Fabian Germann & Raphael Jenni

Automation of the OST-RJ Examination Scheduling

Semester Project

OST – Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Supervision

Prof. Dr. Farhad Mehta

December 2020

Abstract

Exam scheduling is a known NP-complete problem. Finding the best solution for such a problem is near impossible for a human and takes forever for a computer. Computer-aided exam scheduling, or generally speaking problem solving, takes advantage of trying many possible solutions in an automated way and combining it with algorithms that help optimize the solving process. Testing the quality of a solution is carried out with several constraints and their assigned penalties and weights.

In this project, a constraint solver called OptaPlanner was used to model the problem domain and create constraints for it, all written in Java. The constraints correspond to the explicit and implicit constraints the (human) exam planner applies when scheduling the exams. The data is imported from files, processed, solved, and exported as a file and visualized in a web frontend.

The results are not production-ready but build a reasonable basis for future work. All hard constraints can be fulfilled, and some of the soft constraints are optimized. Scheduling the exams becomes much more comfortable, and the manual work can be reduced from several hours to an absolute minimum.

Keywords: Exam Scheduling, Problem Solving, Constraint Programming, OptaPlanner, NP-Completeness.

Executive summary

Exam scheduling is known to be a challenging problem to solve. Manually scheduling the exams was executed in an enormous Excel file and took several weeks. With the help of computer-aided exam scheduling, we provided the first version of an automated solution.

In the exam scheduler's current version, all mandatory rules for feasible schedules are fulfilled. One optimization rule is in place. The rules are defined with a constraint solver called OptaPlanner, an open-source library maintained by RedHat. The exams and students are imported from files, processed, scheduled, and exported as files and visualized in a web application.

The current results are not production-ready but provide a reasonable basis for future work. Scheduling the exams becomes much more comfortable. The manual work can be reduced from several hours to an absolute minimum. Comparing a generated examination schedule with the manually created one has shown the potential to create better examination timetables.

To make it production-ready, all the remaining rules need to be implemented. The frontend requires additional features, a design overhaul, and some quality of life features. For performance improvement, the algorithms for scheduling the exams can be optimized. With the continuation of this project, a fully automated scheduling process is possible. It promises significant improvements in the quality of the schedules and the time and iterations it takes to create such a schedule.

Contents

Chapter 1

Introduction

About 250 exams involving up to 1,500 students take place on the OST campus Rapperswil-Jona each semester. Creating an examination schedule to this extent is a challenge. Many spatial and temporal constraints have to be taken into account. Until now, the timetables have been created manually by the examination planning team.

1.1 Objective

Creating an examination schedule by hand is a complicated and tedious task that takes several weeks to finish. Moreover, this repetitive task must be completed anew every semester. Hence, the overall goal of this project is to come up with a software solution that supports the examination planning team in its job.

The most challenging and time-consuming part of the whole process is the creation of an optimal examination timetable that fulfills certain constraints. Therefore, the focus of this project is on the automatic generation of optimal timetables. The objective is to develop a software solution that is able to find the best possible examination schedule, given a list of constraints.

1.2 Structure of the Report

The report is organized as follows:

Chapter 2 gives an overview of the domain and describes the examination timetable problem in detail – including all constraints that have to be taken into account for an optimal timetable. It also explains why this problem is not easy to solve.

Chapter 3 reflects the literature research that was conducted at the beginning of this project. In addition, existing solutions and possible strategies for the exam scheduling software are described and evaluated.

Chapter 4 describes our solution to the problem. This includes the software architecture as well as our architectural decisions and solution strategy. Moreover, the main aspects of our implementation are explained in detail.

Chapter 5 presents and discusses the result of our semester project. Some generated exam timetables are shown and analyzed in detail. A comparison with a former manually created timetable is provided as well.

Chapter 6 summarizes the outcome of the semester project. In addition, an outlook regarding the developed software is given.

Chapter 2

Problem Analysis

2.1 Domain

The problem domain (Figure 2.1) is straightforward. Students register for exams they want to take. Exams are supervised by one or more supervisors. The goal is to create an exam timetable that consists of multiple entries. A timetable entry is the scheduling of a single exam. It assigns a date and one or multiple rooms for each exam.

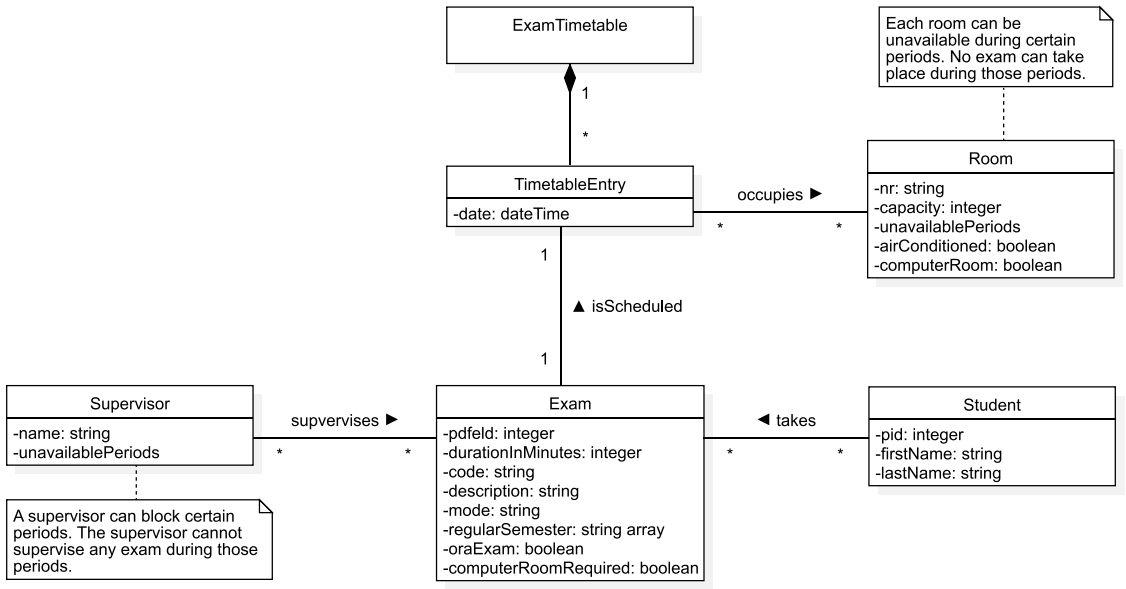

Figure 2.1: Domain Model

Many entities have natural identifiers. Each student is uniquely identified by its “PID”. Similarly, each exam has a unique id which is arbitrarily called “pdfeld”. Rooms have a unique room number. Only the supervisors do not have an id and are just identified by their names.

A room might not be available for certain periods of time as it is used for other purposes. Similarly, a supervisor can block certain time ranges. This means he or she is not available for supervising an exam during this time.

Each exam is assigned to one or more regular semesters. The exam usually takes place in these semesters. In other words, students normally write the exam in these semesters. Each degree program has its own regular semester. For instance, the full-time degree program Computer Science has in total 6 regular semesters.

2.2 Problem Description

Currently, all examination plans for the OST are created by hand. Aside from that, the plans are enormous Excel files that are color-coded and contain a lot of information between the lines. A VBA script, created around 2000, is in place to add some basic collision detection. This process is tedious and error-prone. The creation of a single exam timetable takes several weeks. It is especially complicated as there are a lot of different hard and soft constraints that have to be met according to certain priorities.

Hard constraints are constraints that cannot be broken. For example, a student cannot write two exams at the same time, as this is physically not possible. Soft constraints are constraints that should be fulfilled for a perfect solution but can be broken if needed [1].

The hard constraints:

- Students cannot have two exams at the same time.
- Students can only have two exams that take up to two hours or one exam that takes more than two hours on the same day.
- Students need a break of at least two hours between exams.
- Supervisors cannot have two exams at the same time.
- There can only be one exam in a room at a time.
- During the summer, exams after noon only take place in rooms with air conditioning.
- The room needs to have the capacity for at least two students more than registered.
- There has to be a break of at least 20 minutes between two exams.

Additional requirements:

- Some exams need to be fixed to a certain time slot.
- Supervisors should have the possibility to block certain time slots.

- Rooms can be blocked during certain time slots.
- It should be possible for exams to be split into multiple rooms.

The soft constraints and priorities (1 = high, 5 = low) are:

1. The exams of a regular semester should be distributed as evenly as possible over the examination session.
2. As few students as possible should have more than one exam on the same day.
3. Exams of an individual student should be distributed as evenly as possible over the examination session.
4. The exams are distributed as evenly as possible over the examination time regarding the correction time each lecturer has.
5. Exams should take place in a single room if possible.

All those requirements have to be taken into account for about 250 exams with around 1,500 students.

2.3 The Exam Timetable Problem is NP-Complete

First of all, there might not be a perfect solution to the above-mentioned problem description. There might not be a solution that meets all the requirements and their priorities. But there is certainly a solution that is acceptable and that is better than any other solution there is to find.

Why is solving the exam timetable problem not an easy task? The answer to this question can be found in the theory of problem-solving and computer science, which is known as computational complexity theory. Problems like our exam planner or in general the creation of a timetable belong to the problem category of NP-complete problems [2].

An NP-complete problem is a problem that cannot be solved efficiently. This means there is no existing algorithm that finds an optimal solution to an NP-complete problem in a reasonable time. A brute force search, i.e. go through all combinations and take the best one, takes too long [3].

NP means “nondeterministic polynomial time” and tells us that solving the problem becomes exponentially harder the bigger the set of data is [4]. For the exam timetable problem, this means the more students, exams, rooms, etc. are given, the exponentially bigger the problem becomes. The difference of polynomial (e.g. n^5) and exponential (e.g. 2^n) complexity is visualized in Table 2.1. It shows the runtime complexity of two theoretical processes.

For quite a few steps the polynomial process is slower than the exponential. But at a certain point ($n = 64$ in the example) the time for the exponential process makes a huge jump. At the point, $n = 128$, the time it takes for the exponential process to finish is 78 billion times the age of the universe [4].

Table 2.1: Polynomial and Exponential Runtime Complexity [4]

n	n^5	2^n
1	1 ns	2 ns
2	32 ns	4 ns
4	1.0 μs	16 ns
8	32.7 μs	256 ns
16	1.0 ms	65.5 μs
32	33.5 ms	4.3 ms
64	1.0 s	584 $Years$
128	34.4 s	10^{21} $Years$

Seeing these results, solving the problem by brute force is not an option. The only possibility available is to try to find algorithms that run efficiently and get as close to an optimal solution as possible. Given the algorithms available today, finding “the best solution” is impossible. But the goal is to find a solution that meets all the requirements in a reasonable matter of time.

Chapter 3

Research

Key literature for our semester project is the thesis “Constraint-based Timetabling” by Tomáš Müller from the Charles University in Prague. It shows that timetabling, such as generating an optimal examination timetable, falls in the category of constraint satisfaction problems. Such problems can be solved with the help of constraint programming [5].

In constraint programming, the problem is described in a declarative manner using variables and relationships between them. Then a constraint solver takes over the solution process. In other words, the user states the problem, the computer solves it [6]. In the case of the exam timetable problem, the task is to allocate exams in time and space (rooms) respecting various hard constraints and to satisfy as well as possible a set of desirable objectives (soft constraints). A typical hard constraint is that exams which take place in the same room cannot overlap in time. An objective could be that the exams for an individual student are as evenly distributed over the exam session as possible [5].

3.1 Solution Strategies

Timetabling is a well-known problem across many domains and a lot of research has been carried out. Hence in the first step, we looked for existing solutions. We found quite a few which specifically tackle the exam timetable problem. The range varies from domain specific constraint solver libraries to ready-to-use products for schools and universities.

Looking at the source code of a constraint solver library, such as the CPSolver originated in the dissertation of Tomáš Müller, reveals the true complexity behind it. Developing and implementing a complete solution by ourselves would not make any sense nor be feasible in one semester. But there is not a solution readily available that fits our needs perfectly. So we have two possible strategies for implementing the exam planner:

- Using a constraint solver as a dependency
- Building our solution on top of an existing product

3.1.1 Requirements for Third-Party Software

Independently of the previously stated strategies, we needed some third-party software. We considered the following criteria for the evaluation of third-party software.

Open source: The software should be easy to adapt/extend according to our specific needs. Therefore, open source is an important criterion for us. Furthermore, free software is required as this project has no finance budget.

Customization/API: If the software is not open source it should at least be customizable or provide a comprehensive API such that all our constraints of our problem description can be covered.

Active maintenance: The software should be somehow established and actively maintained by a company or a community. Otherwise, there is the danger of a deprecated dependency. Furthermore, the whole topic is still in research. If algorithms in the dependency improve in the future, our software can benefit as well.

Programming language: Preferably the software is written or can be used in Java or C#. In these programming languages, we have the most experience and can build upon a strong type system and a wide range of libraries.

3.1.2 Constraint Solver vs. Existing Products

Only two existing products of our research fulfill the requirements regarding third-party software. They are listed below with a brief description from their website:

UniTime “UniTime is a comprehensive educational scheduling system that supports developing course and exam timetables, managing changes to these timetables, sharing rooms with other events, and scheduling students to individual classes. ... It can be used alone to create and maintain a school’s schedule of classes and/or exams, or interfaced with an existing student information system. ” [7]

Free Timetabling Software "FET is open source free software for automatically scheduling the timetable of a school, high-school or university. It uses a fast and efficient timetabling algorithm." [8]

At first glance, building the solution on top of an existing product might be promising. The biggest advantage is the fact that the domain is already implemented. However, a closer look reveals crucial disadvantages. Both products are a fairly large piece of software. They provide features, such as LDAP lookup for data import, that are far beyond the scope of our semester project. Also, their domain model is more complex than ours. Last but not least, the algorithms for the optimization problem are given by the product. In conclusion, building an application based on one of these products, requires a lot of effort for nothing except a tight coupling to the product. Therefore we decided that the strategy with a constraint solver library will be pursued.

3.2 Evaluation of Constraint Solver

For the constraint solver, we evaluate the following libraries. All of them fulfill the requirements we defined for third-party software.

- CPSolver (UniTime.org)
- Choco Solver
- OptaPlanner

3.2.1 CPSolver (UniTime.org)

UnitTime.org provides its own constraint solver (CPSolver) as a library. It is written in Java. The source code is hosted on GitHub.

Pros

- The constraint solver is optimized for our problem domain.
- There is an "Examination Timetabling Extension" available which provides a domain model for the constraint solver.
- A Ph.D. thesis explaining the problem and underlying algorithms is available.

Cons

- The library does not have any tests.
- The model provided is tightly coupled to an XML data import.

- There is a lack of documentation, only JavaDoc is available. This makes it difficult to use custom models.
- The library is maintained by a small community (less than 5 contributors on GitHub).

3.2.2 Choco Solver

Choco is a free open source Java library for constraint programming. Its source code is hosted on GitHub.

Pros

- An elaborated documentation with good examples is available.
- The project has automatic tests.
- The library is maintained by a medium community (about 27 contributors on GitHub).

Cons

- The problem modeling is done on a mathematical level.

3.2.3 OptaPlanner

“OptaPlanner is an AI constraint solver. It optimizes planning and scheduling problems, such as ... School Timetabling ... and many more.” [9]. The project is sponsored by Red Hat. OptaPlanner is written in Java and hosted on GitHub.

Pros

- An elaborated documentation with many good examples is available.
- The project has automatic tests and unit test support.
- The library is maintained by a large community (about 94 contributors on GitHub) and sponsored by Red Hat.
- Constraints apply on plain domain objects. There is no need to model constraints as mathematical equations.
- Combines sophisticated artificial intelligence optimization algorithms (such as Tabu Search, Simulated Annealing, Late Acceptance, and other meta-heuristics).

Cons

- Documentation could be a bit more structured, especially regarding the different Java annotations and configurations.
- Constraint definitions must follow a specific pattern.

Chapter 4

Solution

This chapter describes our approach to the given task. It includes the reasoning behind our choice of the constraint solver, the explanation of the software architecture, and the description of the quality measures we defined.

4.1 Constraint Solver Choice

After the evaluation of some possible constraint solvers (section 3.2) we decided to use the OptaPlanner. The main reasons are its good and clean documentation, the possibility to write extensive tests, and the fact that it is written in Java.

The OptaPlanner solves/optimizes a problem based on defined constraints. It does this by including several different algorithms that perform some kind of “search” and some heuristic-based work. These algorithms are combined to strive for a near-perfect solution. Why only “near-perfect”, you may ask. To answer this question, the documentation of the OptaPlanner provides a very good graphic (Figure 4.1). It shows that an algorithm which returns a perfect solution either needs a very long time or a huge amount of RAM/storage. You can also see how the different types of algorithms perform on different sizes of input data/variables. Their website also has an extensive comparison table that compares many different algorithms based on some measures and preconditions [10].

The OptaPlanner allows us to explicitly select and tweak each of those algorithms to get the optimal result for our problem domain in the shortest amount of time. It also includes a benchmarking tool to compare those different algorithms with each other. However, for this semester project, we decided to stick to the default, “no explicit configuration needed” method, as the whole problem domain and constraints must be implemented.

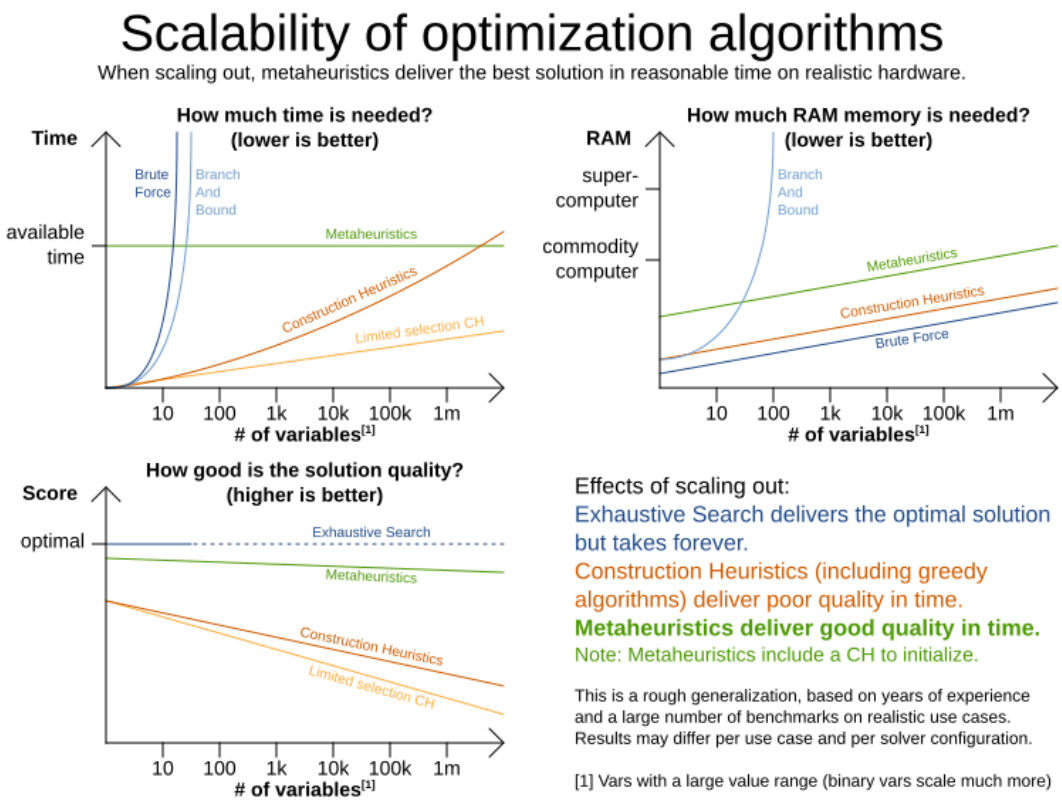

Figure 4.1: Scalability of Optimization Algorithms [10]

4.2 Problem Modeling and Implementation with the OptaPlanner

The OptaPlanner is domain independent, which means there are no ready-to-use or pre-configured domain models available. The problem domain has to be modeled by ourselves and the OptaPlanner needs to be instructed about how to understand it. This is done with Java annotations. The most important annotations are: `@PlanningSolution`, `@PlanningEntity` and `@PlanningVariable`.

4.2.1 Model

The **`@PlanningSolution`** is the heart of the model, the place where everything comes together. In the case of the exam scheduler, this is the exam timetable (Listing 4.1). It contains a list of all exams (the planning entities) and provides the available rooms and time slots, which can be assigned to each exam's planning variables. The planning variables' values are also called problem facts as they do not change during the solving process.

```java
@PlanningSolution
public class ExamTimetable {
   @PlanningScore
   private HardSoftScore score;

   @PlanningEntityCollectionProperty
   private List<Exam> exams;

   @ValueRangeProvider(id = "roomRange")
   @ProblemFactCollectionProperty
   private List<Room> rooms;

   @ValueRangeProvider(id = "timeGrainRange")
   @ProblemFactCollectionProperty
   private List<TimeGrain> timeGrains;

   public ExamTimetable(/*...*/) {/*...*/}
}
```

Listing 4.1: ExamTimetable Class of the OptaPlanner Model

The **`@PlaningEntity`** represents the part of the model that needs to be planned. In our case the exam is the planning entity (Listing 4.2). It has a room and a time slot planning variable (annotated with `@PlanningVariable`). These planning variables indicate when and where the exam takes place and have to be assigned by the constraint solver.

```java
@PlanningEntity
public class Exam {

   @PlanningId
   private long id;

   @PlanningVariable(valueRangeProviderRefs = "roomRange")
   private Room room;

   @PlanningVariable(valueRangeProviderRefs = "timeGrainRange")
   private TimeGrain startingTimeGrain;

   /* Further fields and functions ... */
}
```

Listing 4.2: Exam Class of the OptaPlanner Model

The OptaPlanner supports multithreaded solving. This allows searching for a solution on multiple threads simultaneously. For this functionality, OptaPlanner needs to map problem facts and planning entities to an ID. This ID is used to rebase a move from one thread's solution state to another's. The ID is defined by the `@PlanningId` annotation [11].

In addition, OptaPlanner clones the problem facts and planning entities for each new best solution. This brings some challenges with it. Back referencing a planning entity from a planning variable cannot be made at initialization time as the assignment is changed over and over again. To solve this problem, so-called “shadow variables” can be used. They get automatically updated by the OptaPlanner [12].

The same back referencing problem occurs when working with classes/objects that have no meaning to the OptaPlanner. In our case, a student writes exams. Therefore, an exam has students assigned to it. Those students per se do not matter to the OptaPlanner as it cannot move them around. The students are assigned to exams before handing over the problem statement to the OptaPlanner and they must stay there. They are “locked” to the exam. As a result, the OptaPlanner ignores them, which means that they are excluded from the OptaPlanner context.

To tackle this issue, we first tried to add a reference from the exam to the students at the initialization. However, this did not work. As previously mentioned, OptaPlanner creates clones of the planning entities for each new solution. After some investigation, it turned out to be a common problem that can be solved by instructing the OptaPlanner to clone the referenced objects together with the planning entity [13].

4.2.2 Constraints

After creating the domain model, the OptaPlanner knows what values it can assign to the planning variables to achieve the desired output, i.e., an exam time table. However, up to this point, the OptaPlanner has no indication of what a valid exam time table is and how to measure or to decide that the generated solution is any good. At this point the constraints, and with it the real work, come into play.

The OptaPlanner and many other solvers differ between hard and soft constraints. As described in section 2.2, hard constraints should not be broken by any means. On the other hand, soft constraints are constraints that should be fulfilled to get a better solution. Depending on the hard constraints and the input data, this might not even be possible. Nevertheless, the solver tries to optimize the soft constraints as far as possible.

Constraints can be defined with OptaPlanner’s ConstraintStream API, which is inspired by the Java Stream API and SQL [14]. Listing 4.3 shows the definition of a hard constraint. Soft constraints are defined in the same way. An example is given in Listing 4.4. The code comments describe each part of the constraints in detail.

The starting point of a constraint is typically the creation of pairs of exams, as the comparison of their properties is the primary subject of interest. This step can optionally include a join via attributes. The constraint `roomHasTwoExamsAtTheSameTime` (Listing 4.3) uses

this to get the exam pairs whose exams have the same room assigned, i.e., take place in the same room. Then one or multiple filters are applied. The filters contain the main logic of the constraints. Their job is to only let exam pairs through that match the constraint. After that, a penalty for a negative score or a reward for a positive score is given.

A penalty/reward can be hard or soft and of different weights, which can be adjusted by a custom weight function. The return value of this function multiplied by the base score results in the total score for a constraint match. If no weight function is provided, the default weight of one is used. The constraint `studentsHaveMoreThanOneExamOnSameDay` (Listing 4.4) has a weight function that counts the number of students that have to write both exams of the exam pair. Hence, the more students are affected, the more significant the impact of the score value becomes.

```
public class RoomConstraints extends AbstractConstraints {
  // ...
  public Constraint roomHasTwoExamsAtTheSameTime() {
    // A room can accommodate at most one exam at the same time.
    return constraintFactory
        // Two exams that have the same room assigned get grouped to a pair.
        .fromUniquePair(Exam.class, Joiners.equal(Exam::getRoom))
        // This filter only lets through exam pairs
        // whose exams are overlapping in time.
        .filter(Exam::areOverlapping)
        // Each exam pair, whose exams are in the same room at the same time,
        // get a penalty of 100 hard.
        .penalize(
            "Room has two exams at the same time",
            HardSoftScore.ofHard(100)
        );
  }
  // ...
}
```

Listing 4.3: A Hard Constraint of the RoomConstraints Class

```
public class StudentConstraints extends AbstractConstraints {
  // ...
  Constraint studentsHaveMoreThanOneExamOnSameDay() {
    // As few students as possible should have
    // more than one exam on the same day.
    return constraintFactory
        // Two exams get grouped to a pair.
        .fromUniquePair(Exam.class)
        // This filter only lets through exam pairs
        // whose exams are on the same day.
        .filter(Exam::areOnSameDay)
        // This filter only lets through exam pairs
```

```
13        // whose exams have common students.
14        .filter(Exam::haveCommonStudents)
15        // Each exam pair, whose exams are on the same day and have
16        // common students, get penalized with a soft score of 1 per student
17        .penalize("As few students as possible should have more than one exam ↵
           on the same day.",
18            HardSoftScore.ofSoft(1),
19            // This helper function returns the number of students,
20            // that have two exams, i.e. write both exams of the exam pair.
21            StudentConstraintHelper::countStudentsWithMoreThanOneExamOnSameDay
22        );
23    }
24 }
```

Listing 4.4: A Soft Constraint of the StudentConstraints Class

4.2.3 Score Function

The constraint solver's goal is to maximize the score function. A higher value indicates a better exam timetable. The best way to illustrate the score function is to look at one of our many runs, attempting to create a near-perfect exam schedule. As input, we have the real data from the fall semester of 2019. It contains 196 exams with 1,322 students that should be put into 20 rooms over three weeks. When starting the solving process, the planner starts with an initial score of `-392init/0hard/0soft`, where -392 is the negative sum of all uninitialized planning variables ($-\sum \#\ of\ uninitialized\ planning\ variables$). As there are 196 exams with two planning variables (room and time slot), the sum is $2 * 196 = 392$. The values `0hard` and `0soft` are irrelevant at the moment as for calculating the scores, only planning entities with assigned planning variables are evaluated.

In the first step, the solver tries to assign a room and time slot for each exam based on some heuristics. This step is called the construction phase and is partially random. With the help of a seed, it is set up to be reproducible, though. This phase takes in our case, on a machine with 16 virtual cores (an Intel i9-9900k) and a clock speed of up to 5 GHz, about 5 minutes. When it is finished, we get the first real score: `-4hard/-169soft`.

To calculate the shown score, the OptaPlanner sums up the score of all matches for each constraint. The score of a single match is the product of the weight and the base score, according to the constraint definition explained in subsection 4.2.2. More formally, the score function is expressed as follows:

Let C be the constraints, s_c the constraint base score, M_c the matches of the constraint $c \in C$, and w_m the weight of the match $m \in M_c$:

$$Score = \sum_{c \in C} \sum_{m \in M_c} w_m * s_c$$

Figure 4.2: OptaPlanner Score Function

A solution is feasible if the hard score is 0, e.g., `0hard/-82soft`. As for now, all defined constraints give a penalty (negative score). Therefore, the maximum of the score function is 0. A score of `0hard/0soft` indicates a perfect solution, i.e., all hard and soft constraints are 100% fulfilled. After this being clarified, we will continue with the solving process.

In the next step, the solver takes that "constructed" first result and continues with the second phase, the local search phase. By default, a hill-climbing algorithm is used. "This one simply tries all selected moves and then takes the base move, which is the move which leads to the solution with the highest score." [15]. This is by far not an optimal solution as it can get stuck in a local optimum (a place where every change makes the result worse, but better solutions would exist out there). But it gives us an easy and simple first approach to solve our problem. Later on in the process, different algorithms that have the potential for much better solutions can be chosen. However, this is not part of our semester project.

After running this phase for around two hours, we ended up at a score of `0hard/-35soft`. This is not a perfect score, but the best solution found in the given time. It is even possible that there is no perfect solution for this input data as we are after all working on an NP-complete problem.

4.3 Architecture and Design

Our system architecture shown in Figure 4.3 follows a classic approach. The core is an API application running SpringBoot, which provides an easy to set up and robust environment running Java. The data is persisted in a database. As the data schema was known up-front, we went for a RDBMS. The database of our choice is PostgreSQL because it is open-source, widely used in a lot of enterprise solutions, and a system we are familiar with. Since most data queries are simple CRUD operations, we use JPA to map the entities in the Java code to the database.

The user can interact with the system via a browser-based GUI. For this, we use Angular. The Angular framework provides a robust structure and a huge set of ready-to-use tools.

The single-page application and the Swagger API documentation is served via the Spring MVC included in the SpringBoot container. The client-server architecture allows to deploy the backend on a high performance server, while the frontend can be accessed by any “normal” machine.

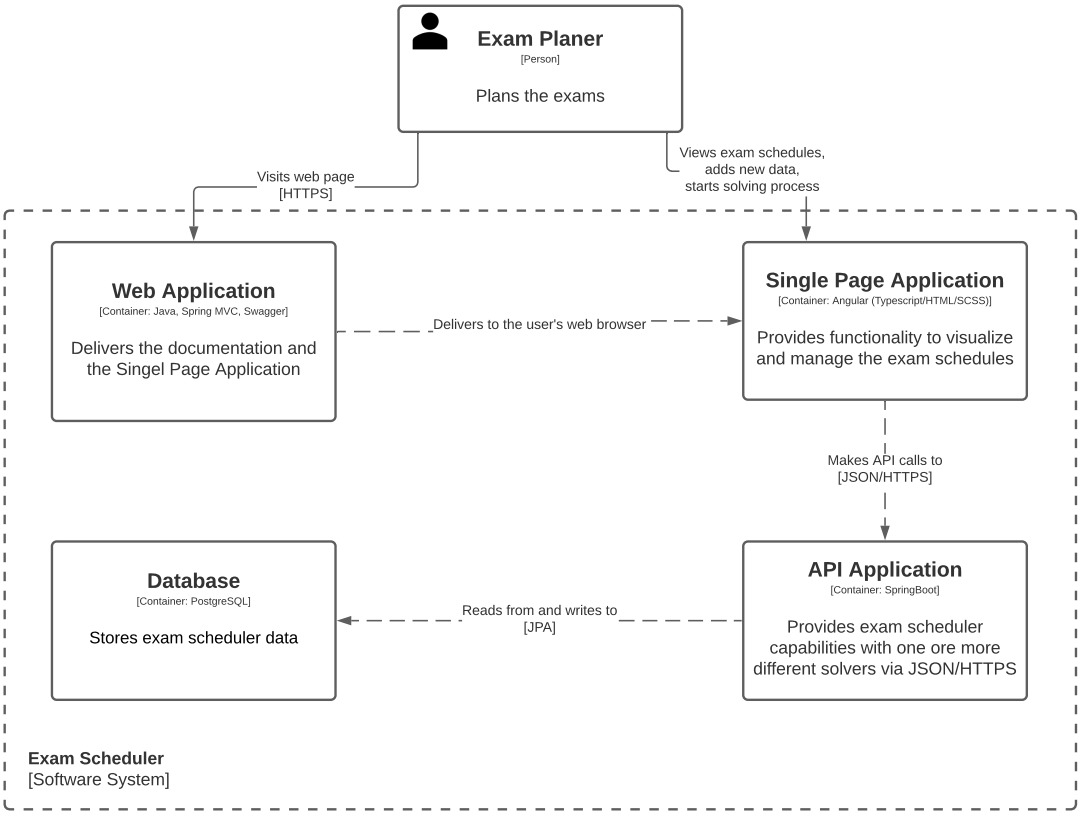

Figure 4.3: C4 – Container Diagram

4.3.1 Core System and API Application Architecture

As the constraint solver choice is not definite and might be subject to change in the future, we must design our core system with that in mind. As seen in the architecture diagram (Figure 4.4), our system is split into two parts. On the left, we have our core functionality, including database access, REST providers, models, and control unit. On the right, we have our solver implementation(s). The system is explicitly designed to provide the possibility to replace or add a solver without having to change something in the core logic. Everything is based on a solver connection interface that all solvers have to implement. This interface requires the functionality to load the core model as input and to return a solved core model as output. How an individual solver does the solving is entirely open, what is relevant is the resulting solution and its score.

In the case of the OptaPlanner (see section 4.2 for details), the setup of the solver is done with code annotations. The constraints are defined in a Java Stream API like language, and an internal scoring system calculates the score.

To compare multiple solvers to one and another, it would be necessary to implement an independent scoring system. As we currently use only one solver and implementing an additional solver is time-wise out of this project's scope, we leave it at that time.

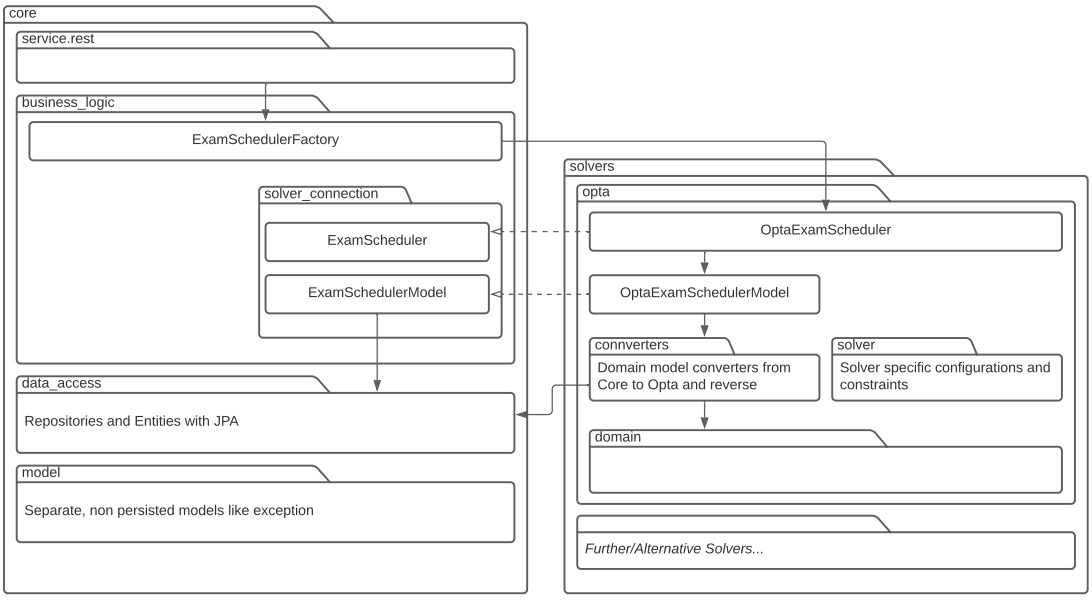

Figure 4.4: Architecture Diagram

4.4 Visualization

To improve the quality of the results our exam scheduler produces, we need a way to verify them. Unit tests are great to check the correctness of the constraint. However, they do not provide an intuitive understanding of the exam schedules. Therefore, we implemented a visualization tool that displays the calculated exam schedule as a big table (Figure 4.5). The columns represent the days and the rows the time slots. Exams are placed into those slots which provide the exam name and the room where it takes place.

This visualization makes it very easy to see conflicts between exams and provides details on how the scheduler places the individual exams throughout the entire given time range. A detail page is not available right now, but it logs all details to the browser's console by clicking on an exam. With the help of filters, we can perform some additional solution evaluation regarding exams for single students, exams in a specific room, and the distribution of exams for a regular semester (Figure 4.6).

Even though this tool is still very much work-in-progress, especially regarding colors and design, it helped us a lot to understand how we can improve the constraints and what errors are hidden in the data.

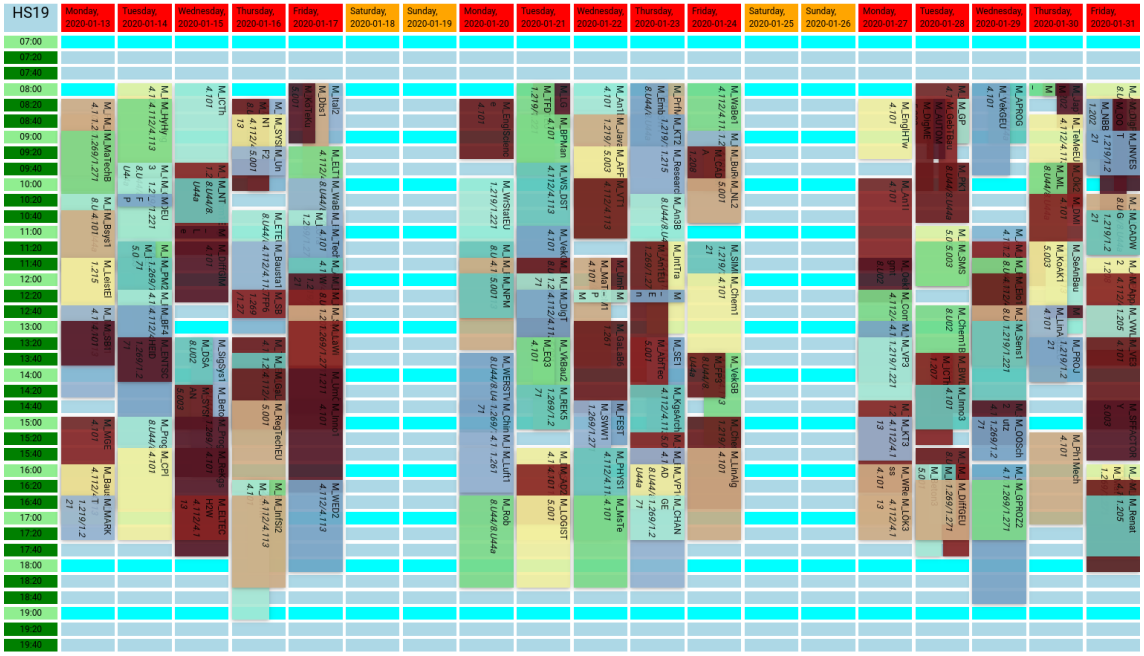

Figure 4.5: Visualization Tool – Overview

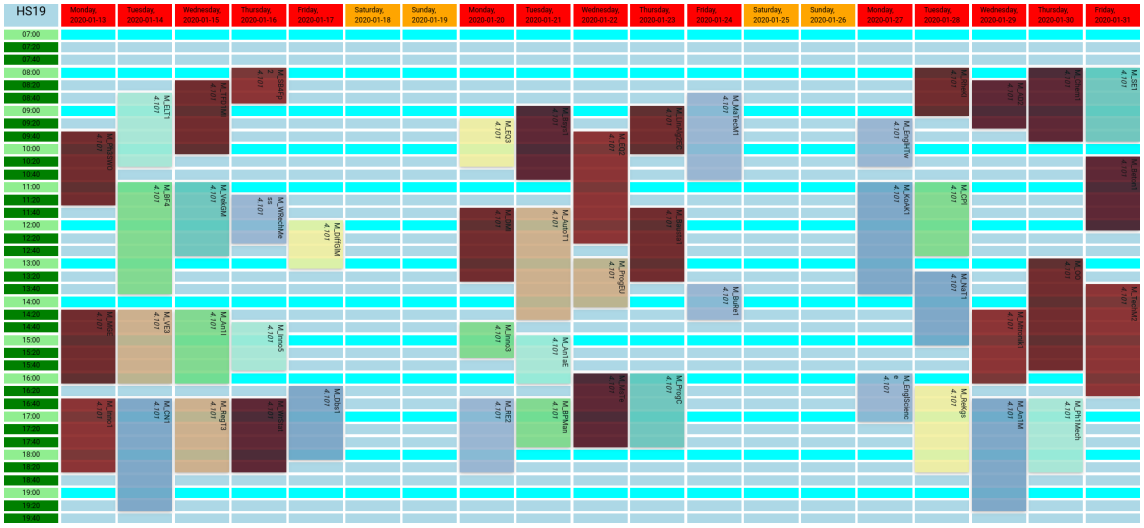

Figure 4.6: Visualization Tool – Room Filter

4.5 Data Import And Export

The input data is given as multiple Excel files. To process the data, we provide a data import API. It validates the input data and converts it into our solver independent data model. If the data is valid, it is persisted in the database, from where it can be used to create exam schedules. The principle “garbage in, garbage out” also applies in the case of the exam scheduler. Therefore, a rather aggressive validation is in place that detects invalid data early. This ensures that the system includes only valid data. The validation rules and input data format is implemented in a flexible and extensible way. Adding new data for additional constraints is very easy. Once an exam timetable is generated, it cannot only be visualized in the web browser, as shown in section 4.4, but also be exported as a spreadsheet or a JSON file. Exported exam timetables can therefore be used easily in an external application.

4.6 Quality Assurance

To assure the quality of our application logic, we have written around 350 unit tests for all classes containing some kind of logic. Integration tests are in place for the database queries, data model, and the Excel importer. We have not explicitly defined an overall test coverage goal as we aim for a component-wise test coverage as high as possible to catch possible errors as early as possible. The solver currently has a line test coverage of 92%, and the data access layer has a test coverage of 85%. The business logic and the (REST) services are subject to change and are therefore not that strictly tested.

As described in the previous section, for testing the quality of the solution, we currently rely on the score provided by the OptaPlanner. However, for ensuring the correctness of the defined constraints, tests are in place that explicitly tests each constraint on its own.

The front-end application that we use for visualizing the results and, later on, as a way for the user to interact with the system currently has no tests. This is due to the time restrictions and the fact that this system is not mission-critical. When continuing this project in the future, testing the logic in the front-end is one task that has to be done.

Chapter 5

Results

In our semester project, we successfully developed the first version of the automated examination scheduler for the OST. We implemented a constraint solver independent architecture and many features, such as a data import API, the visualization of exam timetables, and the possibility to export them. On top of that, we implemented the OptaPlanner as constraint solver, which actually solves the timetable problem. For this, we created an OptaPlanner specific model for the exam timetable problem and a converter that translates our solver independent data model to this specific model and back. Last but not least, we implemented the constraints that ensure valid and optimized exam timetables. Currently, the following constraints are implemented.

Hard constraints:

- Students cannot have two exams at the same time.
- Students can only have two exams that take up to two hours or one exam that takes more than two hours on the same day.
- Students need a break of at least two hours between exams.
- Supervisors cannot have two exams at the same time.
- There can only be one exam in a room at a time.
- The room needs to have the capacity for at least two students more than registered.
- There has to be a break of at least 20 minutes between two exams.

Soft constraints:

- As few students as possible should have more than one exam on the same day.
- Exams should take place in a single room if possible.

To make sure the exam scheduler works correctly, a total of about 350 automated tests are in place. Around 175 tests of them test the OptaPlanner problem model and its constraints. Moreover, we validated the exam scheduler with the real datasets of the fall semester 2018 and 2019.

In the rest of this chapter, we present the generated exam schedules and analyze them in detail. We also compare an existing examination schedule, created by a human, with a solution made by our exam scheduler. Last but not least, we try to disprove a thesis given by our customer: “An exam schedule in which all students have at most one exam per day is impossible.”

While defining the problem model and constraints, we encountered quite a few challenges. We had to look for solutions, and sometimes trade-offs were inevitable. Before jumping directly into the results, it is recommended to read the next section first. It gives additional information about the model’s internals, which are crucial for correctly understanding the following results.

5.1 Simplifications of the Current Model

It is important to state that the current problem model for the OptaPlaner underlies some simplifications due to findings during the requirement analysis and the limited time of the semester project. These simplifications and the reasoning for them are listed below.

Exams can take place only in one room: The largest room is room “4.101” (Aula). It has a capacity of 140 students. Typically, there are around five exams that have more than 140 registered students. These exams urge the need to schedule exams in multiple rooms. However, adding this functionality would make the model more complicated and increase the search space a lot.

Since there are only a few exams that need this feature, we decided not to implement it. This leads to the problem that some exams cannot be scheduled at all. To solve this issue, we increased the capacity of room “4.101” to 200. So exams with more than 140 registered students are scheduled in room “4.101”. Since this room’s capacity is, in reality, smaller, at least a second room needs to be searched manually. Finding another room is not a problem as there are usually enough rooms available. In the worst case, the exam could take place in three rooms. This trick is a bit of a trade-off we had to make. But in our opinion, an acceptable one for the first version of the exam scheduler. Note that this design decision also ensures the constraint “Exams should take place in a single room if possible.” implicitly.

An exam has only one supervisor: During the requirement analysis, we noticed that some exams have many supervisors. We asked ourselves whether all of them need to be considered for the constraint “Supervisors cannot have two exams at the same time.”. While clarifying this question with our customer, we found out that not all listed persons supervise the exam. This list is instead a pool of people that might supervise the exam. The final supervisors will be determined after the creation of the examination schedule. Therefore, considering all listed people makes no sense. Ideally, only the persons are provided that supervise the exam. However, this is not possible at the moment.

We recommend checking whether the process can be changed such that the supervisors are known upfront. Alternatively, only one person who is responsible for the exam should be named. As a workaround, we consider only the first person on the given list.

Exams cannot be fixed to a particular time slot: According to the requirements, some exams need to be fixed to a particular time slot. At first glance, this feature does not look that complicated. However, thinking of a concrete implementation proves the opposite. First of all, this must be possible in the GUI. Fixing the exams via another Excel file is a non-solution as in this process many things can go wrong and the user must be warned appropriately. For instance, the user might fix some exams that result in an infeasible exam schedule. Moreover, OptaPlanner is able to create feasible and optimized exam schedules without this feature. Because of these reasons, we gave this feature a low priority.

Supervisors cannot block certain time slots: Another requirement is that supervisors can block certain time slots. However, according to the task description, this feature is not planned in the first version. Nevertheless, we checked its feasibility. As for now, the required information is not available in the input data. Since the supervisors are given as a list of names for each exam, it might be tricky to provide this information. If this feature is still considered in the future, we recommend providing the supervisors in a separate list. This list can then include the information about the blocked time slots. In addition, each supervisor should have a unique ID that can be referenced in the list of exams.

Computer rooms are not considered: There are only a few exams that need a computer room. However, most of them have so many registered students that they must take place in multiple rooms. Compared to standard rooms, computer rooms have a relatively small capacity. As splitting up exams into multiple rooms is not yet possible, this simplification follows. Instead of removing these exams, we decided not to consider that an exam needs a computer room. This way distorts the result less as the exams are scheduled. However, finding a computer room for these exams manually might be challenging or even infeasible as there are not many computer rooms available.

Rooms are always available: In our first solution, rooms cannot be blocked during particular time slots. We made this simplification as all rooms were always available in the input data of the fall semester 2018 and 2019. Nevertheless, the data model is ready for this feature. Once the input data format is defined, a constraint regarding the unavailable periods can be implemented.

Room air conditioning is not considered: "During the summer, exams after noon only take place in rooms with air conditioning." is the only hard constraint we did not implement. It is only relevant for spring semesters. Since we use only datasets from fall semesters for the validation, we gave this constraint a low priority.

An even distribution of exams over the examination session is missing: Currently, the soft constraints regarding an even distribution of exams over the examination sessions are not implemented. This means the OptaPlaner does not strive for even distribution of exams during the optimization process.

5.2 Analyzing the Generated Exam Schedules

Although we use a simplified model, we can generate feasible exam schedules and do even some optimizations. It is time to look into some test results. For validating our exam scheduler, or more precisely the underlying constraint solver, we have defined two criteria.

The solver should work with different input data: Training or tweaking an algorithm with only one dataset can lead to overfitting. Overfitting means that the algorithms learn or include some specifics of the input data that do not apply to another dataset. We do not expect the algorithms used by the OptaPlanner to learn from a dataset. However, tweaking the weights of the constraints could still result in some sort of overfitting. To verify that this is not the case, we took a completely new dataset, never seen and used before, and ran it through our exam scheduler.

During development we used the dataset of the fall semester 2019. For the validation we used the dataset of the fall semester 2018. The result (Figure 5.1) shows that the new dataset works as well as the other one.

The solver should run no matter how powerful the machine is: During development, we noticed a strange behavior of the scheduler when using different machines for solving. It is clear that different machines and Java versions can start on another seed, leading to slightly different results. However, the differences were significant.

As an example, we take the solving process for the fall semester of 2018. On a powerful computer with current high-end specs, the construction phase takes about 5 minutes and gives the first score of `-8hard/-248soft`. However, when running the solving process on a less powerful machine, a high-end laptop from 6 years ago, the first result we see is `-1228hard/-110soft` after about 45 minutes. We expected the laptop to have longer for the construction phase, but the much lower score is quite surprising.

In the documentation there is no explanation of this behavior. Our guess is that the solver has internally some limits that prevent the construction phase from running too long. These limits are most likely based on the number of moves the solver can do per time interval, but we cannot tell for sure.

After the construction phase, the local search phase started. To determine whether both machines reach a similar score, we let the solvers run overnight for around half a day. The score during the local search phase on the laptop is shown in Figure 5.1. The hard and soft scores are stacked such that their sum represents the total score. The first data point is the score after the construction phase. Hence, the x-axis starts at about 45 minutes.

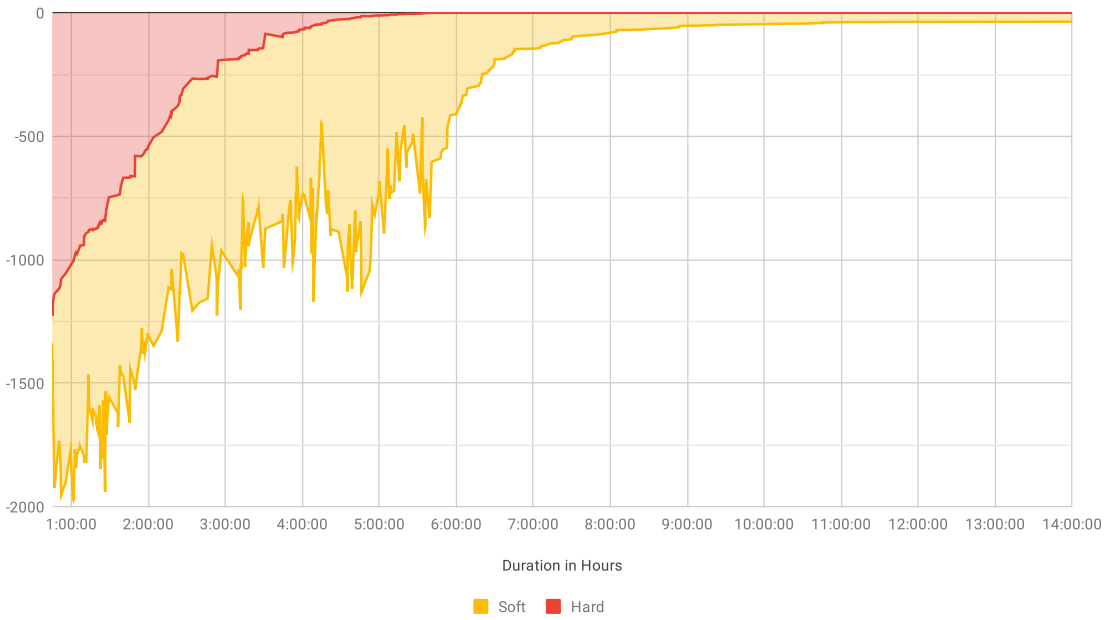

Figure 5.1: Score During Local Search Phase of Fall Semester 2018 (20 min Time Slots)
Laptop (CPU: i7-4710MQ – 4 Cores @ 2.6GHz, RAM: 16GB @ 1600MHz)

The diagram illustrates well that the OptaPlanner first minimizes the hard score. While the hard score successively approaches the value zero, the soft score jumps up and down. After about 6 hours, the hard score reaches its global minimum. At this point, the solver has found a feasible solution, and the soft constraints are left to optimize. From now on,

all further better solutions are feasible and have a higher soft score. At a certain point, finding a better solution takes quite a long time. Three hours passed between the last two results without finding a better solution. After around 14 hours, we stopped the solving process. With a final score of `-0hard/-36soft`, the laptop finds a similarly good result as our powerful test machine. Therefore, we can conclude that the solver also runs on less powerful machines.

When implementing the OptaPlanner problem model, we added the option to change the time slot duration. When changing the time slot duration, we change the number of possibilities the constraint solver can arrange the exams. Hence, the shorter the time slot, the more flexible the model becomes. With a more flexible model, it is more likely to fulfill all constraints. However, with more available options, the time to go through all possible solutions increases.

The question is whether we get better results with smaller time slots. Initially, we set the duration to 20 minutes, which is the minimum break between two exams. This time slot size works quite well across different input datasets. However, exams lasting 90 minutes waste a gap of 10 minutes, as these exams do not fit into 20-minutes-slots perfectly. To cover this corner case, we set the time slot duration to 10 minutes. However, we did not notice a significant improvement in the resulted score. This experiment shows that the real problem is not the wasted time, but much more the fact that there is no way around that some students have multiple exams on the same day. This finding is also in line with the thesis given by our customer: "**An exam schedule in which all students have at most one exam per day is impossible.**"

To verify the thesis, we ran the exam scheduler on our powerful test machine and a time slot duration of 10 minutes. The scores of the solutions found during the local search phase are shown in Figure 5.2. After running the solving process for about half a day, we ended up with a score of `-0hard/-30soft`. According to the logs, the soft score of `-30` comes alone from the constraint that makes sure as few students as possible must write more than one exam on the same day. With this result, we cannot disprove the thesis as we could not find a solution that fulfills the constraint completely, at least not in the given time with the current implementation of the exam scheduler.

To interpret the score, we recall the explanation of the constraint, given in Listing 4.4. In summary: Every time a student has to write two exams on the same day a penalty of one soft is added. So in total, it happens 30 times that a student must write two exams on the same day. To find out how many students are affected per se, further investigation is needed, but it is safe to say that at most 30 students have to write two exams on the same day. Note that no student has to write more than two exams on the same day.

Figure 5.2 illustrates also that the most significant improvements are made within the first two hours. When in need of a “quick” solution, the search for an exam schedule could be stopped around the point where the ratio of improvement over time gets too small. However, for a real exam schedule, it would probably make sense to let them run another day or even up to a week, to max out the quality of the resulting time table.

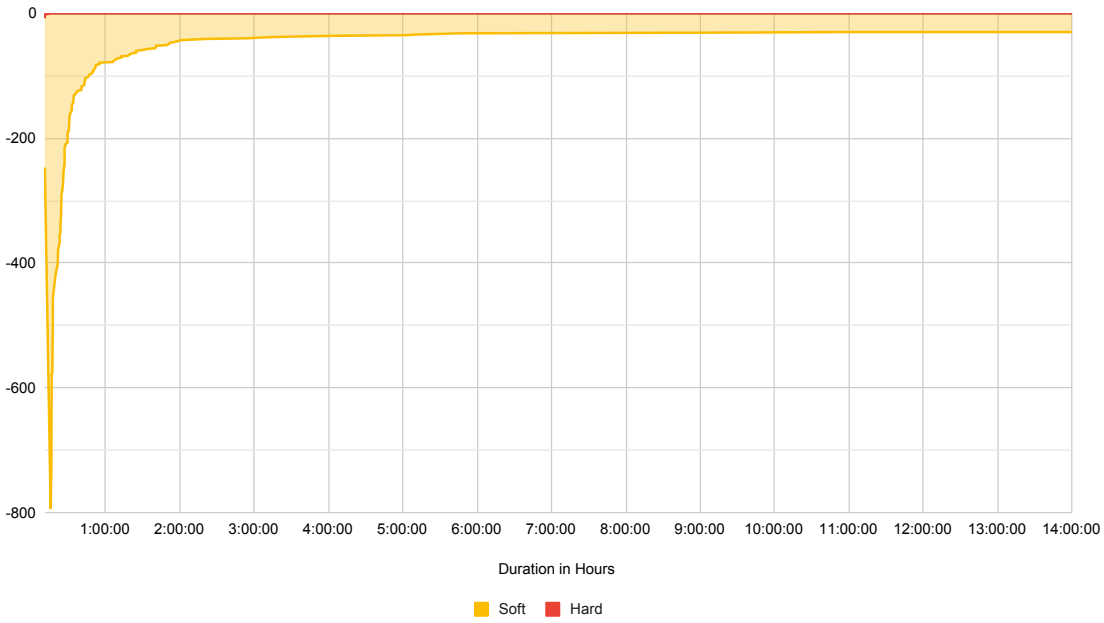

Figure 5.2: Score During Local Search Phase of Fall Semester 2019 (10 min Time Slots)
Tower PC (CPU: i9-9900K – 8 Cores @ 5GHz, RAM: 32GB @ 3200MHz)

5.3 Comparison with a Manually Created Exam Schedule

This section compares the generated exam schedule with the original one from the fall semester of 2019. For the comparison, we must analyze the original exam schedule first. However, analyzing a partially unstructured Excel file is error-prone and time-consuming. Therefore, we transformed the Excel file into our data model. This process required some manual work, but with some Python scripts, at least some parts could be automated. Nevertheless, the time spent is definitively worth it, as we can run SQL queries to analyze the exam schedule and display it with our visualization tool.

First, we want to find out how many students have two exams on the same day. For this, we wrote a SQL query. Its result is shown in Table 5.1. The query also provides information on how many days students have to write two exams. In the original exam schedule, 192 students have to write two exams on the same day. Whereas, in the generated exam schedule, there are only 28 students.

Table 5.1: Comparison of Original and Generated Exam Schedule Regarding Number of Students with Two Exams on Same Day

Number of Days with Two Exams	Number of Students with Two Exams on Same Day	
	In Original Schedule	In Generated Schedule
1	170	26
2	20	2
3	2	0
Total	**192**	**28**

Looking at the visualization of both examination schedules (Figure 5.3 and Figure 5.4) some interesting findings are revealed. In the original exam schedule, there are no exams scheduled between 12:00 and 12:45, such that all students can have lunch at an ordinary lunchtime. On the contrary, in the generated exam schedule, there are many exams during this period. The reason for this is due to the way the examination planner creates the time tables. He or she assigns the exams in such a way as that each room has more or less two exams in the morning and two exams in the afternoon. Therefore, the time range over noon remains free. Figure 5.5 shows an exam schedule for a single room and illustrates this pattern well. In contrast, the OptaPlanner uses granular time slots of 10 minutes. Moreover, no constraint tells the OptaPlanner not to put exams about noon. Therefore, exams are also scheduled during lunchtime. Should such a constraint be added in the future? We cannot answer this question, as we are not the one to decide. It is a question for the examination planning team or even the school management.

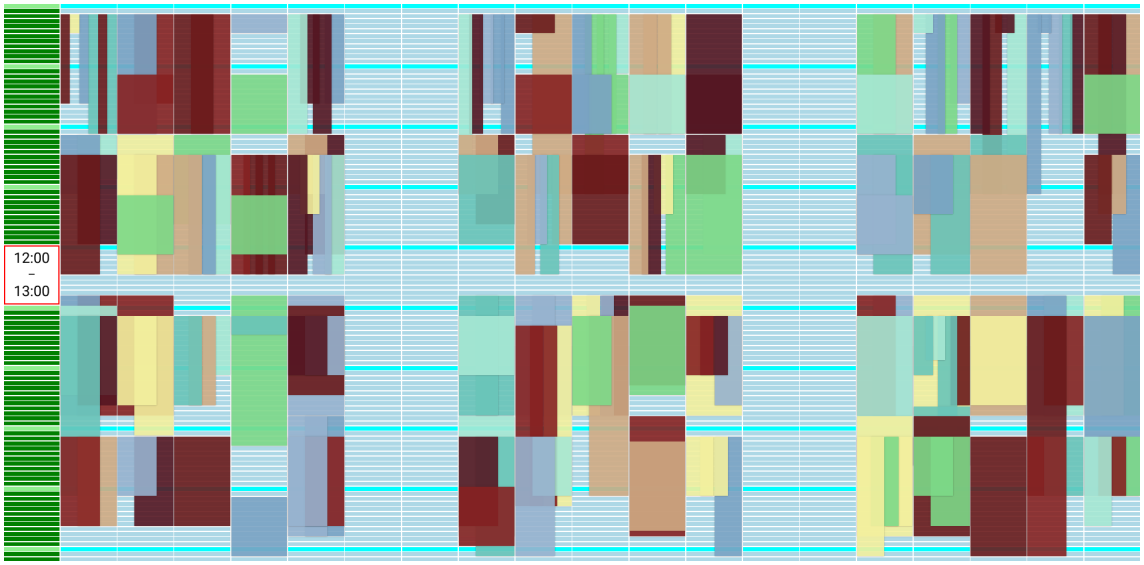

Figure 5.3: Original Exam Schedule (Fall Semester of 2019)

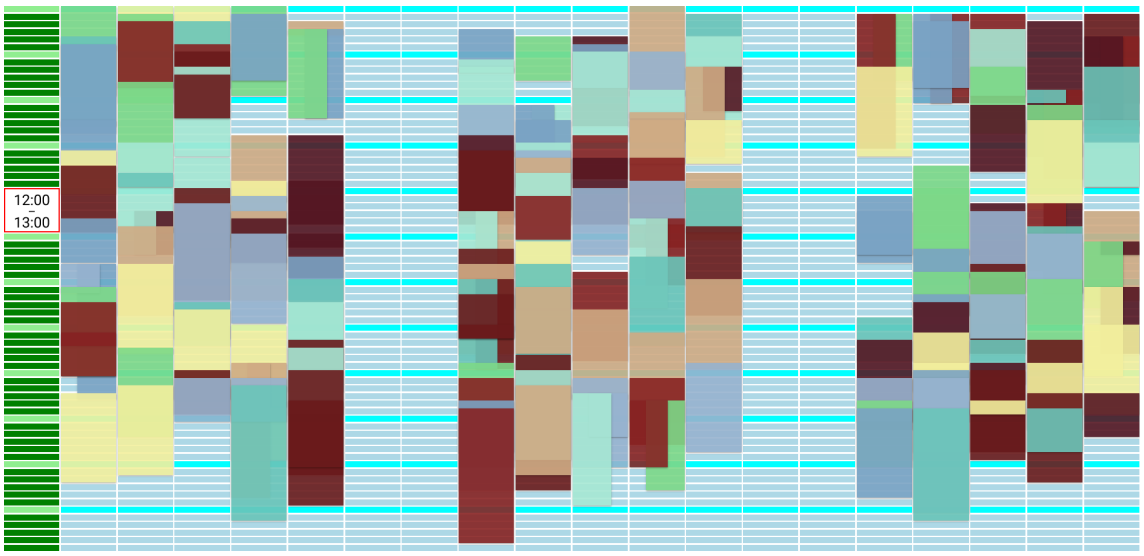

Figure 5.4: Generated Exam Schedule (Fall Semester of 2019)

When we applied the filter for room “4.101” on the original exam schedule, we noticed that sometimes two exams are scheduled simultaneously. According to the comments in the Excel file, these exams are scheduled on purpose like this. The visualization of the exam schedule for room “4.101” is shown in Figure 5.5. In total, five times two exams take place at the same time. These exams pairs are marked in red. The OptaPlanner is not allowed to schedule two exams at the same time. Hence, in this case, he had to schedule five exams at another time. Providing the information, what exams can take place at the same time could undoubtedly improve the exam schedules’ quality and should therefore be considered in the next version.

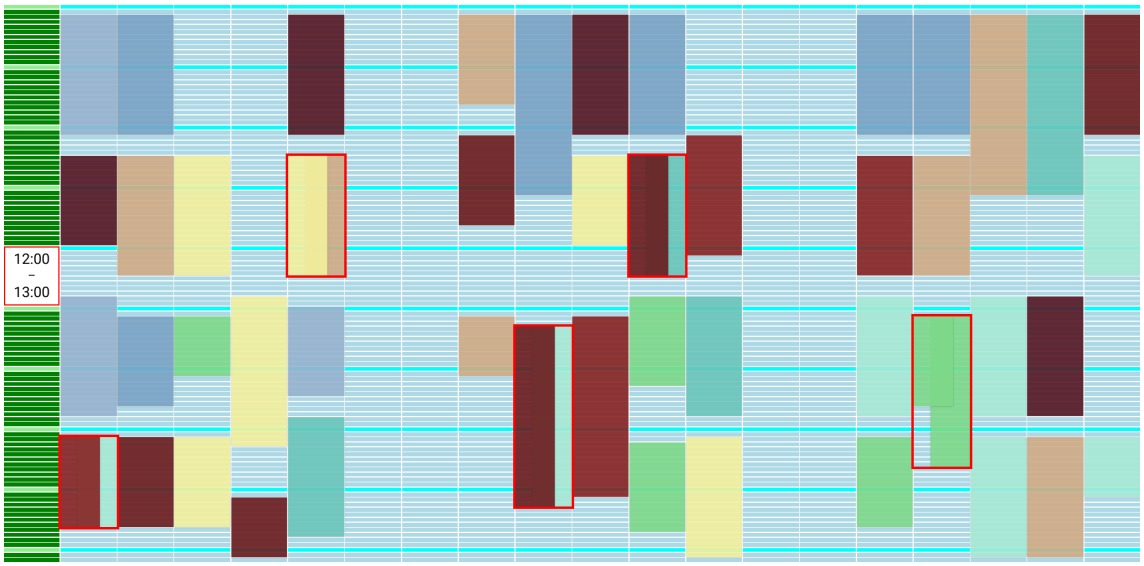

Figure 5.5: Original Exam Schedule (Fall Semester of 2019) for the Room 4.101

Chapter 6

Conclusion

With our semester project results, we have shown that creating an examination timetable for the OST can be automated. Our developed exam scheduler cannot only determine whether there is a feasible examination schedule but also optimize it regarding several constraints. We do not have a production-ready model yet. However, the comparison of a generated examination schedule with the original one has shown the potential to create better examination timetables. We laid a reasonable basis for further development. Once the model is enhanced, scheduling the exams becomes much more comfortable, and the manual work can be reduced from several hours to an absolute minimum.

In our semester project, we sketched several ideas to improve the exam scheduler. At this point, we would like to summarize them to make further development as easy as possible. We see four main topics that should be tackled in further development:

- Enhancing the problem model
- Analyzing and improving the business process
- Transforming the visualization tool into an interactive GUI
- Optimizing the solving process

Enhancing the problem model: Currently, the constraint solver uses a simplified model. The simplifications are listed in section 5.1. The problem model should be enhanced such that it fulfills all constraint as listed in section 2.2.

Analyzing and improving the business process: Some simplifications in section 5.1 are due to findings during the requirement analysis. Therefore, enhancing the problem model is not just a matter of implementation. We strongly recommend to analyze and improve the

exam scheduling process together with the examination planning team. In addition, the results regarding generated exam schedules should be discussed as well. This ensures that only constraints are implemented that are really required.

Transforming the visualization tool into an interactive GUI: Currently, the GUI is mainly used to visualize the generated exam schedule. The data import and export is done via the Swagger API documentation. All functionality required to generate examination schedules should be available in the GUI, such that the examination team can use the application. Apart from that, some advanced analytic tools such as the detailed score information would help to understand the generated time tables.

Optimizing the solving process: Finally, the OptaPlanner provides many ways to tune the solving process, including adjusting the constraint weights, using different search algorithms, and providing some advanced heuristics for the construction phase. It would be interesting to compare different configurations and evaluate the most efficient one, especially with an enhanced problem model.

Appendix A

Code Stats

A.1 Lines of Code (LOC)

Table A.1: Lines of Code (LOC)

File Type	Count	LOC
java	88	3166
java (Test Code)	42	6184
typescript	13	303
html	3	82
Total	**146**	**9735**

A.2 Test Coverage

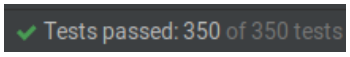

Figure A.1: Passed Tests

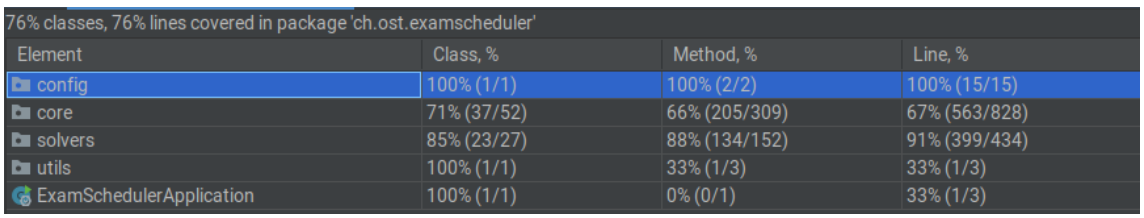

Figure A.2: Test Coverage

Glossary

Angular Angular is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. https://angular.io/ 17

Application Programming Interface An application programming interface (API) is a computing interface that defines interactions between multiple software intermediaries [Wikipedia]. 36

Jakarta Persistence API Jakarta Persistence (JPA; formerly Java Persistence API) is a Jakarta EE application programming interface specification that describes the management of relational data in enterprise Java applications. [Wikipedia]. 36

Java Stream API Method to work with collections of data in a simple stream like way, where operations and filters can be added. This feature was added in Java 8. 14, 19

JavaDoc Javadoc is a documentation generator created for the Java language for generating API documentation in HTML format from Java source code [Wikipedia]. 10

Lightweight Directory Access Protocol The Lightweight Directory Access Protocol is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol network [Wikipedia]. 36

Model View Controller Visual Basic for Applications. Model–view–controller is a software design pattern commonly used for developing User interface that divides the related program logic into three interconnected elements [Wikipedia]. 36

OST Eastern Switzerland University of Applied Sciences: In this document focused on the campus Rapperswil-Jona. 37

PostgreSQL PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance [Wikipedia]. 17

regular semester A regular semester is the semester where an exam normally takes place in. For example: The module “Objektorientiere Programmierung”, short OO, is in the first semester of your IT-studies. Therefore the regular semester of the module/exam OO is I1, which means “Informatik, 1. Semester” 5, 19

SpringBoot Spring Boot is an open source Java-based framework used to create a micro Service. It is used to build stand-alone and production ready spring applications. `https://spring.io/` 17, 18

Swagger Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON. Swagger is used together with a set of open-source software tools to design, build, document, and use RESTful web services. Swagger includes automated documentation, code generation, and test-case generation. `https://swagger.io/` 18, 32

Visual Basic for Applications Visual Basic for Applications. An event driven programming language implemented by Microsoft and integrated in the entire office suite [Wikipedia]. 36

Acronyms

API [Application Programming Interface] 8, 14, 17, 21, 22

CRUD Create, Read, Update, Delete 17

GUI Graphical User Interface 17, 24, 32

JPA [Jakarta Persistence API] 17

LDAP [Lightweight Directory Access Protocol] 9

MVC [Model View Controller] 18

RAM Random Access Memory 11

RDBMS Relational Database Management System 17

REST Representational State Transfer 18, 21

SQL Structured Query Language 14, 28

VBA [Visual Basic for Applications] 4

Abbreviations

OST Eastern Switzerland University of Applied Sciences (See: OST) 1, 4, 22, 31

List of Figures

List of Tables

List of Listings

Bibliography

[1] (2020, Nov.) OptaPlanner User Guide – A planning problem has (hard and soft) constraints. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/8.0.0.Final/optaplanner-docs/html_single/index.html#aPlanningProblemHasConstraints

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability*, 1979.

[3] (2020, Nov.) OptaPlanner User Guide – A planning problem is NP-complete or NP-hard. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/8.0.0.Final/optaplanner-docs/html_single/index.html#aPlanningProblemIsNPCompleteOrNPHard

[4] A. Müller, "Automaten und Sprachen," 2020. [Online]. Available: https://github.com/AndreasFMueller/AutoSpr

[5] T. Müller, "Constraint-based Timetabling," Ph.D. dissertation, Charles University in Prague Faculty of Mathematics and Physics, 2005. [Online]. Available: https://www.unitime.org/papers/phd05.pdf

[6] E. C. Freuder, "In pursuit of the holy grail," *Constraints*, vol. 2, no. 1, pp. 57–61, 1997. [Online]. Available: https://doi.org/10.1023/A:1009749006768

[7] (2020, Oct.) UniTime Website. UniTime.org. [Online]. Available: https://www.unitime.org/

[8] (2020, Oct.) Free Timetabling Software Website. [Online]. Available: https://lalescu.ro/liviu/fet/

[9] (2020, Oct.) Opta Planner Website. Red Hat, Inc. [Online]. Available: https://www.optaplanner.org/

[10] (2020, Nov.) OptaPlanner User Guide – Optimization algorithms overview. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#optimizationAlgorithmsOverview

[11] (2020, Nov.) OptaPlanner User Guide – @PlanningId. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#planningId

[12] (2020, Nov.) OptaPlanner User Guide – @InverseRelationShadowVariable. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#bidirectionalVariable

[13] (2020, Nov.) OptaPlanner User Guide – Cloning a solution. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#cloningASolution

[14] (2020, Nov.) OptaPlanner User Guide – Define the constraints and calculate the score. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#_define_the_constraints_and_calculate_the_score

[15] (2020, Nov.) OptaPlanner User Guide – Hill climbing. Red Hat, Inc. [Online]. Available: https://docs.optaplanner.org/7.46.0.Final/optaplanner-docs/html_single/index.html#hillClimbing