Università
della
Svizzera
italiana

Faculty
of
Informatics

University timetable scheduling

Bachelor Project Report

Aron Fiechter

2018

Advisor: Prof. Dr. Michele Lanza

Assistants: Dr. Andrea Mocci, Dr. Luca Ponzanelli

Contents

Abstract

Creating and managing timetables of courses is an issue for many institutions because of the various constraints that need to be respected in the planning. This is an even harder problem if the data needed to describe the planning problem and its constraints is not well curated and stored in several different places, a situation that often leads to a long and tedious manual work in creating the timetables. In this project, we automate the process by means of state of the art tools to solve planning problems. We describe the design and implementation of a web application that offers an automated timetable creator wrapped in a user-friendly web interface.

Chapter 1

Introduction

1.1 Motivation

Creating a school timetable is a difficult problem because of the various constraints that need to be respected, such as different room sizes, instructor availability, elective courses, and different frequency of courses.
Moreover, a timetable created for a specific academic year might be partially or totally unusable the next year, because of changes in the program (e.g. a course is cancelled or moved to a different semester) or in the constraints data. This fact requires many institutions to recreate a new timetable from scratch every year.
This task is even harder if the data for the courses, rooms and all the constraints is not curated and it is stored in separate and unrelated locations. Even worse, some constraints (such as instructor availability) might arise during planning, making the whole process even more tedious, requiring further (manual) work.
Once a viable schedule is created, it is often necessary to change some details, for example by moving some lectures to different periods or different rooms. This could however violate constraints, and checking this manually, while not very hard, still takes some time, and might require moving other lectures to accommodate the initial change.

1.2 Problem

School timetabling is an NP-hard planning problem [1]. The problem consists in scheduling a set of lectures on a weekly timetable. A single lecture can vary on two variables, which are period (identifying a specific slot during the week) and room.

The search space is huge; we can calculate it using a simple formula[1]:

$$(p \times r)^l$$

where p is the number of possible periods on the weekly timetable, r is the number of available rooms and l is the number of lectures that need to be scheduled.
For example, take the spring semester of the Bachelor of Informatics at USI Lugano, which (at the time of writing) entails 14 courses, each with a specific number of lectures (calculated with respect to the ECTS weight). The total number of lectures of all courses is 70; the Faculty of Informatics has 7 rooms available, and a school week is of 40 periods (5 days, each with 8 time slots). We have:

$$(p \times r)^l = (40 \cdot 7)^{70} = 280^{70} \approx 2 \cdot 10^{171}$$

Even with constraints in place, the search space remains very large. The search space contains all possible solutions, including the optimal solution, which is not necessarily feasible[2]. There might be no feasible solutions: Consider the case of scheduling two lectures over one single possible period, in one single available room. The only solution is to schedule both lectures in the same room during that single period, which of course is not a feasible solution.

1.3 Goal

The goal of this project is to create a web application that automates the process of creating a school timetable.
The web application is an interface to a constraint solver, which lives on the back end.
Given the problem data and a set of constraints, the solver computes a good solution and returns it as a list of lectures. The solution is proposed to the user, which can then customize it by moving lectures to preferred times or rooms and locking them in place. The customized solution can then be fed back to the solver which can refine it while keeping the locked lectures in place.

[1] http://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html#searchSpaceSize

[2] according to the definitions in the OptaPlanner documentation available at http://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html#aPlanningProblemHasAHugeSearchSpace

1.4 Approach

The solver, which is the core of the application, has been developed using OptaPlanner[3], a state of the art constraint solver which provides many optimization algorithms to solve planning problems.
The interface to the solver, that is the back end of the application, is a server written in Scala[4], which exposes a GraphQL[5] API.
The front-end application is implemented in Polymer 2.x[6] and Typescript[7], and uses FullCalendar[8], a JavaScript event calendar, to visualize and manage timetables.

[3]https://www.optaplanner.org/
[4]https://www.scala-lang.org/
[5]https://graphql.org/
[6]https://www.polymer-project.org/
[7]https://www.typescriptlang.org/
[8]https://fullcalendar.io/

Chapter 2

State of the art

This chapter gives an overview of what it means to solve a planning problem (Section 2.1) and explains on a high level how the chosen solver, OptaPlanner, can be used to define such a problem and solve it (Section 2.2).

2.1 Planning problems

A planning problem[1] is an optimization problem that aims to minimize or maximize a goal using a set of resources under specified constraints. Some examples:

- ▷ Minimizing travel time on a cyclic path to visit a set of cities, visiting every city exactly once. This is widely known as the traveling salesman problem, for short TSP[2].
- ▷ Arranging n queens on an $n \times n$ chessboard so that no queen can eat any other queen.
- ▷ Assigning talks of a conference each to a time slot and a room so that several constraints are satisfied such as no overlapping, preferred rooms for some speakers, etc.
- ▷ Assigning lectures to a room and a period in a weekly timetable. This is the problem that we aim to solve in this project.

Planning problems are in general NP-hard or NP-complete. For most of these problems, a brute force approach does not work except for very small

[1]http://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html#whatIsAPlanningProblem

[2]https://en.wikipedia.org/wiki/Travelling_salesman_problem

instances. Therefore, it is common to use metaheuristic algorithms[3] to find an approximate solution which is good enough. Metaheuristic algorithms make few assumptions and have little information about the problem they're trying to solve; because of this, they can be generalized for many different problems. Some metaheuristic algorithms also use some form of stochastic optimization. These algorithms do not guarantee that the optimal solution(s) can be found, rather they usually find sufficiently good solutions.
Finding the best solution is usually not worth the computational effort needed, and approximate solutions are used instead.

2.2 OptaPlanner

OptaPlanner is a constraint solver which can solve planning problems (such as the ones described in Section 2.1) given a description of the solution of the problem together with a mechanism to compute its score. The solution contains a list of planning entities (in our case the lectures) with customizable variables. The solver optimizes the score of the solution by modifying the values of the variables. Finally, problem facts specify the constant aspects of each problem instance (e.g., courses, rooms).
The library provides a detailed guide on how to implement a solver. The solver can be configured to choose different construction heuristics (such as First Fit, First Fit Decreasing, Cheapest Insertion, etc.[4]) in the first phase of solving, which consists in creating an initial solution. This initial solution doesn't have to be optimal, and not even feasible, but construction heuristics try to get the best possible outcome in a very short time. These algorithms are usually greedy and usually run in polynomial time.
Different algorithms can be configured for the local search phase, which starts with an existing solution and tries to optimize it using local moves. Local search algorithms include Simulated Annealing [2, p. 125], Tabu Search [2, p. 154], and can be combined for increased efficacy.
It is possible to add a filter class for the planning entities, which excludes some of the entities from planning (from being modified); in our case, this is perfect to exclude locked lectures during refinement.
Other mechanisms are available, such as classes to assign a difficulty value to planning entities (so that harder entities can be initialized first during the construction heuristic phase) or to sort planning variables by weight.

[3]https://en.wikipedia.org/wiki/Metaheuristic

[4]http://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html#constructionHeuristics

Chapter 3

Approach

This chapter illustrates the core of the approach. Section 3.1 defines the course timetabling planning problem on a conceptual level. Section 3.2 gives an overview of the chosen technologies, explaining the architecture and the role of each part. Section 3.3 describes the modelling of the problem domain to be used with the constraint solver, OptaPlanner. Section 3.4 examines how the constraints have been included in the score calculation. Section 3.5 provides a brief description of the wrapper of the solver on the server side. Section 3.6 depicts the server endpoint interface that allows the client side to interact with the solver. Finally, section 3.7 describes the implementation of the web application demo.

3.1 Problem definition

The timetable scheduling problem aims to arrange a set of lectures in a weekly schedule with the goal of minimizing conflicts. The problem is defined according to the existing definition by the International Timetabling Competition[1]. Each problem instance consists of:

- ▷ a list of courses: Each course has a name, a list of instructors who teach the course, a list of semesters the course is taught in, the number of lectures the course entails, the minimum number of days these lectures should be spread out on, and the number of students registered for the course.
- ▷ a list of periods: Each period is defined by a day of the week (usually between Monday and Friday) and a time slot during the day (e.g. in

[1] http://www.cs.qub.ac.uk/itc2007/curriculmcourse/course_curriculm_index.htm

the range 0-7, with four slots of one hour in the morning and four in the afternoon).

- ▷ a list of unavailable periods: Each unavailable period points to a course and to a period, indicating that the course cannot have any lecture in that specific period.
- ▷ a list of rooms: Each room has a name and a capacity.

A solution consists of a list of lectures, each with an assigned room and a period, representing a weekly occurrence of a course in the schedule.
The problem definition also includes a list of constraints that the solution should satisfy. The constraints can be separated into two categories: hard constraints, which need to be satisfied in order for the solution to be feasible, and soft constraints, which do not necessarily need to be satisfied.
Hard constraints usually model physical limitations; soft constraints model limitations which do not violate physical reality but which satisfy specific preferences about how the solution should be.
In our case there are five hard constraints and four soft constraints, and we list them in Table 3.1 and Table 3.2.

Instructor conflict	An instructor must not have two lectures in the same period.
Semester conflict	A semester must not have two lectures in the same period.
Course conflict	A course cannot have two lectures in the same period.
Room occupancy	Two lectures must not be in the same room in the same period.
Room capacity	A room capacity should not be less than the number of students in the course.
Unavailable period	A specific lecture must not be assigned to a specific period.

Table 3.1: Hard constraints

Each constraint has an associated penalty value which is important to compute the score of a solution, often also called *fitness*.
In our case, all penalty values are negative, so the highest possible score is `0hard/0soft` (where the hard score relates to hard constraints and the soft score relates to soft constraints).
The way that penalty values for each constraint are decided can vary. For example, if the room capacity constraint is violated, each student above the

Minimum working days	Lectures of the same course should be spread out into a minimum number of days.
Semester compactness	Lectures belonging to the same curriculum should be adjacent to each other (so in consecutive periods if they are on the same day).
Room stability	Lectures of the same course (and possibly even those of the same semester) should be assigned to the same room.
Lecture pairs	Lectures of the same course should be scheduled in blocks of two.

Table 3.2: Soft constraints

capacity could count as 1 point of penalty, or we could make each student count 2, or use a standard value of 10 points of penalty, regardless of how much the room is over capacity.
The score of two solutions is compared in order: If a solution A has a worse hard score than another solution B, then A is worse than B, regardless of what the soft scores are.

3.2 Technologies

The core solver implementation makes use of a state of the art library for solving planning problems called OptaPlanner.
The server side of the web application is implemented in Scala, a relatively young general-purpose programming language, using Akka HTTP[2]. It exposes an interface to in-memory data and to the solver using GraphQL, which is a query language that allows the user to retrieve data with high precision using a single request. GraphQL is an alternative to REST that allows the client to specify the structure of the requested data, avoiding the need for further requests. The GraphQL endpoint is also implemented in Scala using Sangria GraphQL[3].
The client side is implemented as a Polymer 2.x application (with TypeScript instead of JavaScript), where Polymer is a library to build Web Components[4].

[2] https://doc.akka.io/docs/akka-http/current/
[3] http://sangria-graphql.org/
[4] https://www.webcomponents.org/

3.3 Domain model for OptaPlanner

We modelled the problem domain according to what OptaPlanner expects. The implementation is done in Scala, although for some features we had to use Java syntax; it is similar to the existing Java implementation available in the `curriculumcourse` folder available in the OptaPlanner examples[5].
The main solution class, called `Schedule`, contains all the lists of all problem facts (`Period`s, `Day`s, `TimeSlot`s, `Semester`s, `Instructor`s, `Room`s, `Course`s, `UnavailablePeriod`s), the list of planning entities (`Lecture`s) and the current solution score (`HardSoftScore`).
The diagram (Figure 3.1) representing the problem shows the solution class (`Schedule`) in green, the planning entity (`Lecture`) in red and the problem facts in blue. As said, the `Schedule` class contains lists of all lectures and all problem facts; this is not shown explicitly on the diagram. Only the relevant fields are shown.

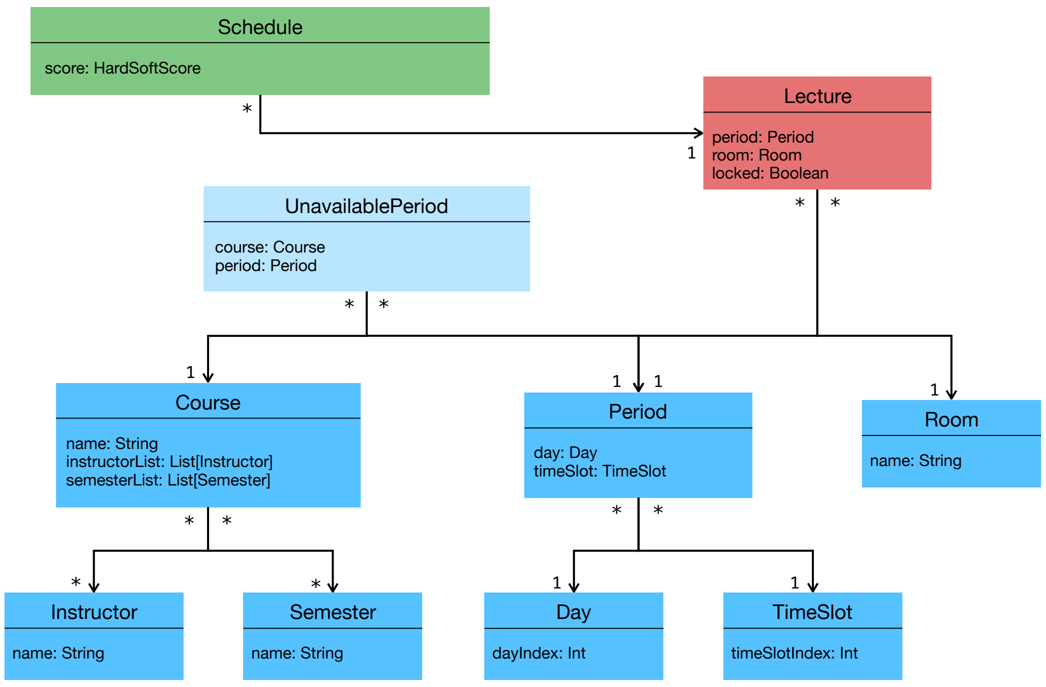


Figure 3.1: Class diagram of the domain model

[5]https://github.com/kiegroup/optaplanner/tree/master/optaplanner-examples

The `Lecture` class implements a Planning Entity, which is what the solver can mutate in order to improve solutions. The two fields that a `Lecture` can mutate over are `period` and `room` (of type `Period` and `Room`, respectively).
However, if the `locked` field of the lecture is set to `true`, the solver ignores that `Lecture` and only varies the unlocked ones. The score still takes into account all lectures, regardless of their locked value. It is important to say that a `Lecture` that is locked *must* have a non-null value for both its `room` and `period` (i.e. it must be already initialized).

3.4 Constraints

The constraints are of two kinds, hard constraints and soft constraints, as described in Section 3.1.
As a rule, any solution that violates any hard constraint is unfeasible, since hard constraints model physical limitations (such as the impossibility for an instructor to teach two lectures at the same time).
To compute the score we need to take into account all constraint violations, so we need to iterate over all lectures and combine them with data from courses, rooms and unavailable periods.
A simple way to do this would be to accumulate constraint violations while iterating over all lectures combined with the other data. The problem with this approach is that the score needs to be calculated every time a new solution is created by the solver, i.e. many times per second, because the solver can mutate the solution very fast by just swapping two lectures or moving one single lecture.
It makes much more sense, from an efficiency point of view, to remember which constraint violations did not change, and to just recompute the score with respect to the moved lecture (or swapped lectures).
To do this, we must rely heavily on maps, and the implementation can become very difficult to understand and to maintain.
Fortunately, OptaPlanner supports Drools[6], which is an engine that can automatically perform incremental score calculation given some rules. Each constraint is described as one or more rules.
An example of a rule is available in Figure 3.2, which implements the room capacity constraint (see Section 3.1. The `when` part describes the condition, which is matched when the boolean statements are true (`room == $room` and `course.studentSize > $capacity`); the condition is applied to all combinations of rooms and lectures. The `then` part applies the penalty (which is computed

[6]https://www.drools.org/

as the number of students that are above the capacity of the room), to the `Hard` part of the global score.
For most constraints it was possible to use the existing rules implemented in the OptaPlanner example, while for some others we had to apply some changes (we transformed the room capacity constraint into a hard constraint) or write the rule from the ground up (for the lecture pairs constraint; see Section 3.1).

```
// Room capacity: A room's capacity should not be
// less than the number of students in the course.
// Each student above the capacity counts as 1 point
// of penalty.
rule "roomCapacity"
    when
        $room : Room($capacity : capacity)
        Lecture(
            room == $room,
            course.studentSize > $capacity,
            $studentSize : course.studentSize
        )
    then
        scoreHolder.addHardConstraintMatch(
            kcontext,
            ($capacity - $studentSize)
        );
end
```

Figure 3.2: Rule that models the room capacity constraint

3.5 Solver architecture

The solver (implemented in `solver.SchedulerApp`) exposes a method `solve(s: Schedule): Schedule` which expects a schedule with uninitialized lectures (no room or period assigned) or partially initialized lectures (if a lecture is locked, it *must* have both a period and a room already assigned). Of course, the input data needs to be consistent: for example, no initialized lecture must be assigned to a room or a period that do not exist.
The solver is configured to run a construction heuristic phase which uses the First Fit Decreasing heuristic. The following two phases, which are local

search phases, use Late Acceptance and Simulated Annealing. The output is again a schedule, with all lectures assigned to a room and a period.
Both input and output objects of type `Schedule` refer to the `domain` implementation used by the solver, which uses simple syntax often similar to Java.

3.6 Server and GraphQL API

As said, the solver lives on the back end, and it is available via a GraphQL endpoint. The server is implemented in Scala using Akka HTTP, and has one single endpoint: both `/` and `/graphql` are ways to access it.
The server stores (during runtime, there is no persistence) all courses, buildings, rooms, as well as a map that stores created schedules by ID. The ID of a Schedule is computed based on the courses and rooms chosen for scheduling. All of this is stored as instances of Scala classes of the `model` implementation, which are different from the classes used by the solver. The server in fact contains a class `SchedulerRepo` which is responsible for the conversion between the two models when dealing with the solver.
The GraphQL interface provides queries and mutations, and they are listed in Table 3.3.

`query`	`buildings: [Building!]!`
	`rooms: [Room!]!`
	`courses: [Course!]!`
	`semesters: [Semester!]!`
	`plannerCourses: [Course!]!`
	`plannerRooms: [Room!]!`
	`plannerScheduleById(id: Int!): Schedule`
	`plannerScheduleByName(name: String!): Schedule`
	`plannerSchedules: [Schedule!]!`

`mutation`	`createSchedule(name: String!): Schedule`
	`refineSchedule(id: Int!): Schedule`
	`editLecture(id: Int!lectureArg: LectureInput!): Lecture`

Table 3.3: GraphQL Schema exposed by the endpoint

The table shows what is called a GraphQL Schema. Before the column we have the field name, after the column we have the field `Type`. The exclamation mark means that the field is not nullable, and the square brackets indicate an array type.
We follow with an example of a GraphQL query to retrieve all buildings with their name and all the rooms inside them, each with name and capacity. We

also show part of the result received from the server; note that the structure of the response matches the one of the query.

```
query {
  buildings {
    name
    rooms {
      name
      capacity
    }
  }
}
```

Figure 3.3: GraphQL query

```
{
  "data": {
    "buildings": [
      {
        "name": "Informatics Building",
        "rooms": [
          {
            "name": "SI-003",
            "capacity": 60
          },
          {
            "name": "SI-004",
            "capacity": 30
          },
          {
            "name": "SI-006",
            "capacity": 60
          }, ...
        ]
      },
      {
        "name": "Main Building",
        "rooms": [
          {
            "name": "CC-150",
            "capacity": 30
          },
          {
            "name": "CC-156",
            "capacity": 30
          }, ...
        ]
      }, ...
    ]
  }
}
```

Figure 3.4: Server response

3.7 Web application UI

We chose Polymer as a library to develop the front-end interface to use the solver. The main page features two tabs: The first tab shows a list of buildings, and the plan was to show calendars with real events for every room once a building is opened. This idea is put aside to focus on implementing the functionalities of the second tab, which is the schedule creator.
The schedule creator shows a list of created schedules, starting with none. An *add* button on the bottom right can be used to open a page that shows the list of courses and rooms selected (on the back end) for scheduling.
Creating a schedule is simple: it is sufficient to enter a name and click *create*. The home of the schedule creator can be seen in Figure 3.5.

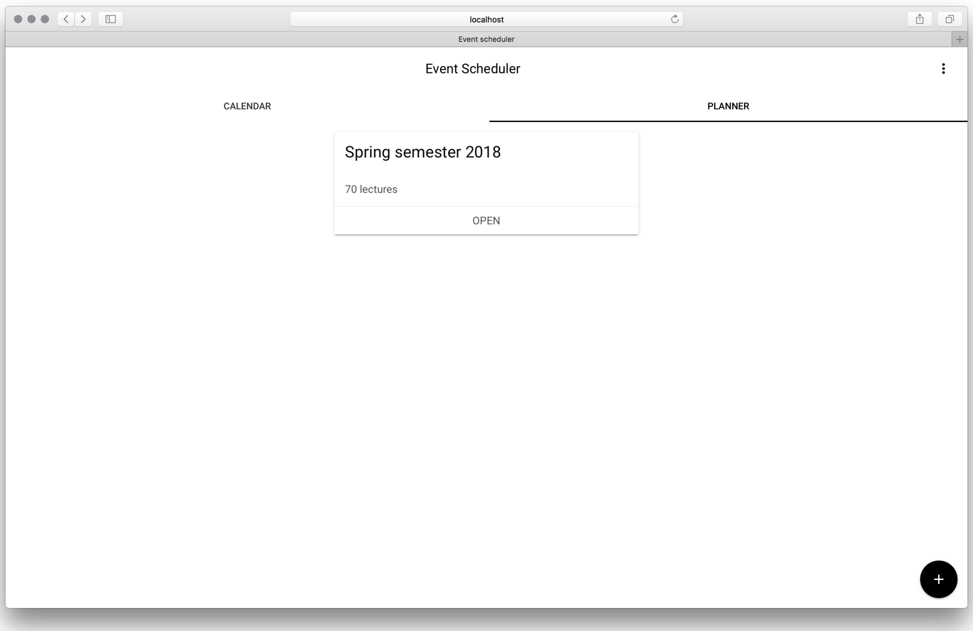
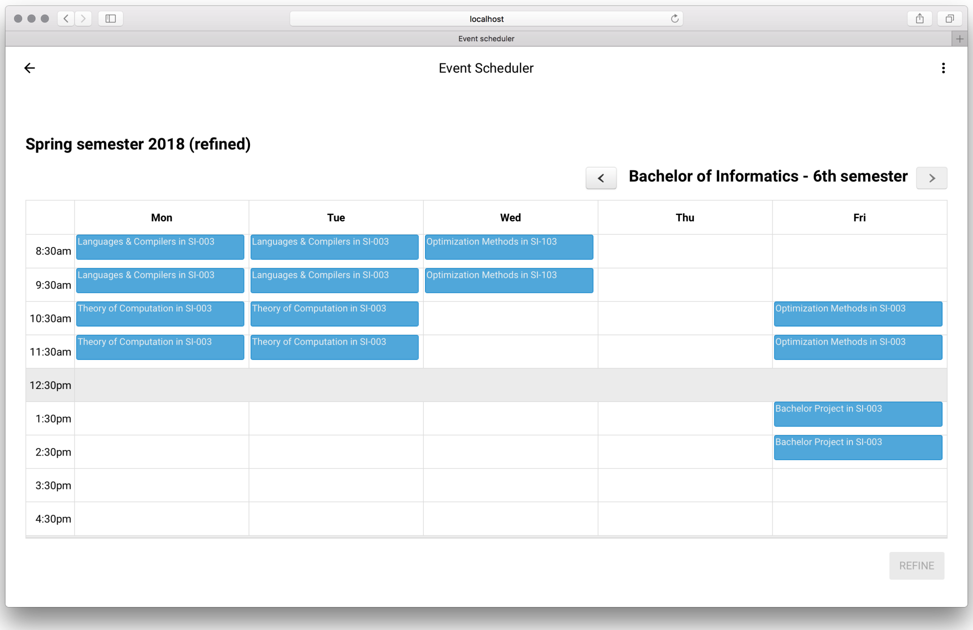


Figure 3.5: Home of the planner

By clicking on a schedule, a new view opens; on this view, the schedule is shown as a weekly timetable, separated by semester. If the schedule is the direct result of the solver, it is not possible to refine it, because the result would just be the same (see Figure 3.6). We call such a schedule *refined*.
However, if it was modified, for example if some lectures were moved and locked, it is possible to click the *refine* button (see Figure 3.7). This starts the solver on the back end using the already existing schedule; only locked lectures are preserved in place.

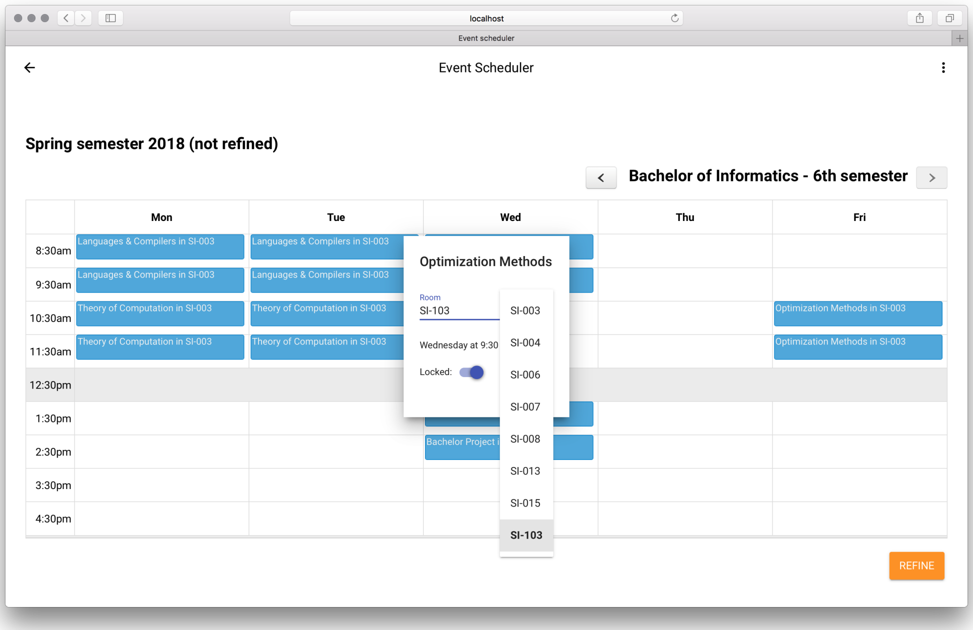


Spring semester 2018 (refined)

Bachelor of Informatics - 6th semester

	Mon	Tue	Wed	Thu	Fri
8:30am	Languages & Compilers in SI-003	Languages & Compilers in SI-003	Optimization Methods in SI-103		
9:30am	Languages & Compilers in SI-003	Languages & Compilers in SI-003	Optimization Methods in SI-103		
10:30am	Theory of Computation in SI-003	Theory of Computation in SI-003			Optimization Methods in SI-003
11:30am	Theory of Computation in SI-003	Theory of Computation in SI-003			Optimization Methods in SI-003
12:30pm					
1:30pm					Bachelor Project in SI-003
2:30pm					Bachelor Project in SI-003
3:30pm					
4:30pm					

Figure 3.6: A refined schedule



Spring semester 2018 (not refined)

Bachelor of Informatics - 6th semester

	Mon	Tue	Wed	Thu	Fri
8:30am	Languages & Compilers in SI-003	Languages & Compilers in SI-003			
9:30am	Languages & Compilers in SI-003	Languages & Compilers in SI-003			
10:30am	Theory of Computation in SI-003	Theory of Computation in SI-003			Optimization Methods in SI-003
11:30am	Theory of Computation in SI-003	Theory of Computation in SI-003			Optimization Methods in SI-003
12:30pm					
1:30pm					
2:30pm			Bachelor Project i		
3:30pm					
4:30pm					

Optimization Methods

Room
SI-103

Wednesday at 9:30

Locked:

SI-003
SI-004
SI-006
SI-007
SI-008
SI-013
SI-015
SI-103

Figure 3.7: A non refined schedule

Chapter 4

Case study

During development, we tested the solver using as problem instance the data for the spring semester of the year 2018 of the Bachelor program of the Faculty of Informatics at USI[1]. The problem instance features 3 semesters, with a total of 14 courses. The number of lectures is 70, and they need to be scheduled on a school week which counts 40 periods (8 time slots per day). As available rooms we used all the rooms of the Informatics Building, so 8 rooms.

The resulting schedules are produced in about 7 seconds and are comparable to the real timetables that are being used in the current semester. While the found solution does not violate any hard constraint, some soft constraints are violated, such as semester room stability and semester compactness (see Section 3.1).

We also observe that the solver is biased towards scheduling lectures in the first days of the week.

In Table 4.1 and Table 4.2 we show one semester of the results produced by the solver and the respective real semester from the official timetable.

The main differences are in the rooms: The official timetable only uses one room (SI-008) while the scheduler used two (SI-003 and SI-008). The bias of the solver towards scheduling in the first days of the week is very visible in the afternoons.

In the tables we omit the last two periods (15:30 and 16:30) because they are empty in both tables.

[1]http://www.inf.usi.ch

Time	**Mon**	**Tue**	**Wed**	**Thu**	**Fri**
08:30	PF2 SI-003	PF2 SI-003	PF2 SI-003	LA SI-003	DS SI-006
09:30	PF2 SI-003	PF2 SI-003	PF2 SI-003	LA SI-003	DS SI-006
10:30	ADS SI-003	ADS SI-003	LA SI-003	DS SI-006	
11:30	ADS SI-003	ADS SI-003	LA SI-003	DS SI-006	
13:30	SA2 SI-006	SA2 SI-006			
14:30	SA2 SI-006	SA2 SI-006			

Table 4.1: Bachelor 2nd semester produced by the solver

Time	**Mon**	**Tue**	**Wed**	**Thu**	**Fri**
08:30	LA SI-008	ADS SI-008	LA SI-008	ADS SI-008	
09:30	LA SI-008	ADS SI-008	LA SI-008	ADS SI-008	
10:30	PF2 SI-008	DS SI-008	PF2 SI-008	DS SI-008	PF2 SI-008
11:30	PF2 SI-008	DS SI-008	PF2 SI-008	DS SI-008	PF2 SI-008
13:30		SA2 SI-008		SA2 SI-008	
14:30		SA2 SI-008		SA2 SI-008	

Table 4.2: Real Bachelor 2nd semester

LA	Linear Algebra
ADS	Algorithms & Data Structures
PF2	Programming Fundamentals 2
DS	Discrete Structures
SA2	Software Atelier 2: Human-Computer Interaction

Chapter 5

Conclusion

The implemented solver produces good results which are comparable to those produced by the Dean's office, but in a much shorter time.
The application allows the user to create a schedule for a previously determined set of courses and rooms. The user can then visualize the solution, modify it and ask the solver to refine it.

5.1 Future work

The web application allows users to create a schedule for a given set of rooms and courses, with predefined unavailable periods. As a future work, the UI can be improved to enable the selection of specific courses, rooms, and other constraints.
Other possible extensions include approving a suitable schedule and publishing it to the official University calendar. This calendar may offer different ways to aggregate the events such as by room, instructor, user, or by day.
The calendar could also be augmented with the possibility to add non-recurring events such as seminars or conferences.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.