local search, each using the Best spots function to generate the best spots. Later in this paper, two other operators are introduced.

- *Schedule*: This operator randomly selects an unscheduled lecture. It then selects the X-best possibilities of scheduling the lecture and picks one of the options according to a weighted chance distribution. This is an insertion operator.
- *Shuffle*: This operator randomly selects a scheduled lecture. It unschedules the lecture and then inserts it again. The lecture is inserted through a weighted chance distribution of the X-best spots (the previous spot might be included in this collection of spots). This operator basically *swaps* the spot of a lecture.

Note that there is no sole deletion operator. Removing a lecture from the timetable is in no situation beneficial; it always has to be inserted again. Removing and inserting is already

	Monday	Tuesday	Wednesday	Thursday	Friday
0: (8:00-8:15)					
1: (8:15-8:30)					
2: (8:30-8:45)					
3: (8:45-9:00)					
4: (9:00-9:15)	201000339 (277...		WB2BD0900 (14...		
5: (9:15-9:30)	201000339 (277...		WB2BD0900 (14...		
6: (9:30-9:45)	201000339 (277...		WB2BD0900 (14...	200700188 (226...	
7: (9:45-10:00)	201000339 (277...		WB2BD0900 (14...	200700188 (226...	
8: (10:00-10:15)	201000339 (277...		WB2BD0900 (14...	200700188 (226...	
9: (10:15-10:30)	201000339 (277...		WB2BD0900 (14...	200700188 (226...	
10: (10:30-10:45)	201000339 (277...		WB2BD0900 (14...	200700188 (226...	
11: (10:45-11:00)	Change quarter		Change quarter	200700188 (226...	
12: (11:00-11:15)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
13: (11:15-11:30)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
14: (11:30-11:45)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
15: (11:45-12:00)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
16: (12:00-12:15)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
17: (12:15-12:30)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
18: (12:30-12:45)	GG1V11012 (171...	HUMB11001 (25...	GG1V11012 (176...	200700188 (226...	200501232 (181...
19: (12:45-13:00)	Change quarter	Change quarter	Change quarter	200700188 (226...	Change quarter
20: (13:00-13:15)				200700188 (226...	
21: (13:15-13:30)	201000154 (141...	200700188 (316...	201000245 (311...	Change quarter	
22: (13:30-13:45)	201000154 (141...	200700188 (316...	201000245 (311...		201000107 (245...
23: (13:45-14:00)	201000154 (141...	200700188 (316...	201000245 (311...		201000107 (245...
24: (14:00-14:15)	201000154 (141...	200700188 (316...	201000245 (311...		201000107 (245...
25: (14:15-14:30)	201000154 (141...	200700188 (316...	201000245 (311...		201000107 (245...
26: (14:30-14:45)	201000154 (141...	200700188 (316...	201000245 (311...	200900174 (262...	201000107 (245...
27: (14:45-15:00)	201000154 (141...	200700188 (316...	201000245 (311...	200900174 (262...	201000107 (245...
28: (15:00-15:15)	Change quarter	Change quarter	201000245 (311...	200900174 (262...	201000107 (245...
29: (15:15-15:30)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	Change quarter
30: (15:30-15:45)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	
31: (15:45-16:00)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	200700060 (200...
32: (16:00-16:15)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	200700060 (200...
33: (16:15-16:30)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	200700060 (200...
34: (16:30-16:45)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	200700060 (200...
35: (16:45-17:00)	200900206 (798)...	200300412 (298)...	201000245 (311...	200900174 (262...	200700060 (200...
36: (17:00-17:15)	Change quarter	Change quarter	Change quarter	200900174 (262...	200700060 (200...
37: (17:15-17:30)				200900174 (262...	200700060 (200...
38: (17:30-17:45)				200900174 (262...	Change quarter
39: (17:45-18:00)				200900174 (262...	
40: (18:00-18:15)				200900174 (262...	
41: (18:15-18:30)				Change quarter	
42: (18:30-18:45)					
43: (18:45-19:00)					
44: (19:00-19:15)					
45: (19:15-19:30)		200700204 (300...			
46: (19:30-19:45)		200700204 (300...			
47: (19:45-20:00)		200700204 (300...			
48: (20:00-20:15)		200700204 (300...			
49: (20:15-20:30)		200700204 (300...			
50: (20:30-20:45)		200700204 (300...			
51: (20:45-21:00)		200700204 (300...			
52: (21:00-21:15)		Change quarter			
53: (21:15-21:30)					
54: (21:30-21:45)					
55: (8:00-8:15)					

Figure 12: An example of a timetable of a single room, divided in quarters of an hour. When a quarter of an hour is filled it shows the course code in the corresponding slot; otherwise it is empty. Note that there is one change-quarter after each lecture to facilitate a smooth transition between lectures.

Algorithm 3 Find the X-best spots to schedule lecture l in the timetable

```
1: procedure BEST SPOTS(Lecture l, int X, Comparer)
2:     Spotslist ← newList()
3:     for Every room r ∈ l_R do
4:         if l_participants ≤ r_capacity then
5:             for i ∈ l_s do
6:                 if IsValidStartingQuarter(i, r, l) = true then
7:                     Spotslist.Add(Spot(i, r))
8:                 end if
9:             end for
10:        end if
11:    end for
12:    Spotslist.Sort(Comparer)          ▷ Sort the list with the given compare function
13:    return Spotslist.GetFirst(X)
14: end procedure
```

done by the *Shuffle*-operator.

Next to these base operations, three more operators are introduced, which are less straightforward than the base operations:

- Branch and Bound heuristic
- Shift
- Chain reschedule

In the following subsections these specialized operators are explained.

4.3.1 Branch and Bound Heuristic

With only the base operators the local search is still quite shortsighted: it schedules each lecture individually. It searches for the best spots available without taking other lectures into account. This shortage is somewhat filled by the introduction of the comparing functions and the implementation of Simulated Annealing, but there might be gains to make when more exhaustive algorithms are used.

Therefore a Branch and Bound algorithm has been implemented. The Branch and Bound heuristic randomly picks a scheduled lecture l, together with a group of other lectures that have the same timeslot, faculty and lecture type (and thereby have the same roomset). This group of lectures are in each others territory when it comes to finding spots. To find an optimal solution for this group of lectures *as a whole* instead of finding an optimal spot for each lecture individually, the Branch and Bound heuristic searches for the best combination of spots for the selected lectures. The global approach of the Branch and Bound algorithm is stated in algorithm 4.

In this algorithm a set of competing lectures of lecture l are put in a list of lectures called *Lectures*. The individual contributions of all the lectures in *Lectures* is summed together to form a lower bound: after all, a solution that has a lower (predicted) score than this sum is worse than the original solution and should not receive precious time for further exploration. For every lecture in *Lectures* multiple spots are calculated, so that the algorithm has different spots to work with. The algorithm starts with assigning a spot to the first lecture; after that

Algorithm 4 Pseudo code of the Branch and Bound heuristic

```
1: procedure BRANCH AND BOUND(Lecture l, int GroupSize, int MaxSpots)
2:     Spots ← new Spot[GroupSize]
3:     Solution ← 0
4:     Lectures ← GetCompetingLectures(l, GroupSize)
5:     for i = 0 → GroupSize do
6:         Spots[i] ← BestSpots(l, MaxSpots)          ▷ Store the best spots of this lecture
7:     end for
8:     LowerBound ← Σ_{Lecturelec∈Lectures} lec_contribution
9:     Nodes ← 0
10:    for all Spots ∈ Spots[0] do
11:        Nodes.enqueue(Node(Lectures[0], s)                          ▷ Create the root
12:        while Nodes ≠ ∅ do
13:            NodeCurrent ← Nodes.dequeue()
14:            Node[ ] Children ← Current.split()
15:            for all NodeChild ∈ Children do
16:                Child.est ← Child.GetEstimate()
17:                if Child.est ≥ LowerBound then      ▷ Expand only if promising enough
18:                    Nodes.enqueue(estimate)
19:                    if Child is leaf then
20:                        LowerBound ← Child.est
21:                        Solution ← Child.solution
22:                    end if
23:                end if
24:            end for
25:        end while
26:    end for
27:    return Solution
28: end procedure
```

it produces child-nodes, in which the next lecture is assigned to a spot. This process continues until no nodes are left to explore. Some sub-functions need more clarification:

- $GetCompetingLectures(l, GroupSize)$ returns a list of lectures (with a maximum size of $GroupSize$) that have the same timeslot, faculty and lecture type as lecture l. This makes sure that the Branch and Bound heuristic is used on a potentially problematic subproblem; it would be a shame if precious time is spent on a trivial subproblem.
- Again, the $BestSpots()$ function is used to get the Best Spots for lecture l, see algorithm 3 for more details. The number of spots that has to be found is denoted by $MaxSpots$. The Best Spots are chosen at the beginning of each Branch and Bound run to keep the internal Branch and Bound calculation fast. When the Best Spots list for a lecture has to be calculated in every node of the Branch and Bound this would take up a lot of processing time.
- Function *split* generates the children of the node it is used on. This function only produces children that do not conflict with earlier assignments of the Branch and Bound; otherwise the solution will be invalid. For instance, if a lecture occupies a spot s then all the children that also use spot s for another lecture are disregarded, since only one lecture can be given at a spot.

- The *GetEstimate*-function in line 16 is an optimistic estimate of the contribution of the objective function, to prevent the algorithm from cutting off potentially good solutions.

Since this Branch and Bound heuristic potentially does an exhaustive search of the complete searchtree, the size of this tree has to be limited. This can be done by adjusting the parameters $GroupSize$ and $MaxSpots$ of algorithm 4. It is important for the algorithm to have enough options available for each lecture; however, exaggerating the number of possible spots results in a very large searchtree which exponentially increases the running time of this sub-algorithm.

4.3.2 Shift

Frequently applying the operators used in the local search, might result in a timetable with as many holes in it as a Swiss cheese. Therefore an operator is introduced that compresses the timetable of a single room by shifting the lectures, see figure 13.

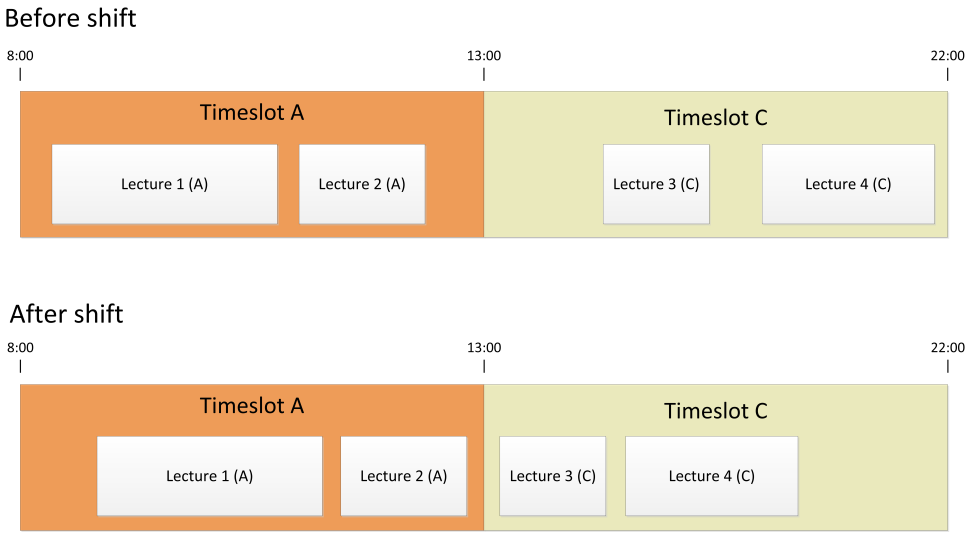

Figure 13: The schematic timetable of a room before and after the Shift-function. The free space is bubbled towards the beginning and end of the day.

Since the penalty received from the QuarterPenalty part in the objective function (see section 3.2) is highest at noon, the lectures should all be shifted towards the middle of the day. However, the quarter to which the lectures should be shifted towards is parameterized, so it can be adjusted.

Algorithm 5 Pseudo code of the Shift operator, used on room r

```
1:  procedure SHIFT(int Middle)
2:      FirstHalf ← GetFirstHalf(r, Middle)        ▷ Sorted late to early
3:      SecondHalf ← GetSecondHalf(r, Middle)      ▷ Sorted early to late
4:      for all Lecturel ∈ FirstHalf do
5:          ShiftRight(l)
6:      end for
7:      for all Lecturel ∈ SecondHalf do
8:          ShiftLeft(l)
9:      end for
10: end procedure
```

In algorithm 5 a day is split in two: the first half and the second half. The function $GetFirstHalf(r, Middle)$ and $GetSecondHalf(r, Middle)$ returns the lectures that are scheduled in room r on the respectively first and second half of the day. The algorithm then tries to shift them towards the middle using $ShiftRight(l)$ and $ShiftLeft(l)$.

4.3.3 Chain Reschedule

Recall from section 3.1.2 that a lecture may depend on another lecture. For instance, lecture B can be dependent on lecture A with a minimum time of 2 hours and maximum time of 24 hours, meaning that lecture B should start between 2 or 24 hours *after* lecture A. Because of this dependency the amount of valid starting times for lecture B is smaller than when lecture B does not have any dependencies, making it more difficult to schedule.

Preliminary results indicate that the algorithm has difficulties dealing with chains of lectures that all directly or indirectly depend on each other. Each lecture in such a chain can only be scheduled when the previous lectures are scheduled. This has as a consequence that leaving one lecture in the chain unscheduled causes that all the lectures that are directly or indirectly depending on this lecture cannot be scheduled. Leaving a lecture in the beginning of the chain unscheduled will have major consequences for the rest of the lectures. This property makes the decision making of the lectures in the beginning of the chain very important; when this is done badly, subsequent lectures will be unable to find a spot since the day may simply be too short, see figure 14 for an example.

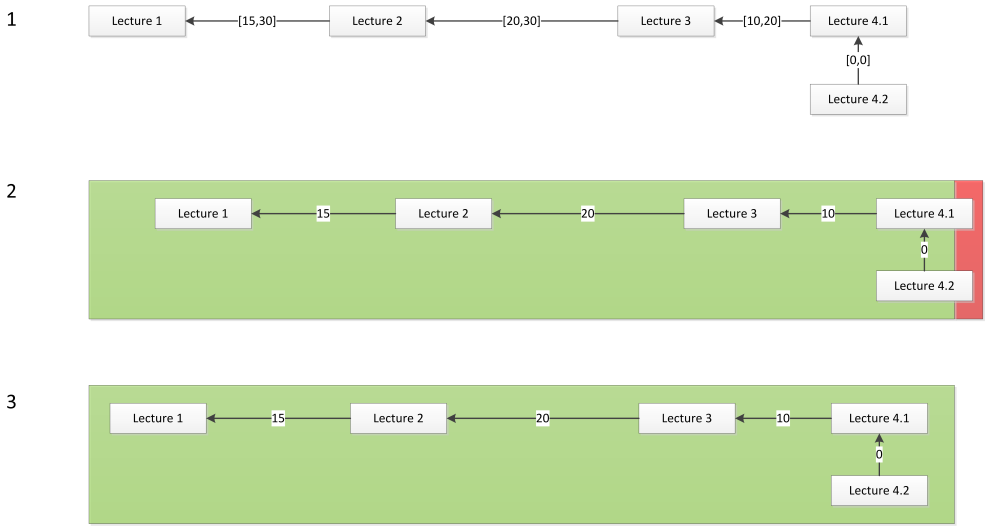

Figure 14: The schematic timetable for a room for one day (the green rectangle is the daylength). In the first part the dependencies of the lectures are denoted, with the timewindows on each dependency-arc. In the second part the first lectures are scheduled on the spots that are most rewarded, filling in the time windows. However, lecture 4.1 and 4.2 cannot be scheduled now, because the day is not long enough (the red rectangle indicates the amount of extra time needed). In the third part this has been fixed by scheduling all the lectures as early as possible, as a result of the Chain Reschedule function.

Therefore the *Chain Reschedule* operator is introduced. This operator unschedules a whole chain of lectures and then tries to insert it again as early as possible so that it has the biggest chance to schedule the *complete* chain of lectures, see algorithm 6.

Algorithm 6 Pseudo code of the Chain Unschedule operator

```
1: procedure CHAIN UNSCHEDULE(Lecture l)
2:     Lectures ← GetChain(l)
3:     for all Lecture k ∈ Lectures do
4:         k.Unschedule()                          ▷ Unschedule each lecture
5:     end for
6:     Lectures.DependencySort()
7:     for all Lecture j ∈ Lectures do
8:         Spots ← BestSpots(j, 10, EarliestPossible)
9:         j.Schedule(Spots.GetRandomWeighted())
10:    end for
11: end procedure
```

The function *DependencySort*() on line 6 sorts the lectures on their depth in the dependency tree: the more lectures a lecture (indirectly) depends on, the deeper it is in the dependency tree.

Again, the *BestSpots* function is called, except with a different compare function: *Earliest-Possible*. This compare function sorts the spots on starting time. The function $GetRandomWeighted$ returns a spot according to a weighted chance distribution based on the contribution of the spots on the objective function.

4.3.4 Restarts

Recall from section 4.1 the global process:

1. Generate a stamp (including only the regular lectures)
2. Optimize the stamp
3. Plug the stamp into the complete timetable
4. Insert the incidental lectures into the complete timetable
5. Optimize the complete timetable

Once the stamp is plugged into the complete timetable the stamp stays unchanged for the further search process. So if choices were made in the optimization of the stamp that benefits the objective function of the stamp, but negatively influence the quality of the solution of the complete timetable (e.g. regular lectures that block incidental lectures on crucial points) then these choices are not reconsidered in the current situation. In other words: a stamp with a higher score may not always lead to an optimal complete solution.

To overcome this problem multiple stamps are stored during the optimization part of the stamp. The first phase of the approach now produces several stamps, so that the effect of different stamps can be explored. The local search of the complete timetable was slightly adjusted so that it could deal with different stamp-solutions:

1. Import a stamp into the complete timetable
2. Optimize the complete timetable
3. Retract the stamp from the complete timetable

4. Import a different stamp into the timetable
5. Optimize the complete timetable

The first two steps are standard; a stamp has to be imported into the complete timetable. However, the last three steps are not; these were added to implement the 'switchstamp'-feature. The import function in step 4 might cause some trouble: an incidental lecture might have found a spot that was left open by stamp 1, but this spot might not be open when stamp 2 was used. In this case the incidental lecture will be unscheduled, giving priority to the regular lectures of the stamp. Note that steps 3 until 5 can be applied as often as the user wants to; in this thesis there were three other stamps stored besides the fully optimized one. This results in a local search graph similar to the one in figure 15.

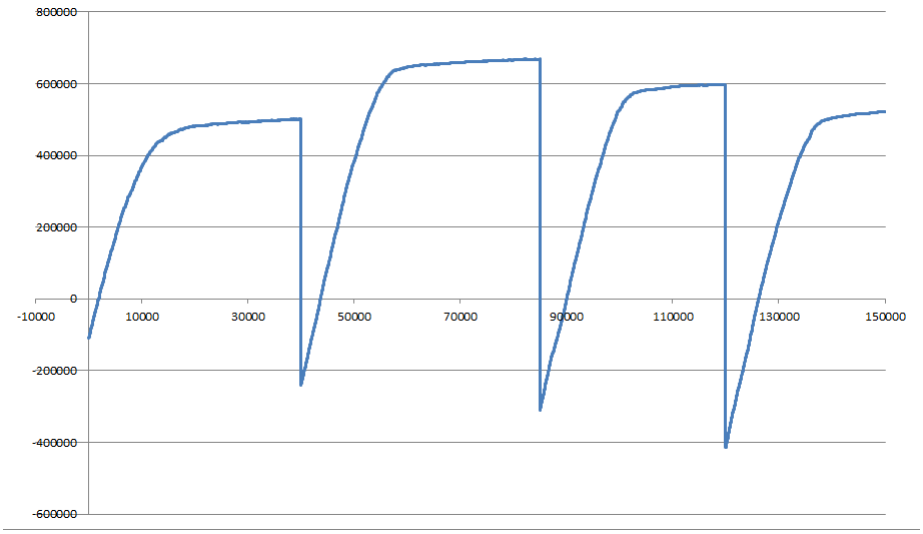

Figure 15: The local search landschape of one of the experiments. The drop in objective function is the result of the unscheduling of lectures because of lectures that overlap or violations of the dependency-constraint.

5 Results

Multiple test runs are done to compare the different settings that can be set in the algorithm. In the first experiments the different combinations of comparing functions as mentioned in section 4.1.2 are tested; these combinations influence the quality of the initial solution and have therefore influence on the total timetable. As a result of these experiments the best combination of comparing functions is chosen; this combination will be used in the other experiments.

As a second experiment, the effect of a local search is compared to a greedy algorithm. This is done by leaving out the local search in the search for a stamp solution and/or total solution. The main reason for this particular experiment is to imitate the searchprocess of Syllabus so that the algorithm of this paper can be compared with an algorithm similar to the algorithm of Syllabus. Recall that we do not know the exact algorithm used in Syllabus, so we cannot firmly conclude that this method (or a similar method) is used within Syllabus. However, due to the performance of Syllabus and its running time it strongly appears to be a greedy algorithm.

The third run of experiments is to test a simple implementation of the *Branch and Bound heuristic* for small sets of lectures. This function is called within the regular local search method.

In the last part of the experiments all the different operators mentioned in section 4 are applied to search for the best timetable. In practice, this setup will be run to create a timetable.

5.1 Test environment

As is explained in section 3.2 the objective function consists of different aspects. The first aspect is the *Quarter Penalty*, giving each quarter q a penalty p_q when a lecture is scheduled during q. The scoring-function of this Quarter Penalty is given in figure 16. Note that negative penalties are given for very early and late points of time to punish undesired starting times. However, these unwanted starting times should not be penalized too high; it is always better to schedule a lecture on a terrible spot than to leave a lecture unscheduled. Opposite to this are the points of time on the middle of the day; these starting times receive positive penalties to stimulate the algorithm to schedule lectures on the middle of the day. Instead of giving strictly positive penalties this representation seems more intuitive since you want to *punish* early and late lectures, and *reward* lectures that are held during the day.

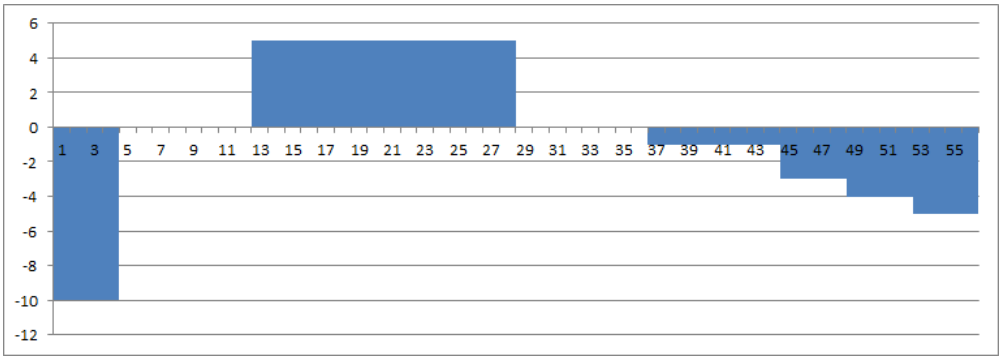

Figure 16: The quarter penalty function used in the experiments. Every quarter maps to a quarter of an hour between 8:00 and 22:00

The BufferPenalty is exactly the same as described in section 3.2. At every moment in time there should be 7 empty rooms to accommodate unforeseen incidents. For instance, if there are only 3 empty rooms of a certain type during a 1 hour lecture, then the Empty Room penalty is equal to: $(7-3)^2 \times EmptyRooms = 4^2 \times 0.5 = 8$.

Utrecht University does not have data to create dependencies between the lectures, so these had to be generated. The dependencies are randomly generated by algorithm 7. In this algorithm the dependency of lecture l is defined by picking a random lecture m (m is part of the same course as l). If the addition of a dependency between l and m does not create a cycle in the dependency list (otherwise l depends on m and m depends on l, which causes a deadlock) the time-window of this dependency is calculated. When the duration and lecture type are the same and the groupnumbers differ (line 8 of algorithm 7), then the minimum and maximum time are set to zero. This is done to create parallel meetings. For instance, the two workgroups of a course (they have same duration and lecture type, but different group number) should start at the same time; this is a common demand from the teachers.

If this is not the case then in 40% of the cases the minimum time of the time-window is 14 hours and the maximum of the time-window is 42 hours. Note that these numbers could easily be adjusted; these values are only taken as an educated guess. In the other 60% of the cases the dependency between lecture l and m is dropped.

Algorithm 7 Pseudo code of way dependencies are generated

```
1:  procedure GENERATE DEPENDENCY(Lecture l)
2:      LecturePrev ← GetRandomLecture(l.Course)
3:      if Cycle(l, Prev) then
4:          return false
5:      end if
6:      l.Previous = Prev
7:      Chance ← RNG.Random()
8:      if Prev.duration = l.duration & Prev.type = l.type & Prev.group ≠ l.group then
9:          l.MinimumTime ← 0
10:         l.MaximumTime ← 0
11:     else if Chance < 0.4 then
12:         l.MinimumTime ← 56
13:         l.MaximumTime ← 168
14:     else
15:         l.Previous = null
16:         return false
17:     end if
18:     return true
19: end procedure
```

And finally, the penalty for scheduling in an external room is 200, and the penalty for leaving a lecture unscheduled is 400. A summary of all the penalties is given in table 10.

The initial temperature and cooling-down rate for the simulated annealing are found during a trial-and-error process, using the equations described in section 4.1.5 to find good values for these parameters. Throughout the experiments the initial temperature is 111.0 with a cooling-down rate of 0.9999299. Recall that the temperature is decreased every iteration (of the 50.000 iterations) in the search process.

Name of the penalty	Penalty name in section 3.2	Value
Quarters of an hour: 0-3	QuarterPenalty	-10
Quarters of an hour: 4-11	QuarterPenalty	0
Quarters of an hour: 12-27	QuarterPenalty	5
Quarters of an hour: 27-35	QuarterPenalty	0
Quarters of an hour: 36-43	QuarterPenalty	-1
Quarters of an hour: 44-47	QuarterPenalty	-3
Quarters of an hour: 48-51	QuarterPenalty	-4
Quarters of an hour: 52-55	QuarterPenalty	-5
Use of external room	RoomPenalty	-200
Leaving a lecture unscheduled	UnscheduledPenalty	-400
Empty rooms penalty	EmptyRooms	-0.5

Table 10: An overview of the penalties used in the implementation of this thesis. Note that RoomPentaly and UnscheduledPenalty are independent of the duration of a lecture; the other penalties do depend on the duration. With these values the objective function can be positive as well as negative.

5.2 Comparing functions

As described in section 4.1.2 there are two room comparing functions and two lecture comparing functions:

- Roomsize comparer
- Roompopularity comparer
- Lecturesize comparer
- Hard to schedule comparer

The tests involve ten different runs of each combination of comparing functions, including the lexicographic sorting where no specific sorting function is used. The operators used in these runs are the base-operators mentioned in section 4.3. The local search was converged after approximately 50.000 iterations for the stamp, and 50.000 iterations for the scheduling of the incidental lectures. The global search process of this experiment can be found in figure 17.

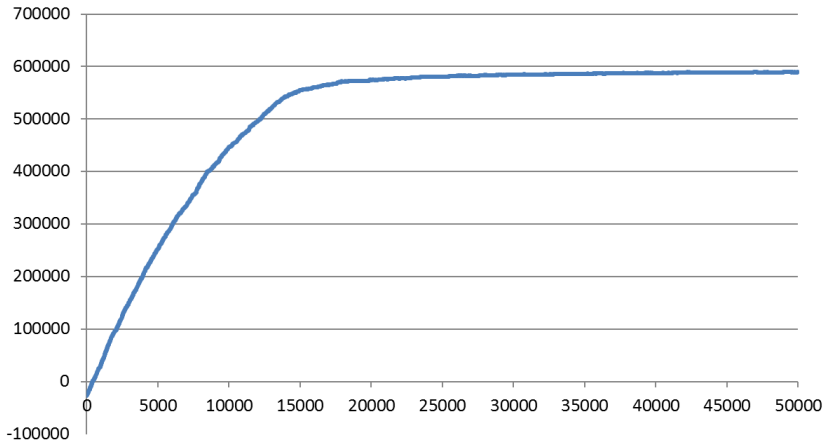

Figure 17: The progression of the objective function of a randomly picked search process.

As can be seen the most progress is done in the first 20.000 iterations. However, the timetable does improve afterwards, although it is only slightly. To make sure each search process is fully converged, the limit of 50.000 iterations is retained.

The results of this experiment are shown in table 11. As it turns out, some combinations of comparing functions prove to be better than others. Since the finishing values of these runs vary significantly among the different combinations, this leads to the conclusion that the simulated annealing is influenced by the different sorting methods used for the generation of the initial timetables.

Lecture Comparer	Room Comparer	Stamp avg start	Stamp avg finish	Total avg start	Total avg finish
Hard to Schedule	Room Popularity	-33012,5	-194,5	-78709,4	554666,7
Hard to Schedule	Room Size	-31594,8	-638,9	-114804,8	506681,9
Hard to Schedule	-	-54579	-1947,1	-107956,8	547146,6
Size	Room Popularity	-36692,8	-125,0	-30737,1	587736,5
Size	Room Size	-36718,8	-2455,7	-71234,8	544450,0
Size	-	-59883,5	-5614,0	-83315,6	555267,2
-	Room Popularity	-52431,9	-1481,9	-124444,3	531458,4
-	Room Size	-36111,4	-111,0	-87851,9	546433,9
-	-	-61118,5	460,4	-126054,7	550625,6

Table 11: The averages of ten runs of the different combinations. The average score of the initial solution of the stamp corresponds with 'Stamp avg start' and 'Stamp avg finish' corresponds with the average score of the timetable after the simulated annealing. 'Total avg start' is the average score of the initial total timetables and 'Total avg finish' corresponds with the total timetables after the simulated annealing (these timetables are the outcome of the algorithm).

The first thing that catches the eye are the scores where no lecture comparing function is used: these are not as bad as one would expect. In fact, some combinations of lecture + room comparing functions are even worse than the timetables generated without a lecture comparing function (for instance Hard to schedule + Room Size); when no comparing function is used, the order in which the input is delivered is adopted, which is a lexicographic ordering.

Looking at the results it is hard to declare an absolute winner in the Lecture Comparer and Room Comparer category separately. It is the *combination* of the two comparing functions that create an initial timetable for the local search. However, the Lecture Size comparing function seems to perform well with all the room comparing functions.

The great winner of these experiments is the combination of the lecture-size comparing function with the Room Popularity function. This combination of comparing functions produces initial solutions that work best for the local search. To further prove this statement, the best 10 results found in this experiment are shown in table 12; note that 9 out of the 10 best results are produced with the Size Comparer and Room Popularity function.

Score	Lecture Comparer	Room Comparer
605000,5	Size	Room Popularity
599193	Size	Room Popularity
598556,5	Size	Room Popularity
597477,5	Size	Room Popularity
589904,5	Size	Room Popularity
582915,5	Size	Room Popularity
579198,5	Hard to Schedule	-
578152,5	Size	Room Popularity
576364	Size	Room Popularity
575160	Size	Room Popularity

Table 12: The best 10 results of the first experiment.

5.3 Effect of local search

In section 4.1 is explained that first a *stamp* is created and optimized to create a timetable for all the regular lectures. This stamp is then imported in the total lecture to schedule the incidental lectures. The workflow is as follows:

1. Create an initial stamp timetable with a greedy algorithm
2. Optimize the stamp timetable using local search
3. Import the stamp into the complete timetable
4. Schedule the incidental lectures into the complete timetable with a greedy algorithm
5. Optimize the total timetable using local search

Since no local search method is used in Syllabus Plus, the influence of a local search on the timetabling problem is an interesting subject. In this experiment the influence of the local search is measured by leaving it out at step 2 and/or 5. The results are shown in table 13. In this table the effect of the local search is clearly visible: these findings underline the superiority of local search over a greedy algorithm.

Stamp LS (step 2)	Total LS (step 5)	Average score
FALSE	FALSE	-125781,5
FALSE	TRUE	509251,7
TRUE	FALSE	-31698,7
TRUE	TRUE	587736,5

Table 13: The average results of the experiments by using or leaving out certain local search parts. The greedy algorithm is based on the compare functions *Size comparer* and *Room popularity*; the base-case comes from the previous experiment, see table 11.

5.4 Branch and Bound Heuristic

In this section the influence of a Branch and Bound heuristic is discussed. As mentioned in section 4.3.1 the choice of the parameters is crucial for the running time: allowing a high branching factor (number of spots) with a high depth (number of lectures) results in a massive searchtree with the amount of nodes in the order of $\#Spots^{\#Lectures}$. Therefore five different

setups are looked into; in each setup a different amount of $\#Spots$ and $\#Lectures$ is used. See table 14 for the details of the setup and the corresponding results.

- Case 1: This is the baseline measurement; the results are from table 11.
- Case 2: In this experiment the searchtree of the Branch and Bound heuristic is biggest. Because of the size of this searchtree, the running time of the algorithm increased tremendously. Therefore the total number of iterations of each Branch and Bound run was restricted to keep the running time reasonable: each run could only explore 15.000 nodes in the searchtree of Branch and Bound. Even with this restriction this setting of the Branch and Bound heuristic led to a running time that was three times longer than the running times of the experiments in section 5.2. The fact that the Branch and Bound heuristic could not complete the search is clearly visible in the results: the average finishing value is only slightly better than the average finishing value of case 1 (the baseline measurement). The conclusion of this test-case: the addition of the Branch and Bound heuristic used on a relative large group of lectures does not outweigh the increase in running time.
- Case 3: The Branch and Bound in this experiment was also restricted to 15.000 iterations, just as in Case 2. The running time of this experiment is slightly shorter than that of Case 2, but the quality of the timetables is worse than those produced in the second case.
- Case 4: The restriction on amount of nodes that can be explored in the Branch and Bound heuristic was dropped in Case 4. The Branch and Bound heuristic with 5 lectures and 10 spots can be called in 2% of the cases to maintain a reasonable running time. However, the effect of the Branch and Bound heuristic is still not significantly better than the base case.
- Case 5: Preliminary results indicate that the Branch and Bound heuristic does not always find a better solution for the group of lectures, so apparently it is used on groups of lectures that are already optimal (for the selection of lectures). In these cases the Branch and Bound algorithm does not alter the timetable. Therefore, it might be beneficial to run the Branch and Bound heuristic more often. To keep the running time reasonable the searchtree of the Branch and Bound is even more reduced by decreasing the amount of lectures and spots. In order to compensate the reduction of time spent on the Branch and Bound heuristic, the number of times that the Branch and Bound heuristic is used could be tripled. As can be seen in the results, this improves the solution significantly.
- Case 6: In this case the search trees of Branch and Bound were kept even smaller. It seems that further reduction in treesize is not beneficial for the quality of the timetables, even though the searchprocess took more time than the runs of Case 5.

The results of the Branch and Bound heuristic are somewhat disappointing. In the smaller test cases it does have a positive effect on the produced timetables, but the increase in running time in the larger test cases is unfortunate. The size of the searchtree exponentially increases when the amount of lectures and spots are increased, eliminating the opportunity to experiment with larger groups of lectures. However, the contribution of a Branch and Bound heuristic on smaller instances produces significantly better timetables so it is still good way to improve the quality of the timetables.

Case	Lectures	Spots	Usage	Runtime	Total avg start	Total avg finish
1	-	-	-	10 min	-30737,1	587736,5
2	7	14	1%	30 min	-20510,5	598403,2
3	6	12	1%	24 min	-31748,7	590053,2
4	5	10	2%	12 min	-27922,1	591859,3
5	4	8	6%	12 min	-14436,2	611322,3
6	3	6	12%	18 min	-20420,7	595352,2

Table 14: Test results of the branch and bound experiments. Preliminary results show that the 1:2-relationship between Lectures and Spots is a good one. The 'Usage' indicates how often the Branch and Bound heuristic is used in the local search.

5.5 Combination of operators

In this final experiment the different operators mentioned in section 4.3 are combined to produce the best timetables. Preliminary results indicate that the following setup produces the best results:

- The shift function is applied on every room every 1000th iteration of the local search.
- Preliminary results indicate that the Chain Reschedule operator can be quite destructive. Therefore this operator is used with only a 1.0%-chance.
- As the results in table 14 suggest, the Branch and Bound heuristic is used on 4 lectures with each 8 different spots. This operator is called in 6% of the cases.
- The local search tries to schedule an unscheduled lecture in 20% of the cases. The high cost of leaving a lecture unscheduled justifies the high percentage of usage.
- In the other cases the algorithm uses the shuffle operator. This remains the main operator of the algorithm.

This setting results in an average score of 653.674,7, with a standard deviation of 12.806,8. To set these results in the context of the current system:

- Using the best greedy method resulted in a score of -125.781,5.
- The best simple local search method resulted in an average score of 587.736,5.
- The absolute best timetable found by the advanced local search method resulted had a score of 668.974,5.

The greedy method clearly cannot deal with this highly constrained problem. A simple local search with only a few problem specific operators proves to be a lot more effective than a mere greedy search. The addition of more sophisticated operators increases the quality of the timetables even further.

6 Conclusion

Utrecht University does not have a fully-automated system to schedule the lectures into a timetable. The algorithm discussed in this paper offers the opportunity to do that. It starts by the data analysis as described in section 2. The essence of the problem was captured by looking for similar properties and filtering out redundant information. Using this boiled down data offered the search algorithm handles to operate.

Even when this algorithm is not implemented at Utrecht University it is still advisable to apply the data purification of section 2, since most changes only affect the front-end of the system. Updating the way the information is digitized would require relatively little work, while the data becomes much cleaner. The introduction of the roomsets is in line with the standardization of the rooms, which is a goal of Utrecht University. Just like the introduction of the timeslots will the introduction of roomsets lead to an improvement of efficiency and clearance.

The Branch and Bound as described in section 4.3.1 sounds promising and it managed to achieve improvements, but the increase in running time makes this implementation of Branch and Bound less applicable. Further research could be done to increase the gains of the Branch and Bound heuristic by refining the way the lectures are selected so that the success-rate of the Branch and Bound operator is increased.

Together with the introduction of the Chain Reschedule and Shift operators the best results were found, where also the use of different stamp-solution proved to have a positive effect on the quality of the timetables. Although some minor details are left out and the input-data is slightly molded to serve as good input-data for the local search, this algorithm still proves to be a promising alternative of the current system, since this algorithm *automatically* produces a timetable. This not only facilitates the workload of the schedulers, it also enables the running of simulations. A simulation could involve the demolition/construction of buildings or rooms to see what the effect is timetable-wise. Instead of having employees to create a fictional timetable that probably won't be used, this algorithm can be used to easily see the effects. Not only the automatic generation of the timetables is a great improvement, also the introduction of the fully adjustable dependencies makes this algorithm a considerable option for Utrecht University to replace its current system, since the current system is unable to model these dependencies. Also, it is proven that a local search clearly outperforms even the best greedy algorithm, where it is unknown if Syllabus uses a sophisticated greedy search or an even simpler and more straightforward search.

References

Barham, A., & Westwood, J. (1978, November). A simple heuristic to facilitate course timetabling. *The Journal of the Operational Research Society*, *29*(11).

Birbas, T., Daskalaki, S., & Housos, E. (1997, December). Timetabling for greek high schools. *The Journal of the Operational Research Society*, *48*(12).

Burke, E. K., Newall, J. P., & Weare, R. F. (1996). *A memetic algorithm for university exam timetabling.*

Burke, E. K., & Petrovic, S. (2002). Recent research directions in automated timetabling. *European Journal of Operational Research*, *140*, 266–280.

Colorni, A., Dorigo, M., & Maniezzo, V. (1997). *Metaheuristics for high-school timetabling.*

Cooper, T. B., & Kingston, J. H. (n.d.). The complexity of timetable construction problems. In *In burke and ross [br96* (pp. 283–295). Springer-Verlag.

Doulaty, M., Feizi Derakhshi, M. R., & Abdi, M. (2013, June). Timetabling: A state-of-the-art evolutionary approach. *International Journal of Machine Learning and Computing*, *3*(3).

Duong, D. X., & Dien, P. H. (2008). *Solving the lecture scheduling problem by the combination of exchange procedure and tabu search tecniques* (Unpublished master's thesis). Hanoi Institute of Mathematics.

El-Jazzar, H. S. (2012). *Three-phase approach for curriculum-based course timetabling problem* (Unpublished master's thesis). Lebanese American University.

Johnson, D. (1990). Timetabling university examinations. *The Journal of the Operational Research Society*.

Johnson, D. (1993, May). A database approach to course timetabling. *The Journal of the Operational Research Society*, *44*(5).

Kalender, M., Kheiri, A., Özcan, E., & Burke, E. (2012). *A greedy gradient-simulated annealing selection hyper-heuristic* (Unpublished master's thesis). Yeditepe University.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Lukáč, M. (2013). *Course timetabling at masaryk university in the unitime system* (Unpublished master's thesis). Masaryk University.

Mansour, N., Isahakian, V., & Ghalayini, I. (2011). Scatter search technique for exam timetabling. *Applied Intelligence*.

maw Chen, R., & fang Shih, H. (2013). *Solving university course timetabling problems using constriction particle swarm optimization with local search.*

Müller, T. (2005). *Constraind-based timetabling* (Unpublished doctoral dissertation). Charles University Prague.

Mumford, C. (2008). A multiobjective framework for heavily constrained examination timetabling problems. *Ann Operations research*, *180*, 3–31.

Pongcharoen, P., Promtet, W., Yenradee, P., & Hicks, C. (2007). Stochastic optimisation timetabling tool for university course scheduling. *International journal of production economics*, *112*, 903–918.

Rudová, H., Müller, T., & Murray, K. (2011). Complex university course timetabling. *J Sched*, *14*, 187–207.

Sánchez-Partida, D., Martínez-Flores, J. L., & Olivares-Benítez, E. (2013). Modeling and solving a timetabling problem considering time windows and consecutive periods. *Lecture Notes in Management Science*, *5*, 25–32.

Socha, K., Knowles, J., & Sampels, M. (2002). A max-min ant system for the university course timetabling problem. In *in proceedings of the 3rd international workshop on ant*

algorithm, ants 2002, lecture notes in computer science (pp. 1–13). Springer-Verlag.

Tripathy, A. (1980). A lagrangean relaxation approach to course timetabling. *The Journal of the Operational Research Society*, *31*, 599–603.

Weng, F. C., & Asmuni, H. b. (2013). An automated approach based on bee swarm in tackling university examination timetabling problem. *International Journal of Engineering and Computer Science*, *13*(2).

Wood, J., & Whitaker, D. (1998). Student centred school timetabling. *The Journal of the Operational Research Society*, *49*(11), 1146–1152.