Università della Svizzera italiana

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 definition

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 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 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] 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 describes in more detail the planning problem (section 2.1) and explains on a high level how the chosen tool, OptaPlanner, can be used with this definition (section 2.2).

2.1 Planning problem

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 (usually between Monday and Friday) and a time slot during the day (e.g. in the range 0-7, with four slots 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

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

A solution is a list of lectures, each with an assigned room and a period, representing a weekly occurrence of a course in the schedule. Each solution has a score, which is calculated according to set weights related to the constraints. The score is separated in two parts (or levels): a hard score and a soft score; 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. The hard score relates to hard constraints, while the soft score relates to soft constraints.
Hard constraints usually model physical limitations; in our case there are five:

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

Soft constraints model limitations which do not violate physical reality but which satisfy specific preferences about how the solution should be:

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

Each constraint has an associated value which influences the score. In our case, all values are negative, so the highest score possible is `0hard/0soft`. The way constraints are associated with scores can vary: for example, if the room capacity constraint is violated, each student above the 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.

2.2 OptaPlanner

OptaPlanner is a constraint solver which can solve planning problems 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 an detailed guide on how to implement a solver. The solver can be configured to choose different construction heuristics (the first phase of solving) and different algorithms for the local search phase.
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 initialised first during the construction heuristic phase) or to sort planning variables by weight.
For more information on OptaPlanner, see appendix A.

Chapter 3

Approach

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

3.1 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 multi-purpose programming language) using Akka HTTP[1]. 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. This GraphQL endpoint is also implemented in Scala using Sangria GraphQL[2].
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[3].

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

3.2 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` OptaPlanner examples[4]
The main solution class, called `Schedule`, contains all the lists of all problem facts (`Periods`, `Days`, `TimeSlots`, `Semesters`, `Instructors`, `Rooms`, `Courses`, `UnavailablePeriods`), the list of planning entities (`Lecture`) and the score (`HardSoftScore`).
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 of are `period` and `room` (of type `Period` and `Room`, respectively).
However, if a 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`.
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. Only the relevant fields are shown:
To know more about the inner workings of OptaPlanner, see appendix A.

3.3 Constraints

The constraints are of two kinds, hard constraints and soft constraints, as described in section 2.1.
As a rule, any solution that violates any hard constraint is unusable, 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

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

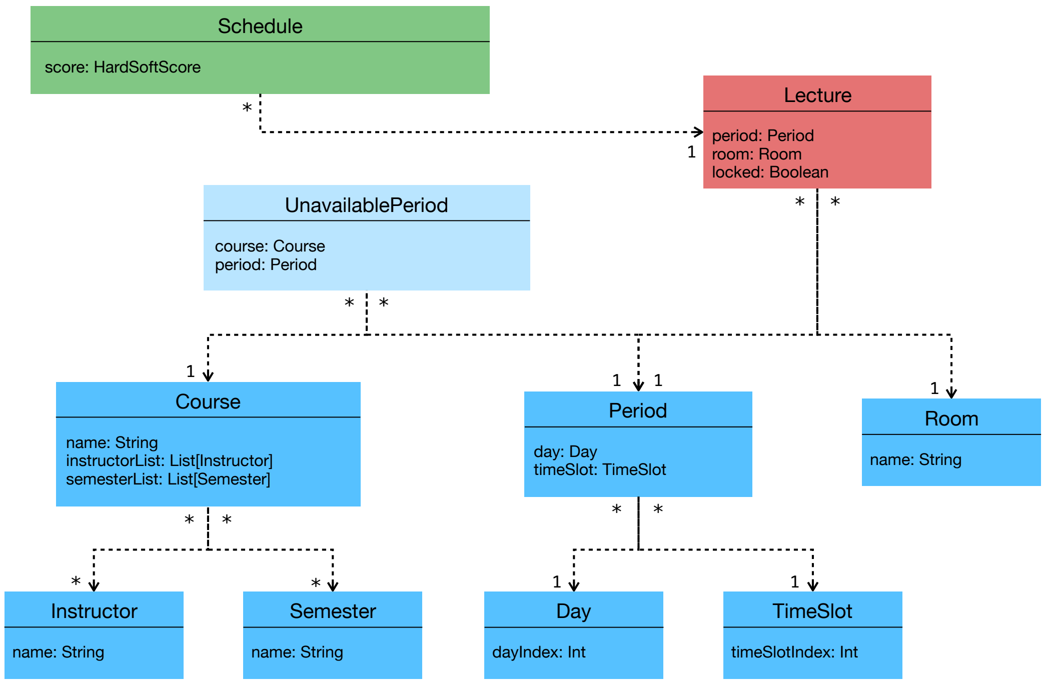

Figure 3.1: Class diagram of the domain model

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, 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 is available in figure 3.2.
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 2.1).

```
// 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

3.4 Solver architecture

3.5 Server and GraphQL API

3.6 Web application UI

Chapter 4

Conclusion

4.1 Future work

Appendix A

OptaPlanner

Appendix B

GraphQL

This is the OptaPlanner appendix...

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.