TimetableScheduler

Solving a university timetabling problem by integer programming

June 2014

University of Southern Denmark
Department of Mathematics and Computer Science

Individual study activity by
Christian Funder Sommerlund
zero3@zero3.dk

Supervisor
Marco Chiarandini
marco@imada.sdu.dk

Individual study activity description

The university timetabling problem considers how to schedule university courses in such a way that various constraints with regard to students, teachers, time and other factors are respected. The Faculty of Science at University of Southern Denmark currently assembles these timetables manually, only assisted by tools to visualize the process and detect conflicts in the proposed timetables. This approach is inadequate with regard to timetable quality and time investment. The goal of this individual study activity is to generate timetables with mathematical guarantees with regard to the specified goals by developing an automatic system capable of transforming the concrete problem to an instance of an integer programming problem and solving it. The automatic system should take input in a human readable data format and produce output in a human readable output format, allowing usage and integration by users with little to no knowledge of computational optimization.

More concretely, in this activity we will:

- Formalize the timetabling problem at the Faculty of Science
- Model the problem as an integer linear programming problem
- Develop an automatic system capable of solving the model
- Collect, organize and import test data from the faculty
- Produce test results using the automatic system with the test data
- Discuss the test results

Contents

1 Introduction

1.1 Problem description

The timetabling problem at the The Faculty of Science at University of Southern Denmark considers how to schedule the teaching of university courses during a semester. A semester contains a number of consecutive weeks, each containing a number of consecutive days, which in turn contains a number of consecutive hour-long timeslots. A course contains a set of student groups as well as a set of sessions for each week in the semester. A student group contains a set of students. A session has a length (in number of consecutive timeslots), a type (lecture session, exercise session, ...), optionally a required room and is taught by a set of staff to a set of student groups. A person may have a role of student for some sessions while having a role of staff (as teaching assistant, for example) for other.

To solve the problem, one needs to find an assignment of sessions to timeslots each week. This must be done in such a way that no hard constraints are violated while simultaneously obtaining the best possible solution quality according to a number of soft constraints.

At the faculty, a semester usually contains around 20 weeks of teaching, Monday through Friday, and occasionally Saturday. Most sessions are currently scheduled during the timespan from 08:00 to 16:00, but teaching from 16:00 to 18:00 is not uncommon. There are no breaks in the timetable structure besides implicit academic quarters[1]. A course typically always have a student group containing all students attending the course for which lecture sessions are scheduled. The presence of additional student groups varies from course to course, but the common approach is to split the students into subgroups of approximately 30 students for exercise and lab sessions. Lecture and exercise sessions are typically 1-2 hours long while lab sessions range from 2-6 hours in length. There is a significant number of students employed as teaching assistants for courses they do not currently attend themselves. The assignment of sessions to rooms is not considered, except to ensure that sessions which require specific rooms (such as laboratories) do not overlap. The timetabler is expected to assign rooms to the remaining sessions post-scheduling.

1.2 Solution method

We show how to model the problem as a mathematical optimization problem. More specifically, we consider consider how to solve the week version of the problem using integer programming. In the week version, one schedules timetables for each week independently. We solve this version to exploit the fact that our problem has a natural week structure already, with almost no dependencies between weeks. There does exist a single dependency though, which is timetable stability. It is preferred by students and staff that sessions within a course are scheduled at identical times across weeks in order to minimize weekly timetable changes and ease human memoization. We implement such a stability constraint by providing the model with access to solutions for previous weeks. This essentially turns the solution process into a so-called online algorithm which finds a solution given solutions for previous weeks. By taking this approach, we are able to obtain mathematically guaranteed optimal solutions for individual weeks in sub-minute computation times on test data (Section 3). Should one have significantly more computation time available, one could somewhat trivially transform our model into a semester version to obtain an optimal solution across all weeks.

Most of our model is designed from a student perspective, in the sense that weights, penalties and the like are applied per student. One example are the penalties applied for overlapping sessions (Section 2.4.1) which are determined by the number of affected students (and their weights) and not some arbitrary penalty alone.

[1] An academic quarter is an implicit 15 minutes long transition period at the start of every hour during which no teaching takes place.

Students are assigned weights to support a number of special cases, and most importantly, to support curriculum-based timetabling. The current business logic at the faculty dictates that timetables must be scheduled before the number of students and their course attendance are known. While it may be possible for the faculty to improve on this process in the future (Appendix A), the timetabler must currently estimate the number of students per curriculum and schedule timetables accordingly. By letting students have a weight in our model, we are able to trivially support this business logic without sacrificing the support for enrolment-based timetabling. To take advantage of this, one simply specifies a single student for each curriculum and assign weights according to the expected number of students. This approach allows for a hybrid of curriculum-based and enrolment-based timetabling, easing the transition for the faculty should their business logic change towards last mentioned.

1.3 Notation

We provide textual representations of parameters, variables and constraints which only assume basic knowledge of integer programming. We also provide mathematical representations of constraints, some whose notation ought introduced.

We declare parameters and variables (mainly in Section 2.1 and 2.2) as mathematical *sets* and *functions*. We say that a set is *ordered* when its elements have some order which they are returned in upon iterating through it. We denote a *2-tuple* (a, b) (i.e. an ordered pair containing the elements a and b) such that the statement $(a, b) \in C$ means the existence of the tuple (a, b) in the set C. We make use of tuples of larger degrees as well.

We use *universal quantification* when describing constraints mathematically. We thus write $\forall a \in A,\ \forall b \in B :$ when the succeeding statement should be applied for every combination of element $a \in A$ and element $b \in B$.

When declaring variables, we use the shorthand notation $a \in \{0, 1\}$ to declare that variable a is a binary variable. We use the shorthand notation $a \geq 0$ to declare that variable a is non-negative. While most of our non-negative variables are of integer nature, we do not explicitly declare them so.

2 Integer programming model

2.1 Parameters

In order to design our model, we need to make various parameters available. Commonly used parameters are defined here, while constraint-specific parameters are defined under their respective sections below.

Name	Long name	Description
S	`Sessions`	The set of sessions to be scheduled. Let S_r be the subset of sessions requiring room r. Let S_p be the subset of sessions attended by person p (as student or staff alike).
P	`Persons`	The set of persons attending sessions in S as students and staff alike. Let P_s be the subset of persons attending session s.
PW	`PersonWeight`	A function, taking a person as input, returning the weight of the specified person.
D	`Days`	The ordered set of days available for scheduling each week.
T	`Timeslots`	The ordered set of timeslots available for scheduling each day. Let T_s be the subset of timeslots valid as starting timeslots for session s (i.e. such that the scheduling of s does not extend past the last timeslot of the day).
R	`Rooms`	The set of rooms specifically required by sessions in S.

2.2 Assignment variables

In order to design an integer programming model of the problem, one needs to consider how to model the assignment variables representing assignment sessions to timeslots during the week. We limit ourselves to consider formulations using binary variables, as commonly seen in the field. Two intuitive approaches then come to mind. First, and perhaps most intuitively and commonly seen, is the approach of creating an assignment variable for every combination of session and timeslot. Such a variable taking a value of one indicates that a session is scheduled to take place at a timeslot while a value of zero indicates the opposite. The number of ones for a session are constrained to equal the length of the session, and the ones are constrained to be scheduled in succession of each other. The second approach uses an identical set of variables, but a value of one instead represents the starting timeslot of a session. Every session is thus constrained to have exactly one assignment variable with a value of one.

The first approach has the advantage of simpler constraints. Consider a basic constraint enforcing that timetables may not contain conflicts with respect to some resource (student, room, ...). The redundancy of ones in this approach allows the constraint to consider the assignment variables of conflicting sessions for a given timeslot with no regard to the starting time or length of a session. This is not equally trivial in the second approach where constraints need to tediously consider all possible starting times that would involve a timeslot.

Based on solve time experiments, we model our problem using the second approach, in which only the starting timeslot of a session is scheduled, and the usage of any consecutive timeslots are deduced from the length and starting time of the session. Variables for invalid combinations of sessions and timeslots are not included, i.e. those with starting times such that the session would extend past the last timeslot of the day. We also provide a support set containing indices for various practical subsets of the set of assignment variables.

Name	Long name	Description
x		The set of binary session assignment variables. For every session in S, it contains a variable for every timeslot in T_s for every day in D. Let x_{sdt} be the variable specifying whether session s starts at timeslot t during day d.
V	`AssignVars`	Support set containing sets of indices of x. Let V_s be the set of indices for session s. Let V_{sd} be the set of indices for session s during day d. Let V_{sdt} be the set of indices for session s where its scheduling covers timeslot t during day d.

2.3 Hard constraints

Hard constraints represent timetable requirements that a solution must respect unconditionally. It is thus important to only model constraints as hard constraints when violations of them are completely unacceptable. Violating this principle might force the solver to break many soft constraints to generate any solution at all, leading to poor results. For this reason, only a few selected timetable requirements are modelled in this way.

2.3.1 Enforcing sessions to be scheduled

All sessions must be scheduled exactly once and thus we do not allow sessions to remain unscheduled in case the model is otherwise infeasible. We expect the timetabler to solve the issue by modifying the input data (for example by moving some sessions to other weeks) and performing a new solve instead.

Reference
S is `Sessions`
V is `AssignVars`

Constraints

$$\forall s \in S : \sum_{i \in V_s} x_i = 1 \tag{1}$$

2.3.2 Preventing staff conflicts

Timetables must not have staff conflicts. This means that no person may be assigned to more than one session for which they have a staff role at any time.

Some persons, like teaching assistants, may have a role of staff for some sessions and a role of student for other. Conflicts between their staff sessions are covered by this constraints, while conflicts between their staff and student sessions are handled by the soft version of this constraint (Section 2.4.1) which covers all attended sessions.

Reference	
P	is `Persons`
D	is `Days`
T	is `Timeslots`
SS	is `StaffSessions`
V	is `AssignVars`

Let `StaffSessions` be a set, indexed by person, containing the subset of sessions for which a given person has a role of staff.

Constraints

$$\forall p \in P,\ \forall d \in D,\ \forall t \in T: \sum_{s \in SS_p} \sum_{i \in V_{sdt}} x_i \leq 1 \tag{2}$$

2.3.3 Preventing room conflicts

Rooms only support the teaching of one session at a time, and violations hereof is not an option.

Reference	
D	is `Days`
T	is `Timeslots`
R	is `Rooms`
S	is `Sessions`
V	is `AssignVars`

Constraints

$$\forall r \in R,\ \forall d \in D,\ \forall t \in T: \sum_{s \in S_r} \sum_{i \in V_{sdt}} x_i \leq 1 \tag{3}$$

2.3.4 Enforcing session time blacklisting

Specific sessions might not be schedulable at specific times during the week for various reasons. These include staff unavailability, laboratory rules and other session specific conditions.

Reference	
S	is `Sessions`
BT	is `BlacklistedTimes`
V	is `AssignVars`

Let `BlacklistededTimes` be a set, indexed by session, of 2-tuple elements containing day and timeslot of the blacklisted starting times of a given session.

Constraints

$$\forall s \in S: \sum_{(d,t) \in BT_s} \sum_{i \in V_{sdt}} x_i = 0 \tag{4}$$

2.3.5 Enforcing session time whitelisting

Specific sessions might have to be scheduled at a specific time, or one of a number of specific times, for various reasons. These include being pre-scheduled as part of a larger plan, dependencies on external resources and limited staff availability.

Reference	
WS	is `WhitelistedSessions`
WT	is `WhitelistedTimes`

It is worth noting that whitelisting constraints trivially could be transformed into blacklisting constraints by inverting the set of timeslots in question. The opposite is true as well. We prefer the availability of both versions though, and thus present formulations of both as well.

Let `WhitelistededSessions` be the subset of sessions affected by whitelisting. Let `WhitelistededTimes` be a set, indexed by session, of 2-tuple elements containing day and timeslot of the whitelisted starting times of a given session.

Constraints

$$\forall s \in WS: \sum_{(d,t)\in WT_s} x_{sdt} = 1 \tag{5}$$

2.4 Soft constraints

Soft constraints represent timetable qualities that the solver should attempt to respect, but may compromise on in order to reach the best feasible solution (or any solution at all). These constraints have objective functions which dictates a "cost" for violating them or a "bonus" for following them. These are eventually assembled into a cumulative objective function in Section 2.5. The solver finds a best feasible solution by finding one with the minimum possible objective function value.

Several values in these constraints (such as various weights) are tied directly to business logic. Whenever possible, such values are declared using constants, functions and sets but left undefined. Suitable definitions for our test data are instead presented in Section 3.3.

2.4.1 Avoiding person conflicts

Reference	
P	is `Persons`
D	is `Days`
T	is `Timeslots`
o	is `Overlaps`
V	is `AssignVars`
PW	is `PersonWeight`
PC	is `PenaltyCost`

A person should not be scheduled to attend more than one session at the same time. While this should never have to happen under normal circumstances, some exceptions exist which prevents us from modelling this constraint as a hard constraint. Most notably, a student could sign up for an unreasonable large number of courses[2] and thereby significantly degrade timetable quality for everyone, or even cause the model to become infeasible[3]. Contrary to the hard constraint preventing staff conflicts (Section 2.3.2), this constraint includes all sessions a person attends (as staff or non-staff alike).

Let `Overlaps` be a set of non-negative variables, indexed by person, day and timeslot. These will be constrained to take the number of overlaps for a given person at a given day and timeslot. Technically, they are only constrained to be *at least* as large as the number of overlaps, but since they are assigned objective function penalties as well, the optimization process will ensure that they do not take values larger than strictly needed. Let `PenaltyCost` be a constant. `PenaltyCost` will be used in the objective function to balance the weight of this constraint with regard to other soft constraints.

Variables

$$\forall p \in P,\ \forall d \in D,\ \forall t \in T: \ o_{pdt} \geq 0 \tag{6}$$

Constraints

$$\forall p \in P,\ \forall d \in D,\ \forall t \in T: \ 1 + o_{pdt} \geq \sum_{s\in S_p} \sum_{i\in V_{sdt}} x_i \tag{7}$$

Objective function

$$Z_{\text{APC}} = \sum_{p\in P} \sum_{d\in D} \sum_{t\in T} PW(p) \cdot PC \cdot o_{pdt} \tag{8}$$

[2] There exists an unfortunate history at the faculty of some students signing up for a very large number of courses which they cannot possibly all attend, even if timetabling allowed so.

[3] If a student is able to cause the timetabling system to stop working by signing up for a large number of courses, he is effectively performing a denial-of-service attack against the system, intentional or not. This is obviously undesired.

2.4.2 Avoiding undesired times

Reference	
PC	is `PenaltyCosts`
S	is `Sessions`
V	is `AssignVars`
SW	is `SessionWeight`

Even though the model may have any number of days and timeslots, all of the combinations may not have equal preference. One typically wants to avoid scheduling sessions early in the morning, late in the afternoon and during weekends. However, given that no other feasible solution is available, we may be forced to schedule during these times and thus cannot simply remove them from the model. Instead, we attempt to avoid these times by assigning objective function penalties to them.

Let `PenaltyCosts` be a set of 3-tuple elements containing day, timeslot and penalty representing the undesired times and corresponding penalties. Let `SessionWeight` be a function, taking a session as input, returning the weight of a given session. Such weight could be the sum of `PersonWeight` weights of the attendees to prefer sacrificing the placement of sessions with few attendees instead of sessions with many attendees.

Objective function

$$Z_{\text{AUT}} = \sum_{(d,t,p)\in PC} \sum_{s\in S} \sum_{i\in V_{sdt}} p \cdot SW(s) \cdot x_i \tag{9}$$

2.4.3 Prioritising session alignment

Reference	
S	is `Sessions`
D	is `Days`
UT	is `UnalignedTimeslots`
PC	is `PenaltyCost`

The faculty is required by the university to schedule 2 and 3 hour long sessions to start at offsets equal to a multiple of their lengths. In other words, if one is to schedule a session of 2 hours on a day starting at 08:00, the session may only be scheduled to start at 08:00, 10:00, 12:00, and so on.

This policy is imposed by the university in an attempt to avoid short 1 hour breaks during the day in which rooms are not booked for any activity. While one may argue that such policy may improve room utilisation under manual timetabling, it makes little sense for an automatic system that may exploit better alignments while still prioritising room utilization implicitly or explicitly. Nevertheless, we are required to adhere to the policy anyway. The policy may be violated in exceptional cases though, so we model it as a soft constraint with an appropriately high violation penalty.

Let `UnalignedTimeslots` be a set, indexed by session, containing the subset of slots for which the scheduling of a given session to start at would cause it to be considered unaligned. Let `PenaltyCost` be a constant to be used in the objective function to balance the weight of this constraint with regard to other soft constraints.

Objective function

$$Z_{\text{PSA}} = \sum_{s\in S} \sum_{d\in D} \sum_{t\in UT_s} PC \cdot x_{sdt} \tag{10}$$

2.4.4 Prioritising lunch breaks

No time is set aside for lunch breaks in the timetable structure at the faculty (Section 1.1). This sometimes leave students (and to a lesser degree, staff as well) with long days of teaching without any breaks besides academic quarters. Various factors like canteen overcrowding and the observation that the academic quarters are shared with other purposes (changing rooms, toilet visits, ...) all contribute to the desire for scheduling lunch breaks not necessarily located at the same time for everyone. Our solution is to schedule a lunch break of a single hour somewhere in the middle of each day whenever possible.

Reference	
P	is `Persons`
D	is `Days`
LT	is `LunchTimeslots`
b	is `LunchTimeBusy`
l	is `LunchDenied`
S	is `Sessions`
V	is `AssignVars`
PW	is `PersonWeight`
PC	is `PenaltyCost`

Let `LunchTimeslots` be the subset of timeslots considered suitable for a lunch break. Let `LunchTimeBusy` be a set of binary variables, indexed by person, day and timeslot in `LunchTimeslots`. These are constrained to equal 1 if any sessions are scheduled for a given person at a given day and timeslot (essentially a logical OR operator). Let `LunchDenied` be a set of binary variables, indexed by person and day. These are constrained to equal 1 if all `LunchTimeBusy` variables for a given person and a given day equals one (essentially a logical AND operator). Technically, variables in either set may obtain a value of 1 in a feasible solution even when not strictly required so by the constraints, but since the `LunchDenied` variables have objective function penalties applied to them, the solver will ensure that these will not obtain a value of 1 unnecessarily in any optimal solution found. Let `PenaltyCost` be a constant to be used in the objective function to balance the weight of this constraint with regard to other soft constraints.

Variables

$$\forall p \in P,\ \forall d \in D,\ \forall t \in LT:\ \ b_{pdt} \in \{0,1\} \tag{11}$$

$$\forall p \in P,\ \forall d \in D:\ \ l_{pd} \in \{0,1\} \tag{12}$$

Constraints

$$\forall p \in P,\ \forall d \in D,\ \forall t \in LT:\ \ |S_p| \cdot b_{pdt} \geq \sum_{s \in S_p} \sum_{i \in V_{sdt}} x_i \tag{13}$$

$$\forall p \in P,\ \forall d \in D:\ \ |LT| - 1 + l_{pd} \geq \sum_{t \in LT} b_{pdt} \tag{14}$$

Objective function

$$Z_{\text{PLB}} = \sum_{p \in P} \sum_{d \in D} PW(p) \cdot PC \cdot l_{pd} \tag{15}$$

2.4.5 Prioritising course spreading

Reference	
S	is `Sessions`
e	is `SpreadPenalties`
D	is `Days`
V	is `AssignVars`
PC	is `PenaltyCost`

The teaching of courses at the faculty usually involves 2-4 weekly sessions of various length and type. It is usually expected that students prepare themselves in-between these sessios by reading textbooks, solving exercises, work on assignments and so on. For this and other reasons, it is desirable that sessions of the same course to be spread out during the week in student timetables.

As we never consider two sessions scheduled for the same day as suitably spread out, we look at the spreading problem from the perspective of the number of days inbetween sessions. This simplifies the problem for the common case, but leaves a slightly awkward handling of the special case in which the number of sessions is larger than the number of days to spread the timetable over (per the pidgeonhole principle, there must be at least two sessions scheduled for the same day. If we apply penalties for scheduling two sessions for the same day, at least one penalty will always be applied).

Let `SpreadPenalties` be a set of non-negative continuous variables, indexed by session. These will contain the active penalties caused by the scheduling of a given session. For each day in the week, sessions are subject to a constraint responsible for pushing penalties into the corresponding penalty variable according to how other sessions are scheduled. The constraints do this by introducing an arbitrary large constant, C, on both sides of its inequality. On the left side, C is multiplied by the session assignment variable for every timeslot of the day, ensuring a sum of C if the session is scheduled during the day. On the right side, C is added as a constant

added to the `SpreadPenalty` variable for the session. Thus C effectively guards the `SpreadPenalty` variable such that it is only forced to obtain a positive value if the session is actually scheduled for the corresponding day so that it activates C on the left-hand side. At this point, all we need to do is add session assignment variables for other sessions, multiplied by appropriate weights, to the left-hand side. These penalties are then pushed to the `SpreadPenalty` variable on the right side if both sessions are scheduled at the corresponding times.

Let `PenaltyCost` be a function, taking a session, a day, a second session and a second day as input, returning the penalty for scheduling the first given session at the first given day while scheduling the second given session at the second given day. These penalties could be based on the number of attendees attending the sessions to avoid sacrificing the spreading of sessions with many attendees as opposed to sessions with few attendees.

Variables

$$\forall s \in S: \;\; e_s \geq 0 \tag{16}$$

Constraints

$$\forall s \in S, \;\; \forall d \in D: \;\; \left(\sum_{i \in V_{sd}} C \cdot x_i \right) + \sum_{s' \in S} \sum_{d' \in D} \sum_{i' \in V_{s'd'}} PC(s, d, s', d') \cdot x_{i'} \leq C + e_s \tag{17}$$

Objective function

$$Z_{\text{PCS}} = \sum_{s \in S} e_s \tag{18}$$

2.4.6 Prioritising timetable stability

It is preferred by students, staff and the faculty alike that timetables for individual weeks are as similar (stable) as possible. Without a constraint taking this into account, sessions may be placed arbitrarily each week by the solver, as long as the solution remains optimal with regards to other constraints. Timetable stability has several advantages, such as easier memorization and less difficulties with planning other recurring events. This constraint strives to ensure stability by adding objective function *bonuses* (i.e. negative values in our penalty minimization model) to assignment variables which represent stable schedulings.

Reference	
S	is `Sessions`
D	is `Days`
T	is `Timeslots`
P	is `Persons`
BB	is `BaseBonus`
PW	is `PersonWeight`
BF	is `BonusFactor`

Bonuses are determined by, for every possible scheduling of a session during a week, and for every person attending the session, inspecting previous timetables for matching sessions at the same time during those weeks. A matching session is said to have a bonus factor according to how well it matches the current session. The bonus factor is important because it allows assignment of a lesser factor to sessions from the same course but of a different type. A lecture session matching a previous lecture session may thus be assigned a maximum factor, being a perfect match, while a lecture session matching a previous exercise session may be assigned a smaller factor, being just a good match. This indicates that if a perfect match will not be honored, we still prefer selecting a time previously used by a session of another type from the same course. One interesting case covered by this approach is when a course alternates between 2/1 lecture/exercise sessions and 1/2 lecture/exercise sessions each week. With this constraint in place, we are able to express preference for reusing the same three slots, regardless of session type distribution, while still preferring a perfect match whenever possible.

Let `BaseBonus` be a constant to be used in the objective function to balance the weight of this constraint with regard to other soft constraints. Let `BonusFactor` be a function, taking a session, a day, a timeslot and a person as input, returning a factor indicating the stability of scheduling the given session at the given day and timeslot for the given person. This factor should be 1 if scheduling the given session for the given person at the given

time is considered perfectly stable, a factor between 0 and 1 if semi-stable and a factor of 0 if unstable. These factors could thus be calculated by considering previous timetables for the given person and return a factor equal to the fraction of previous timetables which contain a matching session at the given time of the week.

Objective function

$$Z_{\text{PTS}} = \sum_{s \in S} \sum_{d \in D} \sum_{t \in T_s} - \left(\sum_{p \in P_s} BB \cdot PW(p) \cdot BF(s,d,t,p) \right) \cdot x_{sdt} \tag{19}$$

2.5 Cumulative objective function

Assembling the objective functions introduced by the various soft constraints, we obtain the following cumulative objective function of penalties and bonuses to be minimized by the solver:

$$\textit{minimize} \quad Z_{\text{APC}} + Z_{\text{AUT}} + Z_{\text{PSA}} + Z_{\text{PLB}} + Z_{\text{PCS}} + Z_{\text{PTS}} \tag{20}$$

3 Test data

3.1 Background

In cooperation with the faculty, we have assembled test data covering 17 courses taught to around 500 first-year students during the spring semester of 2014, for a total of more than 1,200 sessions. This scope was mainly picked because of the existing practice at the faculty of manually scheduling this subset of courses first (i.e. before scheduling courses only taken by older students). The scheduling of this subset of courses is considered the most constrained by the faculty because of the amount of courses shared between curricula (leading to many relationships and conflicts), and thus perhaps the most interesting place to introduce an automatic system.

The test data is based on course data digitally entered by lecturers combined with informal communication between lecturers, departments and the faculty as well as some sensible default choices on unresolved matters. The data should somewhat accurately depict an actual instance of the problem at the faculty.

3.2 Input file

We have chosen to assemble instance-specific input data, such as courses and students, in an input file in JSON format. A basic example is shown in Listing 1 and is described below. For a detailed specification, we refer to the test instance input file which should be sufficiently self-documenting.

At the root, the JSON file contains a course object containing a number of named course fields and a student object containing a number of named student fields. A course field is a list of scheduling objects specifying the scheduling of one or more sessions. A scheduling object contains a numerical week number (or a list of numbers, with repetitions allowed, for scheduling several identical sessions during the same week), a numerical length, a type string, a student group string (or a list of strings) and settings for various constraints (such as whitelisting/blacklisting). Each combination of week and student group in a scheduling object represents a session to be scheduled with the specified attributes.

A student field is an object containing an attendance field and optionally other fields such as weight, to allow grouping students with identical curricula. The attendance field is a list containing attendance objects which each specify a course string and a group string (or a list of group strings) representing student group membership.

The example in Listing 1 thus represents a single student attending 2 hour long 'I' sessions (lecture sessions in faculty jargon) of a single course, twice during the weeks 6, 7, 8 and 9.

Listing 1: Simple input data example

```
{
    "courses":
    {
        "TestCourse":
        [
            {"weeks": [6, 6, 7, 7, 8, 8, 9, 9], "length": 2, "type": "I", "groups": "I"}
        ]
    },
    "students":
    {
        "TestStudent":
        {
            "attendance":
            [
                {"course": "TestCourse", "groups": "I"}
            ]
        }
    }
}
```

3.3 Model settings

We have decided on the following model settings for the problem at the faculty:

- `Days`: Monday through Saturday.
- `Timeslots`: 10 timeslots. The first one starts at 08:00 and the last one ends at 18:00.
- `Rooms`: Four different laboratories used by various lab sessions in the input data.
- Avoiding person conflicts (Section 2.4.1): A `PenaltyCost` of 32 is used.
- Avoiding undesired times (Section 2.4.2): The `PenaltyCosts` set used are shown in Table 1. The `SessionWeight` function used returns the the `PersonWeight` sum over the persons attending the session.
- Prioritising session alignment (Section 2.4.3): A `PenaltyCost` of 16 is used.
- Prioritising lunch breaks (Section 2.4.4): The `LunchTimeslots` set used are the timeslots starting at 11:00, 12:00 and 13:00. A `PenaltyCost` of 2 is used.
- Prioritising course spreading (Section 2.4.5): The `PenaltyCost` function used returns a maximum penalty of 4 scaled linearly over the desired days in-between sessions.
- Prioritising timetable stability (Section 2.4.6): A `BaseBonus` of 1 is used. The `BonusFactor` function used returns the average stability factor over the previous weeks attending the course. A factor of 1 is used for a session of same course and type at the given time, 0.75 for a different type and 0 otherwise.

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
08:00 - 09:00	2	2	2	2	2	10
09:00 - 10:00						8
10:00 - 11:00						8
11:00 - 12:00						8
12:00 - 13:00						8
13:00 - 14:00						8
14:00 - 15:00					1	8
15:00 - 16:00					1	8
16:00 - 17:00	1	1	1	1	3	9
17:00 - 18:00	2	2	2	2	4	10

Table 1: Penalty table for the avoiding undesired times constraint

4 Results

4.1 Main results

We have implemented a framework, TimetableScheduler, in the Java programming language. A brief overview is provided in Appendix B. It parses an instance of the problem from an input file (Section 3.2) for which it generates an integer programming model (Section 2) and solves it using the commercial Gurobi solver. Once the solution is proven optimal by the solver, it is returned to TimetableScheduler which exports it as human readable HTML timetables.

Table 2 shows key computational results from running our implementation with the test data we described in Section 3. All 21 subproblems were solved to optimality within a few seconds on the test machine (a laptop running Gurobi Optimizer 5.6 using 4 threads of an Intel Core i7-3630QM CPU). It is noteworthy that most subproblems were solved instantly at the root, leaving only a few to branching and bounding.

	Time to optimality	Nodes explored	Total penalties	Stability bonus	Final objective value
Week 2	~ 1 second	0	8	-0.00	8.00
Week 3	~ 1 second	0	63	-32.25	27.75
Week 4	~ 1 second	0	71	-70.62	0.38
Week 5	~ 1 second	0	74	-86.16	-11.83
Week 6	~ 1 second	0	89	-113.97	-24.97
Week 7	~ 2 seconds	436	102	-108.56	-6.56
Week 8	~ 1 second	0	90	-145.64	-55.65
Week 9	~ 2 seconds	0	19	-123.16	-104.16
Week 10	~ 3 seconds	1,699	58	-116.65	-58.66
Week 11	~ 3 seconds	270	12	-87.24	-75.24
Week 12	~ 1 second	0	5	-66.42	-61.42
Week 13	~ 1 second	0	0	-16.31	-16.31
Week 15	~ 1 second	0	0	-5.57	-5.57
Week 17	~ 1 second	0	33	-10.38	22.62
Week 18	~ 1 second	0	17	-74.11	-57.11
Week 19	~ 1 second	0	17	-72.78	-55.78
Week 20	~ 1 second	0	8	-39.89	-31.89
Week 21	~ 1 second	0	8	-38.40	-30.40
Week 22	~ 1 second	0	0	-33.47	-33.47
Week 23	~ 1 second	0	0	-8.33	-8.33
Week 24	~ 1 second	0	8	-18.17	-10.17

Table 2: Key computational results of running our implementation with the test data. Results vary slightly with each execution because the solver arbitrarily picks an optimal solution, which in turn affects the solutions of subsequent weeks because of the stability constraint. When no nodes are explored, the solution was found without branching by using cuts and heuristics.

Table 3 shows the penalty distribution in our solution. Person conflicts were avoided altogether, and the alignment constraint was only violated once during the semester. The single violation is by a three hour long session shared by all students which would otherwise incur larger penalties for violating the undesired times or lunch break constraints. The remaining penalties are caused by a large number of inexpensive violations of the remaining three constraints.

	Z_{APC}	Z_{AUT}	Z_{PSA}	Z_{PLB}	Z_{PCS}
Week 2	0	0	0	8	0
Week 3	0	37	0	10	16
Week 4	0	49	0	6	16
Week 5	0	49	0	8	17
Week 6	0	53	0	16	20
Week 7	0	58	0	16	28
Week 8	0	60	0	12	18
Week 9	0	13	0	4	2
Week 10	0	56	0	2	0
Week 11	0	12	0	0	0
Week 12	0	3	0	2	0
Week 13	0	0	0	0	0
Week 15	0	0	0	0	0
Week 17	0	17	16	0	0
Week 18	0	17	0	0	0
Week 19	0	17	0	0	0
Week 20	0	8	0	0	0
Week 21	0	8	0	0	0
Week 22	0	0	0	0	0
Week 23	0	0	0	0	0
Week 24	0	8	0	0	0

Table 3: Overview of penalty distribution in the optimal solution to the test data presented in Table 2. Z_{APC} is the constraint for avoiding person conflicts (Section 2.4.1). Z_{AUT} is the constraint for avoiding undesired times (Section 2.4.2). Z_{PSA} is the constraint for prioritising session alignment (Section 2.4.3). Z_{PLB} is the constraint for prioritising lunch breaks (Section 2.4.4). Z_{PCS} is the constraint for prioritising course spreading (Section 2.4.5).

4.2 Inspection and analysis

It is worthwhile to take a closer look at one of the tougher and more penalized weeks in our solution in order to understand the difficulties the solver had to deal with and compromise on. We have selected week 7, for which a timetable for first-year pharmacy students in class S10 is shown in Table 4.

Most of the penalties concerning this timetable centers around the lab sessions on Thursday, which incur penalties for lack of course spreading (Section 2.4.5), lack of lunch break (Section 2.4.4) and teaching during the 16-17 slot (Section 2.4.2). The two FF506 sessions Monday and Tuesday morning also incur penalties for undesired slots.

Inspecting the input file reveals that the two lab sessions on Thursday are pre-scheduled using whitelist constraints (Section 2.3.5). Further inspection reveals that all pharmacy classes have similar lab sessions, all in the same laboratory, in a pre-scheduled rotation Monday through Thursday. The teaching during the 16-17 slot is thus unavoidable. Likewise is the lack of course spreading for the second lab session. The remaining course spreading penalty and the lunch break penalty are indirectly caused by global congestion of the midday timeslots by of a number of shared sessions across curricula, namely FF503 and KE527 lectures which cannot be scheduled outside the 10-12 midday slots without causing overlap with lab sessions.

The single FF506 lecture session of the week is whitelisted to Monday and is to be attended by all pharmacy students. Because another pharmacy class has lab sessions from 10-12 and 13-16 on Monday, all possible

<table>
<tr><th></th><th>Monday</th><th>Tuesday</th><th>Wednesday</th><th>Thursday</th><th>Friday</th><th>Saturday</th></tr>
<tr><td>08 - 09</td><td rowspan="2">FF506 (I)</td><td rowspan="3">FF506 (TL)</td><td rowspan="4"></td><td></td><td rowspan="4"></td><td rowspan="10"></td></tr>
<tr><td>09 - 10</td><td rowspan="2">KE527 (TL)</td></tr>
<tr><td>10 - 11</td><td rowspan="2">KE527 (TE)</td></tr>
<tr><td>11 - 12</td><td></td></tr>
<tr><td>12 - 13</td><td>FF503 (I)</td><td>FF503 (I)</td><td rowspan="2">FF506 (TE)</td><td>KE527 (I)</td><td>FF503 (I)</td></tr>
<tr><td>13 - 14</td><td></td><td></td><td rowspan="4">KE527 (TL)</td><td>FF503 (I)</td></tr>
<tr><td>14 - 15</td><td rowspan="2">FF503 (TE)</td><td rowspan="2">FF503 (TE)</td><td rowspan="2">KE527 (TE)</td><td rowspan="4"></td></tr>
<tr><td>15 - 16</td></tr>
<tr><td>16 - 17</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>17 - 18</td><td></td></tr>
</table>

Table 4: Week 7 timetable for first-year pharmacy students in class S10. Cells contain course codes and session types with 'I' being a lecture session, 'TE' being an exercise session and 'TL' being a lab session.

schedulings would incur a penalty for either overlap or teaching during the undesired 8-9 slot (which is less undesired than overlap).

The FF506 lab session scheduled for Tuesday morning may only be scheduled at one of three possible times during a day to avoid violating the expensive alignment constraint (Section 2.4.3). These are 08-11, 11-14 and 14-17. Regardless of day, both first and last option would incur undesired slot penalties while the middle option would incur a penalty for lack of lunch break. This, and the fact that a total of 7 FF506 lab sessions using the same laboratory are scheduled for different classes this week, leaves the solver with no choice but to incur 8-9 slot penalties for some of the sessions.

These observations generalize to most of the timetables for week 3 through 8. Because a large number of lab sessions all require the same laboratory, these and other sessions are forced into suboptimal schedulings incurring a large number of minor penalties to avoid incurring expensive overlap penalties.

4.3 Model variations

With our implementation and main results in hand, we have performed some interesting experiments on the test data using variations of our model. Table 5 summarises our results.

We initially observe that the optimal solution to the base version without the stability constraint has an almost identical penalty cost as an arbitrary optimal solution of our full model (Table 2). Our implementation of timetable stability thus virtually comes for free penalty-wise.

We then perform two more experiments in which we additionally leave out the session alignment constraint (Section 2.4.3) and the lunch break constraint (Section 2.4.4) respectively. We leave the stability constraint disabled for both of these to ensure that its arbitrary nature does not impact the reproducibility of our results.

Leaving out the alignment constraint yields a ~22% reduction in penalties. Alignment penalties account for 16 penalty points in the base variation. Removing these from the calculation leads to a ~20% reduction in penalties instead. Timetable quality may thus be improved significantly if such a constraint was not present. Similarly, leaving out the lunch break constraint yields a ~22% reduction in penalties after discounting for 76 penalty points in the base variation. A similar improvement in timetable quality may thus be obtained if a lunch break slot for students and staff was not a priority.

	− stability	− stability − alignment	− stability − lunch break
Week 2	8	8	**0**
Week 3	63	63	**53**
Week 4	71	71	**65**
Week 5	73	73	**65**
Week 6	89	**81**	**69**
Week 7	102	**91**	**81**
Week 8	90	**77**	**73**
Week 9	17	**3**	**7**
Week 10	54	**50**	**48**
Week 11	12	**8**	**8**
Week 12	4	**0**	**0**
Week 13	0	0	0
Week 15	0	0	0
Week 17	33	**0**	**0**
Week 18	17	**0**	**0**
Week 19	17	**0**	**0**
Week 20	8	**0**	**0**
Week 21	8	**0**	**0**
Week 22	0	0	0
Week 23	0	0	0
Week 24	8	**0**	**0**
Total	674	**525**	**469**

Table 5: Comparison of penalties in optimal solutions for variations of our model. Minus signs indicate which constraints were left out in the variations. Bold values indicate improvement compared to the base variation without the stability constraint.

5 Conclusion

We have formalized the variation of the university timetabling problem used by The Faculty of Science at University of Southern Denmark and discussed our approach to solving it using integer programming. We then argued that our approach allows for both enrolment-based and curriculum-based timetabling as well as a hybrid of these. We have described the various constraints of the problem and presented mathematical formulations for inclusion in an integer programming model.

In order to obtain experimental results, we have assembled a test instance of the problem based on real scheduling data from the university covering ~500 first-year students attending ~1,200 sessions. Our implementation splits the test instance into weekly subproblems and solves them to optimality in a few seconds each, on a modern laptop computer using a commercial solver.

We have discussed our results, and after experiments with variations of our model concluded that timetable quality could be further improved by reducing the number of sessions requiring the same laboratory, removing of the session alignment constraint and removing the lunch break constraint.

Appendix A Timetabling process improvements

During our work, especially during the data collection phase (Section 3), we identified a number of process improvement the faculty could consider implementing in order to ease the transition to automatic timetabling as well as ensure higher quality timetables in general. We have chosen to briefly share our insights below.

It is noteworthy that the faculty recently has systematised the process in which lecturers report vital timetabling data for their courses using an online form. The scheduling of most courses are now based solely on data registered through this form. It does, however, have a number of limitations and undesired properties which several of our suggestions consider how to improve.

A.1 Enrolment-based timetabling

Timetables are currently scheduled before students enrol for courses, as discussed in Section 1.2. This is not a significant problem for the first couple of years where most students attend a predefined curriculum, but the longer a student studies, the greater the risk of becoming out of sync with ones curriculum (common reasons include having to retake previously failed courses or studying at a slower pace) and greater the share of elective courses. These situations currently cause additional post-enrolment rescheduling and in some cases significant overlap for the students. Switching to enrolment-based timetabling whenever possible could remove many of these problematic situations.

A.2 Distribution of session hours

The online form only allows lecturers to enter a single number of session hours for each type of session each week. If a lecturer wants to teach three lecture sessions of two hours during a given week, he would thus enter 6 in the appropriate field. While this approach arguably is simple to implement and use, it has the serious drawback of losing information about the number and sizes of sessions to be scheduled (did the lecturer want two sessions of three hours, three sessions of 2 hours, or something else entirely?). The timetabler currently makes an arbitrary decision about this, mainly based on experience. This kind of business logic is not easily implemented in an automatic system, so we suggest that lecturers instead specify the desired number and sizes of sessions of each type each week.

A.3 Plain text fields

The current form has a number of plain text fields in which the lecturer specifies constraints in an informal way. These include specifying room requirements, teaching free periods, desired session placements during the weeks as well as a list of courses taught by the same lecturer (used to prevent staff overlaps). Parsing the text entered into these fields automatically seems infeasible. We thus suggest removing most of the fields and instead add fields allowing lecturers to semantically specify these constraints. A single text field could still be left available for exceptional requirements that require manual handling.

A.4 Room availability issues

As mentioned in Section 4.2, the large number of lab sessions constrained to the same laboratory incur unavoidable penalties for themselves and other sessions alike. Obtaining access to more laboratories for these sessions may be able to boost timetable quality significantly. Another solution could be to move some of the sessions to other weeks.

Similarly, we showed (Section 4.3) that the alignment constraint (Section 2.4.3) degrades overall timetable quality significantly. Since removing it only benefits timetable quality, the university could consider whether its removal would be possible.

Appendix B Implementation overview

Our framework implementation in the Java programming language is organized into Java packages as follows:

Package	Description
CALLBACKS	Implementation of solver callbacks for things like logging progress and accepting suboptimal solutions if the solve progress stagnates.
CONSTRAINTS	The actual constraints of the model. Each constraint is implemented in its own class.
EXPORTERS	Solution exporters. Only a HTML exporter is currently provided.
IMPORTERS	Scheduling data importers. Only a JSON importer is currently provided.
MODEL	Contains classes representing the parameters of the problem. Importers import data into these. Also contains a class for storing solutions.
PROBLEMS	Contains main classes representing concrete problems. These are responsible for setting up importers, parameters, solver, exporters, ... and starting the solve process.
SOLVERS	Implementation of solver APIs. Only an implementation of the Gurobi Optimizer API is currently provided.
UTIL	Various utility classes.

The framework requires Java 7 or later as well as Gurobi Optimizer 5.6 or later (if using different version than the Gurobi library in the lib folder of TimetableScheduler needs to be updated correspondingly) installed.

After compilation, main classes in the PROBLEMS package may be executed directly. Alternatively, one may compile TimetableScheduler using the provided MAKEFILE which also packs the application into a .jar file that uses the PROBLEMS.NATPROBLEM class as the main class. The .jar file may be executed to solve the provided test instance as follows:

```
java -jar bin/TimetableScheduler.jar data/Y2014_Spring.json
```