

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN

Optimizing your exam session

Supervisor:

Pierre Schaus

Master degree in Computer Science and Engineering

Thesis submitted for the

option Artificial Intelligence

by Martin Crochelet

Readers:

Hartert Renaud

Van Cauwelaert Sascha

option Networking and Security

by Romain Vanwelde

Contents

Abstract	1
Acknowledgments	3
Introduction	5
1 Constraint programming	9
1.1 Constraint Satisfaction Problems (CSP)	9
1.2 Constraint programming	9
1.3 Constraint Optimization Problems (COP)	11
1.4 Global Constraints	11
1.4.1 All Different Constraint	12
1.4.2 Global Cardinality Constraint (GCC)	13
2 Model	15
2.1 Parameters of the problem	15
2.1.1 The session	15
2.1.2 The courses	16
2.1.3 The students	17
2.1.4 The professors	17
2.1.5 The constraints parameters	18
2.2 Variables and their Domain	18
2.3 Constraints	19
2.4 Extension of the model	21
2.4.1 Variables and their Domain	21
2.4.2 Constraints	22
2.5 Optimization	23
3 The Solver	25
3.1 Model Variables	25
3.2 Constraints	26
3.2.1 Constraint 1 - No conflicts	26
3.2.2 Constraint 2 - Linking orals for student with oral exams	27
3.2.3 Constraint 3 - All slots for an oral exam must be different	27
3.2.4 Constraint 4 - Slot assignment	28
3.2.5 Constraint 5 - Slot restriction	28
3.2.6 Constraint 6 - Full availability of the professor for an oral exam	29
3.2.7 Constraint 7 - Maximum capacity for an oral slot	29
3.3 Optimization	30
3.3.1 Intermediate variables and constraints	30

3.3.2	Weighted Sum	32
3.3.3	Large Neighbourhood Search (LNS)	32
3.3.4	Variable Objective Large Neighbourhood Search (VO LNS)	34
3.3.5	Comparison between Weighted Sum and VO LNS	36
3.4	Avoiding Heavy Tail	37
3.5	Relaxation	39
3.5.1	Random Relaxation	39
3.5.2	Worst Solution Relaxation	41
3.5.3	Adaptive Neighbourhood Relaxation	42
3.5.4	Worst Solution & Adaptive Neighbourhood Relaxation	42
3.6	Comparison between the previous three methods	43
3.7	Search	44
3.7.1	Binary First Fail	44
3.7.2	Conflict Ordering Search	44
3.7.3	Comparison	45
3.8	Generated Instances	45
4	Server	47
4.1	Architectural choices	47
4.2	The REST Interface	49
4.2.1	The choice of Spray.io	49
4.2.2	The solver encapsulation	50
4.2.3	The REST protocol	53
4.3	The Meteor Application	58
4.3.1	The Meteor way	58
4.3.2	The data structure	59
4.3.3	Security	61
4.3.4	Interesting points of the Web Application	61
5	Possible improvements	65
5.1	Improvements for the solver	65
5.2	Improvements for the REST api	65
5.3	Improvements for the Meteor application	66
6	Conclusion	67
A	User Guide	71
A.1	Login	72
A.2	Creation of a new scheduling problem	73
A.3	Displaying statistics	79
A.4	Displaying Schedule	80

Abstract

The examination scheduling problem is a recurrent problem that all universities across the world encounter several times a year. Here at EPL, all student representatives meets together with the secretary in order to get as best schedule as possible. However, those schedule don't always satisfy all the students. In order to automate this fastidious process, we created the WIME application. This Constraint Programming application running on top of OscaR finds solutions to this problem, and optimizes them in order to satisfy as much students as possible. All remaining work needed to create the examination schedule is now reduced to collecting the appropriate data and let our solver find a solution.

Acknowledgments

We would like to express our sincere thanks to Renaud Hartert, our master's thesis assistant for his availability, the invaluable help he's brought us all along this academic year and his guidance.

Furthermore, we would like to place on record our thanks to our promoter, the professor Pierre Schaus for the help he's provided us in the expression of the GMG.

We would also like to give a special thank to all our friends who encouraged and supported us during all those study years.

Martin. More personally, I would like to thank Marina Sargsyan for all the support and love she's showed me during these long days of writing and implementation.

Romain. I would like to thank my family who supported me during those 5 study years, and especially my mom who didn't hesitate to come to Louvain-La-Neuve on week-ends to encourage me.

Introduction

To `schedule`[1]: "arrange or plan (an event) to take place at a particular time (from late Latin *schedula* "slip of paper")". This is what is currently done three times a year by a secretary on what has become the modern age version of a "slip of paper", an Excel file. After a dreadful journey of gathering all the professors availabilities and students enrolments, the secretary submits the problem to the student representatives of the Ecole Polytechnique de Louvain (EPL). Bearers of all the knowledge about the courses and their conflicts, they assemble to crack the schedule.

Half a day later, weary of all those discussions, the student representatives introduce their solution to their pairs. And there goes the second round of headaches: students complaining about their schedules, whining about the unfairness of the solution and (sometimes) trying to modify it by themselves. Now, one can understand all the hassle around this particular problem: *Would you trust the fate of a yearful of hard work into the hands of someone else?* Would you not try to optimize your chances of success by spacing *at maximum* your exams?

Context

With all the tumult generated by the examination scheduling problem, a cold-hearted generator seems a fair solution. Timetabling is a live example of the difficulties of a combinatorial problem: by hand, it takes a tremendous amount of time to solve. At the time of the writing, already 10 conferences have been organized by the PATAT in order to popularize the subject of courses and examination scheduling. From "self-generating multimeme algorithm"[2] to "ant algorithms"[3], the algorithms used by the different universities are diverse and efficient. In the rise of the Scala programming language and its Operations Research toolkit Oscala, the opportunity to alleviate the burden of the secretary and student representatives while harnessing the expressiveness and power of a state of the art Constraint Programming library seemed too interesting to miss.

Problem

As stated above, examination scheduling is a well known problem which has existed for years. Originally reduced to the problem of assigning a set of exams into a limited

number of timeslots; With these exams being subject to a set of constraints (both hard and soft)[4]. This problem has been proved to be NP-Hard since the basic scheduling problem can be reduced to the Graph Colouring Problem [5]. Indeed, each examination can be represented as a vertex. Each pair of courses taken by a student will be represented by edges linking both vertices. The fact that two different vertices cannot have the same color represents the clashing condition. The basic definition of the problem only defines single slot examinations while the Ecole Polytechnique de Louvain case encompasses contiguous multi-slot exams (i.e. all-day exam) and oral exams (multi-slot exam spread all over the session). For a more precise definition of the terms, please refer to chapter 2.

Motivation

In a world where all fastidious tasks tends to be atomized, it seems unthinkable that this long and complicated problem is still consuming time and energy of secretary and students year after year. Furthermore, the opportunity to delve into OscaR's complexity and the expressiveness of the constraint programming approach to solving AI problems was something we were really looking forward. Aside from the pure gain of time for the concerned parties and the intellectual curiosity, we are motivated by the realization of a full-blown application that will hopefully benefit the whole faculty.

Objectives

The motivations of our work being laid out, let us now define the objectives of our master thesis. These aspirations are twofold: on one hand, we want to implement a solver using the OscaR library that will be able to work out a valid and practical timetable for the EPL. The solver will be intended to replace the work of the student representatives by providing its solution from the compiled data (students enrolments, professors availability, courses specifications, etc.). On the other hand, we want to build an accessible application easing the usage of the solver by providing a clear and convenient way to deal with the exam session scheduling. This application should provide an efficient way to enter and modify the session specifications (compiled data) as well as administer means of browsing and measuring the solved timetable.

Approach

The optimization process will run in constraint programming on top of OscaR[6] solver. Thanks to the expressiveness of the Constraint Programming (CP) techniques, it will

be easier for the end to state his variables and session-specific constraints. Furthermore, the existence of constraints libraries in the Oscala framework will not only ease and improve the modelling of the problem but also the evolvability and maintainability of the model as a result of the usage of Oscala. This choice of implementation does not limit us to CP-only techniques as the chosen framework authorizes the hybridization (e.g. Importing the notion of neighbourhood from Local Search thanks to Large Neighbourhood Search).

The web application part of the master thesis will be composed of two parts: first, a REST shell englobing the solver implementation shielding it from the exposure to a concurrent environment. This shell will be implemented using the `spray.io`[7] (soon-to-be akka-http) framework written in Scala. This choice of implementation will promote a greater modularity and resilience to the whole application. Lastly, the web part will be implemented with the `Meteor.js`[8] framework that allows built-in seamless reactivity from backend to frontend.

Contribution

This master thesis presents the realisation of an examination scheduler, WIME (When Is My Exam). WIME is a web application that runs on top of a constraint programming solver. The WIME web application interacts with our solver to send data received from the end user, and displays in live statistics of the solutions and solutions themselves. The solver uses several Constraint Programming techniques in order to perform the search in an as efficient way as possible.

All code described or mentioned in the master's thesis can be accessed at the github repositories: solver and REST api¹ and Meteor application². A live example of the complete application can be found at <http://wime.cloudapp.net> (until the 30th of June 2015)

Roadmap

The remainder of this document is structured as follows: the next chapter provides the necessary background material by explaining basics of Constraint Programming. Then, the formal model of the EPL's scheduling problem is described. Following it is the way we implemented our solver and the optimization we enabled within. On top of that comes the description of the server architecture and how to interact with the web interface. A conclusion wraps up the main contributions of our work and we present some avenues of future work.

¹<https://bitbucket.org/mcrochelet/exam-session-scheduler-solver-rest-service/overview>

²<https://bitbucket.org/mcrochelet/exam-session-scheduler-frontend-web-application>

Constraint programming

1.1 Constraint Satisfaction Problems (CSP)

A **Constraint Satisfaction Problem** is a mathematical problem composed of a set of variables (with each variable having its own domain) and a set of constraints expressed over the variables. It is often represented by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{x_1, \dots, x_m\}$ is the set of variables, $\mathcal{D} = \{Dom_1, \dots, Dom_m\}$ is the set of corresponding domains and $\mathcal{C} = \{c_1, \dots, c_n\}$ is the set of constraints that hold on all those variables. A solution to such a problem is found when all the variables are assigned to a value of their domain so that no constraint is violated.

1.2 Constraint programming

Constraint Programming is a programming paradigm developed from (Constraint) Logic programming. It has been built to search and solve CSP in an efficient and expressive way. As stated in the course notes of Pr. Deville [9]

$$CP = Search + Model \tag{1.1}$$

Where the model is the description of the problem (the CSP, see chapter 2 for ours) and the search is the techniques and algorithms exploited by the CP paradigm. The search is composed of two main steps :

- The first step is **the propagation (or filtering)**. Responsible for the removal of impossible (having no support) values from the domains, it trims down the search space (pruning) until no more constraints can remove values from domains. Let's take a simple example.

Example 1.1. Assume x, y , two variables with domains $dom(x) = \{2, 3, 4, 5, 6\}$ and $dom(y) = \{0, 1, 2, 3, 4\}$. Two constraints hold on those variables. $c_1 \equiv x < y$ and $c_2 \equiv x > 2$. The process lasts as long as a constraint can remove a value. At the end of the filtering process described in Figure 1.1, we can see that the solution has already been found. This is not the case for all problems and, most

$$\begin{array}{c}
 x \in \{2, 3, 4, 5, 6\} \quad y \in \{0, 1, 2, 3, 4\} \\
 \downarrow c_1 \equiv x < y \\
 x \in \{2, 3, \textcolor{red}{4}, \textcolor{red}{5}, \textcolor{red}{6}\} \quad y \in \{\emptyset, \textcolor{red}{1}, \textcolor{red}{2}, 3, 4\} \\
 \downarrow c_2 \equiv x > 2 \\
 x \in \{\textcolor{red}{2}, 3\} \quad y \in \{3, 4\} \\
 \downarrow c_1 \equiv x < y \\
 x \in \{3\} \quad y \in \{\textcolor{red}{3}, 4\}
 \end{array}$$

Figure 1.1: Filtering example

of the time, we need the second step of the search described hereafter.

The filtering algorithm can be implemented with several different consistency modes: the bound consistency or the domain consistency. In the later, a value is kept in the domain if and only if it participates in a least one solution while in the bound consistency, only the bounds (lower and upper frontier of the domain) are required to satisfy their constraints. In example 1 as both modes achieve the same results, it will be more efficient to use the bound consistency since less testing will be executed. Note that since the bound consistency only validates the frontier of the domain, it might keep inconsistent values. Example 2 is bound consistent (will not filter values anymore), but not domain consistent (can still filter on some values).

Example 1.2. Assume x, y , two variables with domains $dom(x) = \{2, 3, 4, 5, 6\}$ and $dom(y) = \{2, 6\}$. One constraint holds on those variables. $c_1 \equiv x + y = 8$.

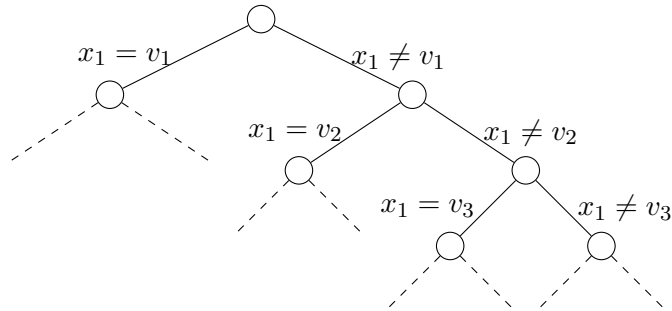


Figure 1.2: Binary Branching

- The second step is **the branching (or search)**. In this phase, we will explore the search space. Since the search space can be represented as a tree, three decisions have to be made: the branching strategy, the variable/value/branch selection heuristic and the exploration strategy. A popular branching procedure is the binary labelling: at each non-terminal node, it creates two branches: one assigns a value to an unbounded variable, the other rejects the value from its domain. A trendy variable selection heuristic is the minimum remaining values (MRV) choosing the variables whose domain comports the smallest number of (remaining) values (fail-first). The search strategy defines then the order the branches are visited; backtracking allows us to explore both side of the search tree.

1.3 Constraint Optimization Problems (COP)

The Constraint Optimization Problems are a specific case of Constraint Satisfaction Problems extended by an optimization function $f(X)$ (and hence modelled quite similarly). The goal changes from the research of one (or all) valid solution(s) to the pursuit of an optimal solution with regard to $f(X)$. Accordingly, once the first solution is found, a constraint enforcing the objective score of the next solution to be strictly higher (in case of maximization) than the ongoing best objective is included. Each time a new solution is found (which will be better thanks to the new constraint), the value of the objective is updated. For the case of an maximization problem, the constraint can be expressed as $c_{opt} \equiv f(\mathcal{X}) > currentBest$ and each time a solution is found, we update $currentBest = f(\mathcal{X})$

1.4 Global Constraints

As stated in [10]:

[...] a constraint C is often called “global” when “processing” C as a whole gives better results than “processing” any conjunction of constraints that is “semantically equivalent” to C

More strictly, C. Bessière and P. Van Hentenryck define three notions of globality: the semantic, operational and algorithmic ones. We are here interested by the operational globality since both the **AllDifferent** and **GCC** constraints can be decomposed into a clique of simple arithmetic constraints [11]. More intuitively, we can understand a global constraint as a constraint holding on an unspecified number of variables. We explain the two global constraints we use in the modelling of the EPL’s problem: the **AllDifferent** and **GCC** constraints.

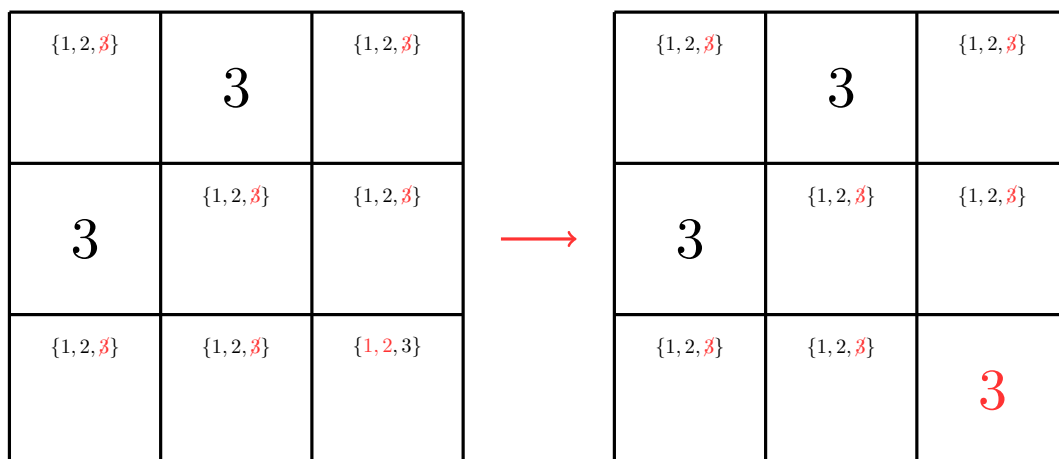


Figure 1.3: AllDifferent constraint

1.4.1 All Different Constraint

The **AllDifferent** is an often used constraint defined as holding if and only if all the variables participating in its scope are assigned different values [12] (a formal definition can be found in [13]).

Example 1.3. Let's take the square drawn hereafter (Figure 1.3). There are 9 variables, represented by tiles. All variables have the domain the set $\{1, 2, 3\}$. The only existing constraint is an **AllDifferent** on each row, and on each column.

With the assignment on the left part, we can observe the filtering on the other variables' domains. Since the **AllDifferent** takes the entire row (resp. column) into account, bound/domain consistency process will detect that an assignment of the bottom-right tile to either 1 or 2 would render the problem infeasible. The value is accordingly assigned to 3 and the filtering finishes (there are no more values to remove from the domains).

As stated above, the **AllDifferent** is not operationnaly global: the obtained filtering would not have been possible with a decomposition of the constraint in a clique of binary constraints (domain of last tile would still be $\{1, 2, 3\}$).

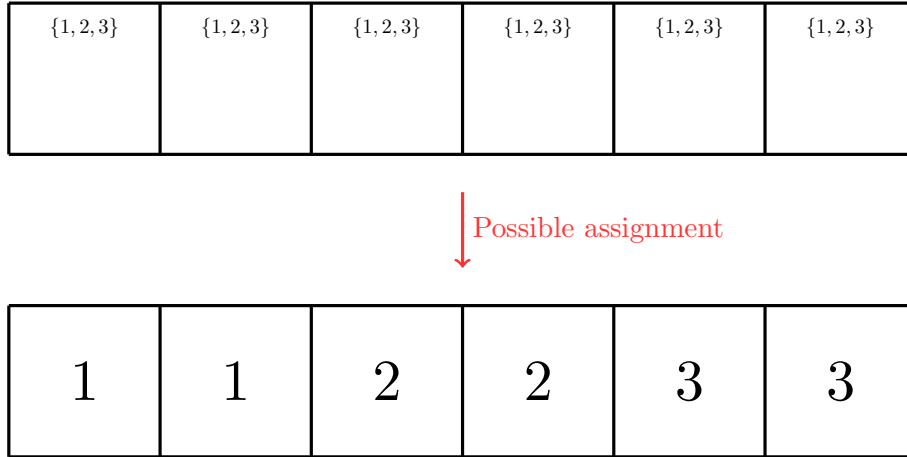


Figure 1.4: GCC constraint

1.4.2 Global Cardinality Constraint (GCC)

The GCC is a generalization of the previous constraint in the sense that it does not enforce uniqueness of a value in the variables (of its scope) but rather enforces that the value i occurs between l_i and u_i times for the variables participating in its scope [11].

Example 1.4. Let's consider 6 students. Each of them is represented by a tile on the Figure 1.4. They all need to present an oral examination which is given on three different slots (1, 2 and 3). For each student, the variable represents the slot in which he will present the oral examination. Since the slot lasts only an hour, no more than two students are allowed in an examination slot. This will be expressed by the GCC constraint. It will receive the array of all the variables $X \equiv x_i \mid \forall i \in 0, \dots, 5$, the array of $l_i \equiv [0, 0, 0]$ and the array of $u_i \equiv [2, 2, 2]$.

Model

In this chapter, we will describe the model we implemented, the variables we used, their domains, and the constraints that hold them. We begin by introducing the simple model and follow on to its extension.

2.1 Parameters of the problem

We begin by introducing the general parameters describing the problem then move on to the courses specific ones. The students, professors and constraints parameters are finally introduced and precede the modelling variables used in the solver.

2.1.1 The session

First, let us begin with the general parameters: any examination session is characterized by the number of days it spans and the number of possible slots per day (e.g. the maximum number of different exams a student can have in the same day). For modelling purposes, we define the total number of sessions which is simply the product of the two former variables.

```
/* session values */
nDays          // Is the number of days of the session
nSlotPerDay     // Is the number of slots in a day. If we want an
                // examination scheduled either in the morning
                // or in the afternoon we set this variable to 2.
nSlots          // Is the number of slots in the entire session
                // = nDays * nSlotPerDay

/* Ranges */
rDays = {0 until nDays} // Is the range of all days in the session
rSlots = {0 until nSlots} // Is the range of all slots in the session
```

Since no examination can be scheduled on Sundays, we filter out any corresponding values from `rDays` and `rSlots` and store the new ranges in two parameters: the `availableDays` and the `availableSlots`.

```

availableDays = rDays \ Sundays      // Is the range of all days where
                                     // an examination can be scheduled
availableSlots = rSlots \ SundaysSlots // Is the range of all slots where
                                     // an examination can be scheduled\\

```

2.1.2 The courses

In the EPL's case of exam scheduling, a course can be (exclusively) of one of the three following categories: Unislot, Multislot or Oral. In order to support the different categories, we introduce a new type **subcourse** whose role is to represent the different slots needed by a course to be scheduled.

Definition 2.1. (Course). A course is a non empty set of subcourses.

Definition 2.2. (Subcourse). Atomic building bloc of a course, each subcourse represents one slot taken by the course examination.

Notation 2.1. (Subcourse notation). $courseLabel^{slotID}$ where $courseLabel$ is the label of the course and $slotID$ is the slot position of the subcourse.

Definition 2.3. (Courses). $Courses$ is the set of all the courses.

Example 2.1. $LINGI2990 \in Courses$ needs 4 slots for its examination, it will then be composed of 4 courses represented as:

$$LINGI2990 \equiv \{LINGI2990^1, LINGI2990^2, LINGI2990^3, LINGI2990^4\}$$

Definition 2.4. (Subcourses). $Subcourses$ is the set of all subcourses.

Example 2.2.

$$Subcourses_{LINGI2990} \equiv \{LINGI2990^1, LINGI2990^2, LINGI2990^3, LINGI2990^4\}$$

Definition 2.5. (Unislots, Orals and Multislots). $Unislots$, $Orals$ and $Multislots$ are sets of courses respectively containing only courses of category Unislot, Oral and Multislot. Three notes are worth mentioning:

- Since the examination rules of the EPL state that a professor can not ask a student to make two performances for the same course on different days of the session:

$$\forall course \in Multislots : |course| \leq nSlotPerDay$$

- Since by definition unislot courses only require one slot:

$$\forall \text{course} \in \text{Unislots} : |\text{course}| = 1$$

- Since the membership to one of the category is exclusive:

- $\text{Unislots} \cap \text{Orals} = \emptyset$
- $\text{Unislots} \cap \text{Multislots} = \emptyset$
- $\text{Orals} \cap \text{Multislots} = \emptyset$

From the definitions here-above, we can deduce that:

$$\text{Courses} = \text{Unislots} \cup \text{Orals} \cup \text{Multislots}$$

Example 2.3. Suppose *LINGI2262* and *SINF2275* are two courses of the oral category needed respectively 4 and 2 slots. Then

$$\begin{aligned} \text{Orals} &\equiv \{\text{LINGI2262}, \text{SINF2275}\} \\ &\equiv \{\{\text{LINGI2262}^1, \text{LINGI2262}^2, \text{LINGI2262}^3, \text{LINGI2262}^4\}, \\ &\quad \{\text{SINF2275}^1, \text{SINF2275}^2\}\} \end{aligned}$$

Definition 2.6. (Written). $\text{Written} \equiv \text{Unislots} \cup \text{Multislots}$

2.1.3 The students

Definition 2.7. (Students). *Students* is the set of all student that are enrolled in at least one examination during the session.

Definition 2.8. (enrolled). *enrolled* is the set of all pairs (student, course) such that the student is registered in the course. If the course is a multislot, the student is enrolled in all subcourses of the course, while if it is an oral, the student is only enrolled in the course (and not in all subcourses of the oral).

2.1.4 The professors

Definition 2.9. (Professors). *Professors* is the set of all professors giving a course in which at least one student is enrolled.

Definition 2.10. (teachesWritten). *teachesWritten* is the set of all pair (professor, subcourse) such that the professors gives the course corresponding to subcourse (and the course needs a written examination).

Definition 2.11. (teachesOral). *teachesOral* is the set of all pair (professor, subcourse) such that the professors gives the course corresponding to subcourse (that the course needs an oral examination).

2.1.5 The constraints parameters

Definition 2.12. (assignment). *assignment* contains pairs of (subcourse, slot) representing the assignment of a specific exam to a specific slot. Those values represent courses that need to be scheduled on a specific date for external reasons. (e.g. a course scheduled by the end user: the examination needs to be held the first/last day of the session, the examination needs really specific material which is only available on a specific day, ...).

Definition 2.13. (restriction). *restriction* contains pairs of (subcourse,slot) representing the restriction for a specific exam to happen on a specific slot. Those values represent courses that can't be scheduled on specific dates due to external reasons. (e.g. the professor is unavailable on specifics days, the specific material isn't available on some days, ...).

Definition 2.14. (nStudentPerSlot). *nStudentPerSlot* contains, for each course that is an oral, the maximum number of students admissible in the same slot.

2.2 Variables and their Domain

Variable 2.1. (date). *date* is the set of variables representing slots scheduled for at least one subcourse. *date* associates to all subcourses a variable corresponding to the slot assigned for the corresponding examination. Each of those variable has a domain that is the set of all available slots.

$$\forall course \in Courses, \forall subcourse \in course : \quad date_{subcourse} \in availableSlots$$

Variable 2.2. (*studentDate*). *studentDate* is the set of variables representing slots scheduled for at least one course followed by a student.

studentDate associates to each pair of (student, course) (such that student is enrolled in course) a variable corresponding to the slot in which the student will present the examination of his course. If the examination is written, all students inscribed in this examination will present the exam at the same slot(s) (which is the $date_{course}$). But if the examination is an oral one, each student will present their own oral examination in a specific slot at which the examination is given.

$$\forall (student, course) \in enrolled \quad : \quad studentDate_{(student, course)} \in availableSlots$$

Notation 2.2. $date_{course} = \{\forall subcourse \in course, date_{subcourse}\}$

2.3 Constraints

Now that the basic variables of the model have been laid out, we describe all the constraints that hold on previously defined variables.

Constraint 1. No conflict

Each student has all his exams in different slots.

$$\forall student \in Students : AllDifferent(courseSet) \\ with \ courseSet \equiv \{studentDate_{course} \mid (student, course) \in enrolled\} \quad (2.1)$$

In order to improve the resilience of the solver, we define this constraint as a soft constraint. A soft global constraint is basically an extension of its hard version with a new cost variable representing its degree of violation [14]. A definition of the variable-based, minimisation soft all different can also be found in [14]. The violation cost is then entered in the optimisation function in order to be minimized.

Constraint 2. Linking orals for student with oral exams

For each oral exam, the slot in which the student presents the exam must be a slot where the exam is scheduled.

$$\forall course \in Orals : \forall \{student \mid (student, course) \in enrolled\} : \\ studentDate_{(student, course)} \in date_{course} \quad (2.2)$$

Constraint 3. All slots for a multislot exam must be contiguous

$$\forall course \in Multislots : \forall i \in \{1, \dots, |course|-1\} : date_{course^{i+1}} = date_{course^i} + 1 \quad (2.3)$$

Constraint 4. All slots for a multislot exam must be held in the same day
 The verification can be done only by looking if the first and last subcourse are held on the same day.

$$\forall course \in Multislots :$$

$$date_{course^1} / nSlotPerDay = date_{course^n} / nSlotPerDay \quad (2.4)$$

where $/$ is the integer division mapping a slot to a day and $n = |course|$, the number of slots required by the multislot.

Constraint 5. All slots for an oral exam must be different

For each course needing an oral examination, all the slots scheduled for said course must be different

$$\forall course \in Orals : AllDifferent(course) \quad (2.5)$$

Note : We may be tempted to replace this constraint by a more restrictive one:

$$\forall i, \forall course \in Orals : date_{course^i} < date_{course^{i+1}}$$

However, this new constraint could not hold simultaneously with the following constraint without modifying the signification of the model. Indeed, since next constraint allows us to fix a subcourse to a specific slot, it would implicitly express that all other slots needs to be scheduled *after* the assigned slot.

Constraint 6. Slot assignment

Some exams can be preassigned to specific slots. Those pre-assignments are stored in the *assignment* set.

$$\forall (subcourse, slot) \in assignment : date_{subcourse} = slot \quad (2.6)$$

Constraint 7. Slot restrictions

Some exams can be excluded from specific slots. Those restrictions are stored in the *restriction* set.

$$\forall (course, slot) \in restriction, \forall d \text{ in } date_{course} : d \neq slot \quad (2.7)$$

Constraint 8. Full availability of the professor for an oral exam

Professors need to be present at their oral examination and they cannot be available somewhere else. It is thus needed to have no other examination for a professor at the same time as one of his/her oral examinations.

$$\begin{aligned} \forall \text{professor} \in \text{Professors} : \forall (\text{professor}, \text{subcourse}) \in \text{teachesWritten} : \\ \text{date}_{\text{subcourse}} \notin \text{date}_{\text{oralSet}} \\ \text{with } \text{oralSet} \equiv \{ \text{date}_{\text{course}} \mid (\text{professor}, \text{course}) \in \text{teachesOral} \} \end{aligned} \quad (2.8)$$

Constraint 9. Maximum capacity for an oral slot

Each oral exam has a specific number of slots. Each slot has a maximum capacity. The number of students scheduled to a specific slot cannot exceed this limit.

$$\begin{aligned} \forall \text{oral} \in \text{Orals}, \forall \text{suboral} \in \text{oral} : \\ \sum_{\text{student}} (\text{studentDate}_{(\text{student}, \text{oral})} == \text{date}_{\text{suboral}}) \in \{0, \dots, \text{nstudentPerSlot}(\text{oral})\} \end{aligned} \quad (2.9)$$

2.4 Extension of the model

In order to represent the reality more faithfully, we need to extend the ongoing model. More than slots, the real key of the problem is the number of days the students have between each of their exams. We define here new variables and constraints built on the basic model.

2.4.1 Variables and their Domain

Variable 2.3. (dayDate). *dayDate* is the set of variables representing days scheduled for at least one subcourse

dayDate associates to all courses a variable corresponding to the day assigned for the corresponding examination. Each of those variable has a domain that is the set of all available days.

$$\forall \text{course} \in \text{Courses} : \text{dayDate}_{\text{course}} \in \text{availableDays}$$

Variable 2.4. ($studentDayDate$). $studentDayDate$ is the set of all the variables representing days in which a student presents his examination.

$studentDayDate$ associates to each pair of (student, course) (such that student is enrolled in course) a variable corresponding to the day in which the student will present the examination of his course. If the examination is a written examination, all student inscribed in this examination will present the exam at the same day (which is the $dayDate_{course}$). But if the examination is an oral one, all students will present their own oral examination in different days during which the examination is given.

$$\forall (student, course) \in enrolled \quad : \quad studentDayDate_{(student, course)} \in availableDays$$

2.4.2 Constraints

Those new variables imply new constraints on top of those previously defined.

Constraint 10. Linking basic variables with new ones.

To better understand how the linking will be done, let's take a simple example: a session of 7 days, and having 2 slots per day. The Figure 2.1 illustrates the matching between slotsID and daysID.

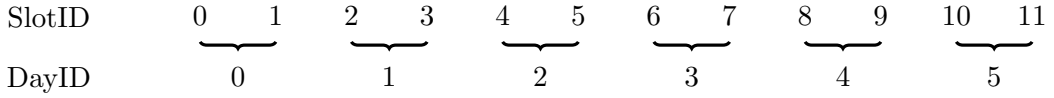


Figure 2.1: Linking slotIDs with dayIDs

$$\begin{aligned} &\forall course \in Courses, \forall subcourse \in course : date_{subcourse} \in \\ &[dayDate_{subcourse} * nSlotPerDay, \dots, (dayDate_{subcourse} + 1) * nSlotPerDay[\end{aligned} \quad (2.10)$$

$$\begin{aligned} &\forall (student, course) \in enrolled, \forall subcourse \in course : studentDate_{subcourse} \in \\ &[studentDayDate_{subcourse} * nslotPerDay, \dots, \\ &\quad (studentDayDate_{subcourse} + 1) * nSlotPerDay[\end{aligned} \quad (2.11)$$

2.5 Optimization

Now that we have defined all the constraints that our model needs to satisfy, and since our problem is a COP, we move on to the definition of the objective function(s). Several sub-objectives have been implemented and are described below. All of those try to maximize the comfort of students.

Definition 2.15. (The Global Minimal Gap (GMG)). The Global Minimal Gap (GMG) is the minimal gap (number of days or slots) that can be found for a student participating in the session. $GMG \equiv \min(studentsMinimalgap)$ with $studentsMinimalGap \equiv \min(\{\forall student \in Students, minimalGap(student)\})$ where $minimalGap(student)$ is the minimum value of the matrix representing the differences between the scheduled slots (or days) of the courses (and subcourses) the student is enrolled in.

Sub-Objective 1 - Having the smallest global minimal gap (GMG)

All students have gaps between their examinations (i.e. the number of free slots - or days). For every student, we will retrieve the smallest gap, and then retrieve the smallest gap among all students. This GMG will be the first value that we will optimize since it improves the comfort of all students globally. We ensure this way that the global comfort is improved before the individual one. All gaps can be expressed either in terms of slots (the number of free slots), or in terms of days. If the GMG is expressed in terms of slots, we will speak about the GMSG (Global Minimal Slot Gap), and if it's expressed in terms of days, we will speak about GMDG. Most of the time, we will use the GMDG (Global Minimal Day Gap) since students care about the number of days between their examinations (rather than the number of slots).

Sub-Objective 2 - Having as few students as possible in the GMG

Once the GMG has optimized enough, we begin the optimization on the number of students having the GMG as their own minimal gap (for the search strategies, please refer to chapter 3, section 3.7). It may be physically not possible for some students to have a better minimal gap than the GMG however we still want to improve, if possible, the comfort of the other students.

Sub-Objective 3 - Trying to improve the minimal gap of all remaining students

With the optimization of the two previous Sub-Objectives, we typically have the distribution of students based on their minimal gap as represented here under (Figure 2.2). We will try to push this distribution as much as possible to the right (in other words, try to increase the minimal gap between 2 exams for each student).

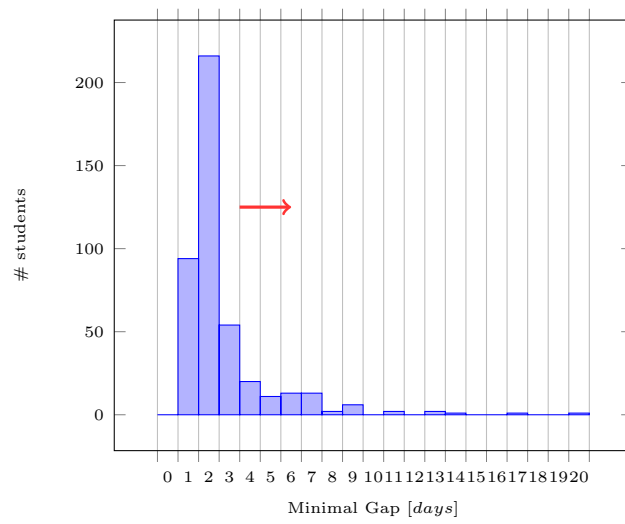


Figure 2.2: Barplot of number of students based on their minimal gap

The Solver

Previous section has presented in details the model with the variables, their domains, constraints that hold on them, and objectives on which we will optimize our solutions. Let's see now how our solver handles the previously described problem.

3.1 Model Variables

```
/* Course scheduling */
val unislotsSchedule: Map[String, CPIntVar]
val multislotsSchedule: Map[String, Array[CPIntVar]]
val oralsSchedule: Map[String, Array[CPIntVar]]

/* Student scheduling */
val studentSchedule: Map[String, Map[String, Array[CPIntVar]]]
```

All `CPIntVar` are variables having as domain all the possible slots for the examinations.: the domain is the `availableSlots` range of the previous chapter.

Unislot examinations are simple since they only need one variable to represent the scheduled slot (remember, the size of the set of subcourses is one for the unislots). All variables in `unislotsSchedule` map have the `courseID` as key and the variable as value.

Musltislots exams are more interesting, indeed, once we fix one of the slots of the examination, all others are fixed too. This means that variables are not independent. This allows us to optimize the problem via its definition: instead of creating lots of independent variables and linking them later with constraints, we create a variable for the first slot, and views with offset (from the first variable) for other slots ($variable_1 + index$). This permit us to limit the number of useless `CPIntVar`. All variables in `multislotsSchedule` map have the `courseID` as key and an array of variables as value. The first value of the array is the `CPIntVar`, the following values are the views.

For oral examinations, two specific cases need to be addressed.

- Some professors do not impose any constraints on their scheduling. So the slots can be spread all over the session. This gives us more flexibility in accepted solutions. This first case (examinations spread all over the session) is modeled with all different CPIntVar.
- On the other hand, some professors want all oral examinations to be presented on slots of some consecutive days. This second case (consecutive slots) is modeled as for multislot examinations, with the first slot represented by a CPIntVar, and following ones represented with a view with offset of this first variable.

All variables are in `oralsSchedule` map have the `courseID` as key and an array of variables as value. Content of the array is either lots of different variables, or, as for multislots, one variable and many views of this variable.

`studentSchedule` represents the examination schedule for each student. It is a map with `student` as key, and a new map as value. This new map represents all exams that this student needs to take, and their variables (thus, `courseID` as key, and the variable as value). In case of unislots exams, or multislots exams, we just put a reference to the previously created variable. But if the exam for the course is an oral examination, we have to create a new variable representing the slot in which the student will present the examination.

3.2 Constraints

3.2.1 Constraint 1 - No conflicts

Our model states that no student can have two examinations at the same slot. We could face an instance which is overconstrained, and on which this statement couldn't be held. We thus decided to set this constraint as a soft constraint. We will store the violation score for each student using the map `studentViolation`.

```
val studentViolation: Map[String, CPIntVar]
...
/**
 * C1 : Each and every student has all his exams in different slots
 */
for (student <- data.students)
  add(softAllDifferent(studentSchedule(student).values.flatten.toArray,
    studentViolation(student)))
```

3.2.2 Constraint 2 - Linking orals for student with oral exams

For this constraint, we iterate on all students, and all oral exams they are enrolled in. We then add to the solver an `or` constraint, which stipulates that one of the following conditions must be true. The conditions are for each slot in which the oral examination is held, the slot equals the slot in which the selected student presents his exam.

```
/**
 * C2 : Slot in which a student presents an oral exam must be one of the
 *       slots in which the oral exam is offered
 *       Linking decision variable of students with decision variables for
 *       oral exams
 */
for (student <- data.students)
  for (course <- data.coursesFollowedBy(student); if
      (oralExams.contains(course)))
    add(or(oralSchedule(course).map {
      potentialSession => potentialSession ==
        studentSchedule(student)(course)(0)
    })))
```

3.2.3 Constraint 3 - All slots for an oral exam must be different

Here, we set an `allDifferent` condition on all variables of an oral examination.

```
/**
 * C3 : All oral session slots must be different for a specific course.
 */
for (course <- oralExams)
  add(allDifferent(oralSchedule(course)))
```

A simple `allDifferent` constraint doesn't allow us to avoid symmetric search spaces. Indeed, all solutions where slots assigned for oral examination are the same but in a different order are strictly equivalent. It is thus sufficient to explore only one of those search spaces in order to reduce the global search space. This is done by adding a constraint that forces the oral slots to be strictly growing. But, as saw in the model, we need to exclude from this ordering variables that was already assigned by external means.

```
/**
```

```

* C3bis : All non assigned slot must be in a strictly ascending order.
* This constraint breaks the symmetry between slots for oral examination
* Swapping the allocated slots for different oral examination (of a same
  course)
*
*/
for ((exam, sessions) <- data.sessionFixedFor)
  for (i <- sessions.size + 1 until getCPIntOfCourse(exam).size)
    add(getCPIntOfCourse(exam)(i) < sessions(i))

```

3.2.4 Constraint 4 - Slot assignment

This constraint assigns some slots to some examinations. We first ensure that either we only fix one slot (for unislots or multislots), or we fix multiple slots for an oral examination. For orals examinations, we also ensure that the number of fixed slots isn't bigger than the number of available slots.

```

/**
* C4 : Slots assignment
*   Some exams are assigned to specific slots
*
*/
for ((exam, sessions) <- data.sessionFixedFor)
  if ((oralExams.contains(exam) || sessions.size == 1) &&
      data.numberOfSlotsForOral(exam) >= sessions.size)
    for (i <- 0 until sessions.size)
      add(getCPIntOfCourse(exam)(i) == sessions(i))
  else
    println(exam + "tries to block " + sessions.size + " slots, but only 1
      can be blocked")

```

3.2.5 Constraint 5 - Slot restriction

After slot assignment, we need to accommodate slot restriction. For a specific course, we iterate on all its restricted slots, and we ensure none of *course slot* coincides with one of those.

```

/**
* C5 : Slot restriction

```

```

*      Some exams are restricted to specific slots
*/
for ((exam, sessions) <- data.sessionUnavailableFor)
  for (restrictedSession <- sessions; examSession <- getCPIntOfCourse(exam))
    add(examSession != restrictedSession)

```

3.2.6 Constraint 6 - Full availability of the professor for an oral exam

```

/**
 * C6 : A written exam cannot be in same slot than an oral given by the
 *       same prof
 *       (full availability of the prof for the oral exam)
 */
for (profId <- data.writtenGivenByProfessor.keys) {
  val oralVars = data.oralsGivenByProfessor.getOrElse(profId, Set())
    .flatMap(oralsSchedule(_))

  for (written <- data.writtenGivenByProfessor(profId)) {
    val writtenVar = getCPIntOfCourse(written)
    add(allDifferent(oralVars ++ writtenVar))
  }
}

```

3.2.7 Constraint 7 - Maximum capacity for an oral slot

For each course, we now ensure that the number of students in an oral slot does not exceed the maximum number of students allowed in a given slot. This is done using the GCC constraint. This constraint ...

```

/**
 * C7 : Ensure that number of students presenting an oral in session i
 *       [0..nSession[ belongs to range [0, nStudentPerOralSession[
 */
for (course <- oralExams)
  add(gcc(oralForStudentsSchedule(course), 0 until nSession,
    Array.fill(nSession)(0),
    Array.fill(nSession)(data.nStudentPerOralSession(course))))

```

3.3 Optimization

3.3.1 Intermediate variables and constraints

Since we want to optimize the comfort for students, we have to know the gap size between examination they are enrolled in. For this purpose, we may want to create an array of variables representing the gap between all possible pairs of exams for the courses. But most of those variables are useless since there exist lots of pairs with no students enrolled in both courses. So there is no need to compute all those variables. Figure 3.1 represents the actual pairs of courses that are taken by student. We can see that lots of pair aren't taken by students and therefore, this justifies our creating only the variables representing a valid combination. With 236 courses, 27.730 possible pairs exist, and among those, only 2.034 pairs are actually taken by students. This represents 7,34% of the total number of possible pairs. This data corresponds to master students enrolled at UCL - EPL in 2014-2015 for the session of June 2015..

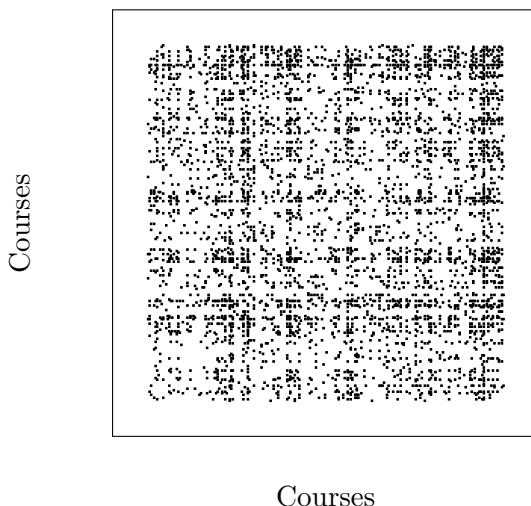
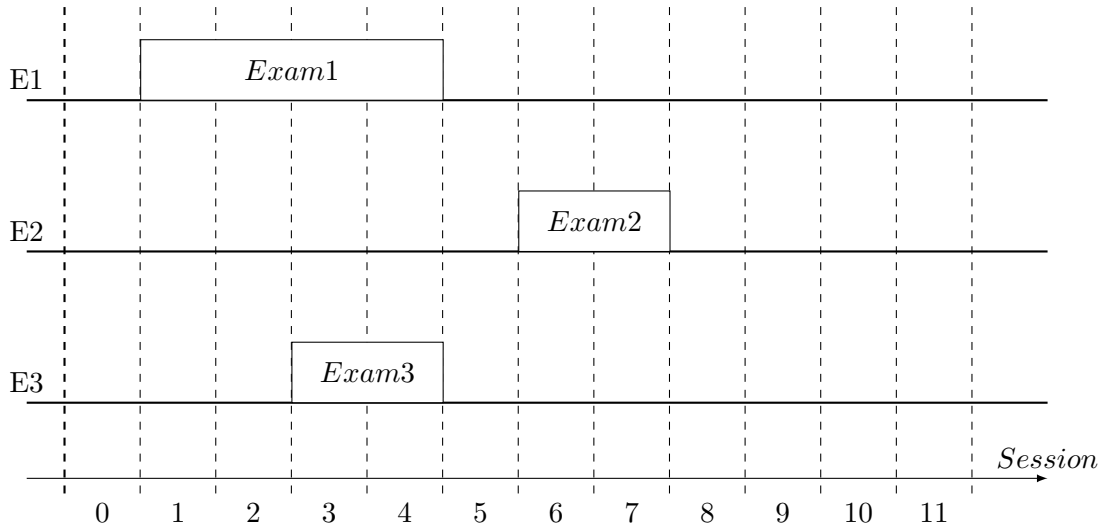


Figure 3.1: Matrix of course enrolment

Another way to proceed is to create a map where, for each student, we create a variable for each pair of courses he is enrolled in (`spaceBetweenCoursesOf`). But in this case, we may create lots of useless variables. Indeed distances (gaps) between two written course exams will always be the same, no matter who is the student. To avoid creating a huge number of useless variables, each time we want to create a variable for distance between two written courses, we verify that a corresponding variable hasn't yet been created (In the `spaceBetweenWrittenCourses` map). If so, we just set a reference to that variable.

```
// Contains space between all written course exams (identical for all
// students)
val spaceBetweenWrittenCourses: Map[(String, String), CPIntVar] = Map()
// Contains space between followed courses of each students
val spaceBetweenCoursesOf: Map[String, Map[(String, String), CPIntVar]] =
  Map()
```

In the map described previously, we will store the gap value. Since we don't know the sequence of the exams, nor whether they will overlap, computing this gap size between them isn't trivial. E.g. for **exam1** and **exam2**, we will have to compute gap between start of one and end of the other exam on both sides ($begin_1 - end_2$ and $begin_2 - end_1$). If both values are negative, it means that exams overlap, and we set the gap value to 0. Otherwise, one value will be negative, and the other will be positive, we just have to retrieve the positive one. A simple example is shown at Figure 3.2.



e_A	e_B	$begin_{e_A} - end_{e_B}$	$begin_{e_B} - end_{e_A}$	gap
(Exam1, Exam2)		-6	2	2
(Exam1, Exam3)		-3	-1	0
(Exam2, Exam3)		2	-4	2

Figure 3.2: Computing the minimal gap for examination

Objective 0 - Avoiding clash for students

Since we set the first constraint as a **softAllDifferent**, the first objective will thus become avoiding conflict between students. We will thus try to set the violation variable score to 0.

3.3.2 Weighted Sum

A first optimization function that could be used is a weighted sum. We first created a weighted sum that expresses those requirements :

1. No conflict for students
2. Having the GMG as big as possible
3. Having as few as possible students having this GMG as minimal gap
4. Improving the minimal gap of remaining students

Since having the GMG set to 0 is equivalent to having conflicts, optimizing the GMG will also first try to remove conflicts for students. In order to ensure that a solution with a better objective1 than another will always be better in the weighted sum, no matter what the other less important objectives are, we multiply the value of objective1 by the upper bound of the less important objectives.

```
// weighted sum
maximize(objective1 * UBOjective2 - objective2)
```

3.3.3 Large Neighbourhood Search (LNS)

In order to have the best results as fast as possible, we can't stay with a model fully expressed in CP, but we have to include the efficiency of Local Search. Figure 3.3 illustrates the part of the search space which could be covered. The space is so huge that we wouldn't be able to explore the entire search space. We will be stuck in a small part of the this space, with the inability to explore the rest of the search space.

Large Neighbourhood Search (LNS) is a possible way to avoid being stuck in the same part of the search space during the entire search. LNS works as follow :

1. Find a solution.
2. Relax the last solution found. This means that we will fix only some variables of the last found solution.
3. Restart the search subject to some search limit. Most of the time expressed with a number of backtrack or a time limit.

This way of exploring the search space (jumping inside the search space) allows us to find better solutions faster. The choice on the relaxation function performed and

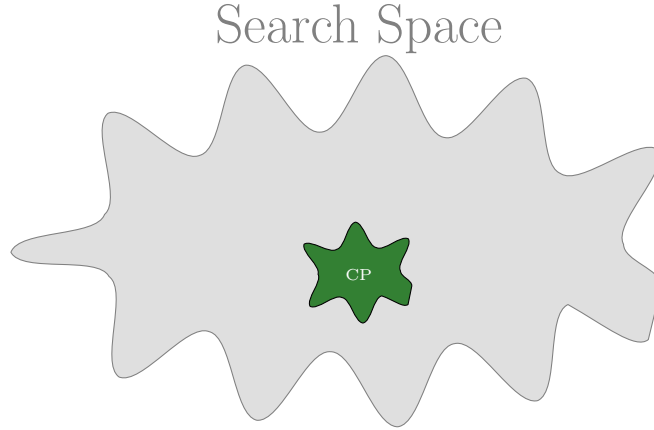


Figure 3.3: Explored search space with a pure CP search

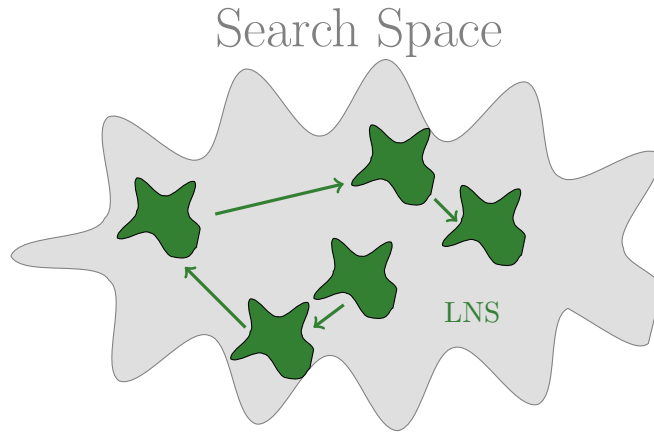


Figure 3.4: Explored search space with Local Neighbourhood Search

on the search limit will be detailed in further sections.

Expressing the optimization this way has some drawbacks. The solver will try to improve the score of the objective function. But since improving only objective2 already improves the score, the solver will optimize on objective2 at the expense of objective1. Objective1 will be improved only when objective2 will not be improved any more. Upon running the program, we observe that it reduces the number of students present in the minimal gap, until there aren't any students anymore in this gap. It then goes back to reducing the number of students present in the newly created minimal gap. It didn't optimize on one objective and then on the other, but it optimizes on both together. The fact that the optimization operates on both objectives together reduces the efficiency of the pruning. And finally, this way of expressing the optimization tends to make the introduction of objective4 very hard

while keeping an efficient solver (each additional term in the weighted sum weakens the pruning). But there exists alternatives to weighted sum. Pierre Schaus presents in his article **Variable Objective Large Neighborhood Search** [15] a possible way to get rid of most of the above-described issues.

3.3.4 Variable Objective Large Neighbourhood Search (VO LNS)

Variable Objective Large Neighbourhood Search (VO LNS) is an extension of Large Neighbourhood Search in which we optimize on some sub-objectives at a time, and we dynamically change the sub-objectives along restarts. Variable Objective LNS has two major benefits.

- It allows fully controlling the different prioritizations between all sub-objectives, and
- it offers much more aggressive filtering since it focuses on specific sub-objectives and not on (weighted) sums which – let’s recall it – are not efficient as objective functions.

In order to change a (weighted) sum into an VO-LNS formulation we will rephrase a minimization function into optimizing several objectives as described in Figure 3.5.

WS Formulation		VO-LNS Formulation	
Minimize	$z = o_1 + o_2 + \dots + o_n$	Optimize	$obj = (o_1, o_2, \dots, o_n)$
Subject to	$constraints$	Subject to	$constraints$

Figure 3.5: Transformation of the way to express optimization

In VO LNS, each of the variables $o_1, \dots, o_n$ is assigned to a specific filtering mode among **NoTighten**, **WeakTighten** and **StrongTighten**. We need to be careful on the way we set those modes since a misconfiguration could degrade the quality of solutions. The three mode are :

- **NoTighten** deactivates the objective. The objective will thus not impact the search.
- **WeakTighten** on an objective updates the bound of the objective such that all solutions found in the B&B DFS will have to be better or equal than the previous value of the objective.
- **StrongTighten** on an objective updates the bound of the objective such that all solutions found in the Branch & Bound (B&B) Depth First Search (DFS)

will have to be strictly better than the previous value of the objective.

Before using those modes, it is important to realize that they need to be correctly configured, otherwise, we could be subject to bad optimization (this is true also for simple problems). For this example, we use X_1 and X_2 as variables with domains $[0..2]$. And the optimization objective is $obj_3 = X_1 + X_2$ (that we want to minimize). We introduce two intermediate objectives $obj_1 = X_1$, $obj_2 = X_2$. We apply a simple backtrack search that first assigns lower ID variables, with higher value in their domain. The execution steps are given on Figure 3.6.

Experiment 1			Experiment 2		
obj_1	obj_2	obj_3	obj_1	obj_2	obj_3
W	W	S	S	W	S
2	2	4	2	2	4
2	1	3	1	2	3
2	0	2	0	2	2
1	0	1			
0	0	0			

Figure 3.6: Example of misconfiguration of objectives mode

We can observe that the best value for obj_3 is reached in the first experiment, but not in the second one. The reason is that, in experiment 2, obj_1 is set in the **StrongTighten** mode. This means that the objective must improve at each step. When it reaches 0, obj_1 can't no more improve, neither do obj_3 .

Now that we understand how tighten mode works, we can look how to transform our weighted sum into several sub-objectives and associated tighten modes. The weighted sum objective function can be reformulated as the minimization on objective1 and objective2 with objective1 being set in the **StrongTighten** mode, and objective2 being set in the **NoTighten** mode [let's recall that objective1 represents the GMG and objective2 represents the number of students having the GMG as personal minimal gap]. Once the best value for objective1 has been found (or at least, when we think we can't improve objective1's value anymore), we switch the objective1 mode to **WeakTighten** and objective2 to **StrongTighten**.

```
// Equivalent VO LNS
solver.minimize(objective1, objective2)
solver.obj(objective1).tightenMode = TightenType.StrongTighten
solver.obj(objective2).tightenMode = TightenType.NoTighten

...
```

```

/* Restarts */
while (condition) {
    startSubjectTo(failureLimit = 1800) {
        /* When we reach the best possible value for objective1, we switch on
           objective2 by switching objectives mode */
        if (bestObjective1 == maxObjectivePossible) {
            solver.obj(objective1).tightenMode = TightenType.WeakTighten
            solver.obj(objective2).tightenMode = TightenType.StrongTighten
        }
        /* Random Relaxation */
        for (i <- 0 until decisionVariables.size; if rand.nextInt(100) <
            environnement)
            add(decisionVariables(i) == cp.lastSol(decisionVariables(i)))
    }
}

```

3.3.5 Comparison between Weighted Sum and VO LNS

Figure 3.7 represents the optimisation scores obtained using Weighted Sum on one side, VO LNS on the other side.

We immediately observe that VO LNS is much faster than weighted sum at the beginning, and finds quite directly a solution with no students having two examinations on the same day. (*Optimization scores under 0 correspond to situations in which some students have multiple exams on the same day, and optimization scores over 0 correspond to situations where no students have two examinations on the same day. The ranges corresponding to objective1 are expressed in the Table below.*)

It is very clear that the technique of focusing on specific sub-objectives one at a time considerably improves the pruning, and allows us to find better solutions more quickly.

Objective1	Range for our data	Range Expression
0	[-464,-1]	$[-1 * \#students, (0 * \#students) - 1]$
1	[0, 463]	$[0 * \#students, (1 * \#students) - 1]$
2	[464, 927]	$[1 * \#students, (2 * \#students - 1)]$
...	...	...
n	...	$[(n - 1) * \#students, (n * \#students) - 1]$

We can also observe out of Figure 3.7 that the VO LNS search switches its objective tightening mode after approximately 3 seconds. The value of objective1 at this point is 1. Indeed, with data introduced in the solver, maximum value for objective1 is 1. Since we cannot improve objective1 anymore(*), the solver switches objective1

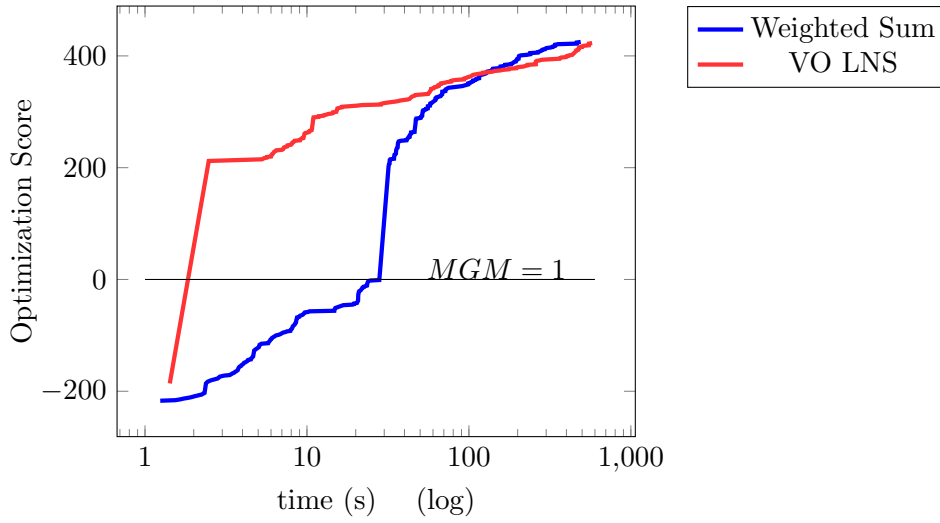


Figure 3.7: Comparison of Weighted Sum & VO LNS optimization

mode to **WeakTighten** and objective2 mode to **StrongTighten**. The minimization on the number of students having GMG as minimal gap can now begin.

(*)Using our data, objective1 can't go above value 1. Let's have a look at our set of data fed into the solver :

- The number of weeks of the session is set to three, beginning on a Monday, finishing on a Sunday, and no exam are given on Sunday's. Figure 3.8 shows us that with a GMG of 2 days, we can't set more than 9 exams (blue cells) in such a session.
- In our sample data, there are 2 students enrolled in 11 courses.

Both conditions imply that it will not be possible to obtain a GMG of 2, and that, in our case, the switch of mode must be done as soon as objective1 equals 1.

3.4 Avoiding Heavy Tail

Before being able to apply the LNS strategy, we need to find a first solution. While none are found, we can't apply the LNS principle since it needs to relax the previously found solution. We observed the time taken by our solver to find a first solution, doing the experiment 16 times. The table below (Figure 3.9) shows the number of backtracks and the time needed for each experiment to find a first solution.

We can see that on the simplified version of VO LNS (when only objectives 1&2 are declared), the time spent and the number of backtracks is quite constant (around one second, and 2.300 backtracks).

Mo.	Tu.	We.	Th.	Fr.	Sa.	Su.
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21

Figure 3.8: Calendar representing the maximum number of examinations under the constraint of having one free day ($GMG = 2$) between all the examinations.

#	VO LNS w/o Objective0		VO LNS w. Objective0	
	time (ms)	# of backtracks	time (ms)	# of backtracks
1	1.041	2.349	4.948	74.164
2	1.005	2.405	71.521	2.058.478
3	1.109	2.506	74.345	2.077.927
4	1.549	2.307	1.225	2.324
5	992	2.373	73.207	2.145.292
6	1.117	2.281	84.506	2.242.843
7	1.038	2.326	82.491	2.251.791
8	1.035	2.155	4.809	74.616
9	1.048	2.489	82.730	2.157.376
10	1.063	2.429	91.004	2.272.771
11	1.033	2.568	70.954	1.910.004
12	1.041	2.386	92.327	2.202.520
13	1.194	2.330	1.045	1.958
14	1.029	2.207	84.178	2.280.295
15	1.053	2.402	71.093	2.004.938
16	966	2.282	83.665	2.262.637

Figure 3.9: Time and number of backtracks performed to find a first solution.

On the other hand, once we enrich the optimization function with the **NoClash** objective (objective0) and the distribution objective (objective4), we get a totally new behaviour. Time spent to find a solution vary much more. This goes from one second and 2.000 backtracks up to 100 seconds and more than 2 million backtracks. This is an heavy tail behaviour. We are actually searching a solution in a search space with no such solution near the actual search space (this results from bad choices made in the first steps of the search). To avoid being stuck in those heavy tail, we will also apply a search limit while looking for a first solution. If we spend too much time without finding solutions, we restart, in order to escape the possible heavy tail we were in. As shown in Figure 3.9, we can find that the fastest two solutions have been found with

less than 2.400 backtracks. We will thus do some experiments on restarts with the number of backtracks near this value, see Figure 3.10.

backtrack limit	# restarts	time (ms)
1.250	5,6	2.549
1.500	5,2	2.578
1.750	5,0	2.428
2.000	4,2	2.487
2.250	4,8	2.366
2.500	4,6	2.376
2.750	6,0	2.873
3.000	4,6	3.960

Figure 3.10: Mean of the number of restarts and of the time to find first solution with different backtrack limits

We can observe out of Figure 3.10 that the number of restarts is quite the same whatever the backtrack limit is (It varies between 4 and 6). This is normal since, as observed on Figure 3.9, in most cases we fell in an heavy tail. The time spent to find solutions first decreases, and then increases. This can be explained as follows :

- With low backtrack limits, we could miss a solution that would have required only some additional backtracks. We will thus spend more time by restarting the search from scratch.
- With high backtracks limits, we will waste much more time when the search will remain stuck in an heavy tail.

We thus have to identify the lowest backtrack value that will find a solution in most non-heavy tailed configurations. This being said, we remember that the number of backtracks of the best two solutions found in Figure 3.9 was less than 2.400 backtracks. At the same Figure, on the left part, we also saw that all the backtrack numbers were close to 2.300 backtracks. And finally, best time with the restart strategy were achieved with a number of backtracks from 2.250 to 2.500. On this basis, we decided to set the backtrack limit to 2.500.

3.5 Relaxation

3.5.1 Random Relaxation

The first relaxation implemented is a pure random relaxation. At each restart, we assigned to 70% of the variables the value they had at last solution. Those 70% were chosen randomly. This relaxation worked very well on the weighted sum objective function. But when we switched to the VO LNS search, this weren't giving good

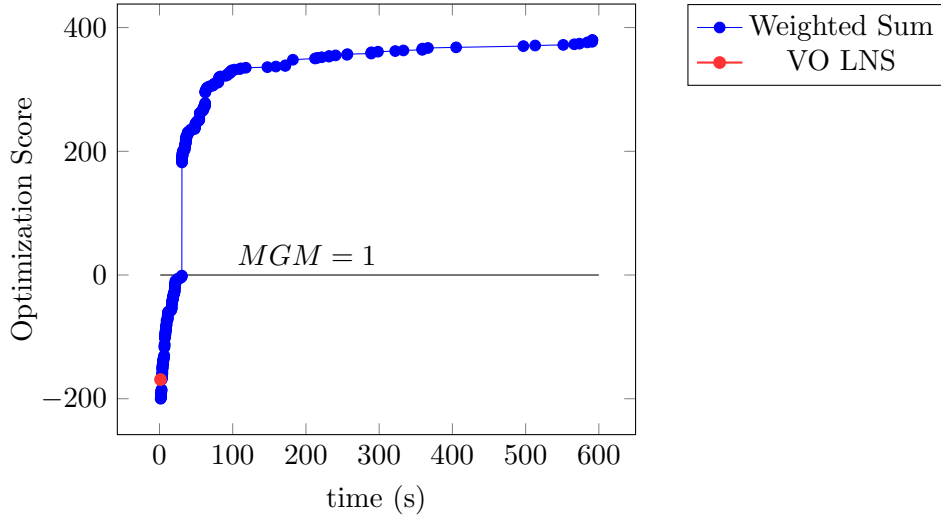


Figure 3.11: Evolution of objective score with a pure random relaxation on 70% of the variables

results at all. Figure 3.11 shows us the evolution of the objective for the two operating ways.

We see that for Variable Objective Large Neighbourhood Search, the search doesn't succeed in improving objective0, and remains stuck with the objective value at 0. Let's explain the reason why a pure random relaxation, with 70% of the variable fixed isn't working well in this case.

At the beginning of the search, we set objective1 in the **StrongTighten** mode, and all the other objectives are set in a **NoTighten** mode. The solver successfully finds a first solution, but doesn't succeed in finding a new solution that is strictly improving objective1 value (in our case, finding a solution where objective1 would equal 1). By looking at the statistics of the search, we immediately observe that

- All searches always completed (discovered the entire corresponding sub-tree).
- The number of backtracks is always 0.

The reasons why no more solutions are found becomes clear : By randomly assigning 70% of the previous solution, we also assign values that make the improvement of objective1 impossible. The solver immediately detects it, prunes the entire sub-tree, and finishes the search with no backtracks.

Two methods are possible to solve/avoid this problem.

- The first one is to relax variables that prevent the improvement of the objective function. Doing this at random is still important in case of restart without solution. We will thus relax randomly some other variables.

- The second method is to make the neighbourhood variable. If no solution has been found, we enlarge the neighbourhood (assigning less variables). If the entire sub-tree has been explored, we enlarge the neighbourhood much more. And finally, if a solution has been found, we decrease the neighbourhood (we could find other solutions near this one in the tree).

Those two ways to proceed are explained and tested in the following sections.

3.5.2 Worst Solution Relaxation

We saw in the previous section that randomly fixing 70% of the variables of the previous found solution wasn't a good idea because of the high probability to fix to variables preventing the objectives improvement.

- We thus decided to automatically relax variables that prevent those objectives improvement.
- We also decided to avoid fixing orals due to the strong correlation between all variables for a same oral examination.
- Among all remaining courses, we decided to randomly fix 70% of the corresponding variables. *Remember that a randomized part in the relaxation is essential in order to get different assignment at each restart.*

Figure 3.12 shows that VO LNS is really quicker than weighted sum objective at the start, but after long time, their results become quite similar.

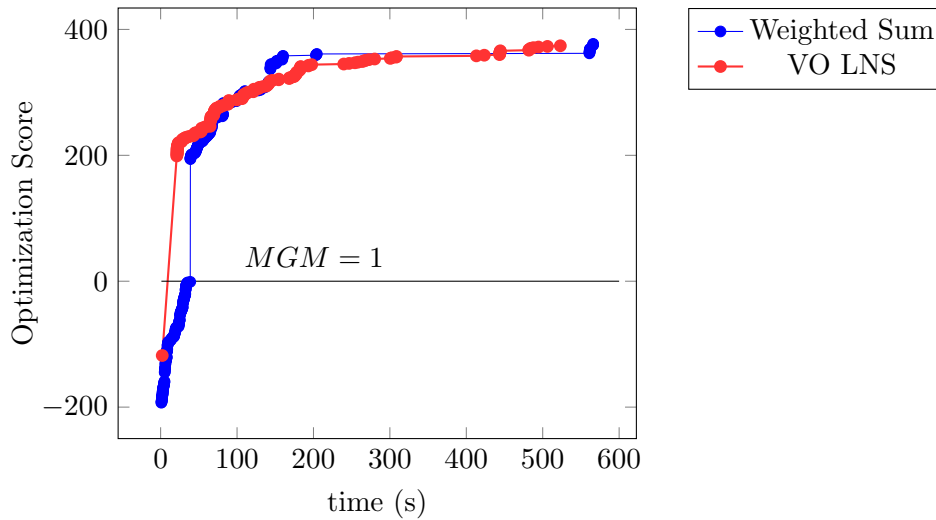


Figure 3.12: Evolution of objective score with a relaxation on the worst values in terms of objective1 and keeping fixed 70% of the remaining variables (selected at random).

3.5.3 Adaptive Neighbourhood Relaxation

Another method that can improve the search is to make the environment vary at all searches. If no solutions are found during a search, maybe that we should enlarge the neighbourhood (by assigning less variables). This allows us to get more diversity and gives us more chances to find a solution. If the search has been completed, we need to enlarge the neighbourhood much more. And finally, if a solution has been found, there could be a solution in the near neighbourhood. We thus get back to a small neighbourhood, that will allow us to find such solution quicker.

- The initial neighbourhood is set to 30% (this means that we assign 70% of the variables)
- We decrease it by 1% if no solutions have been found
- We decrease it by 10% if the sub-tree has been entirely explored
- We set it to the maximum between 70%, and the actual value incremented by 1% if a solution has been found.

The results of the searches is shown on Figure 3.13.

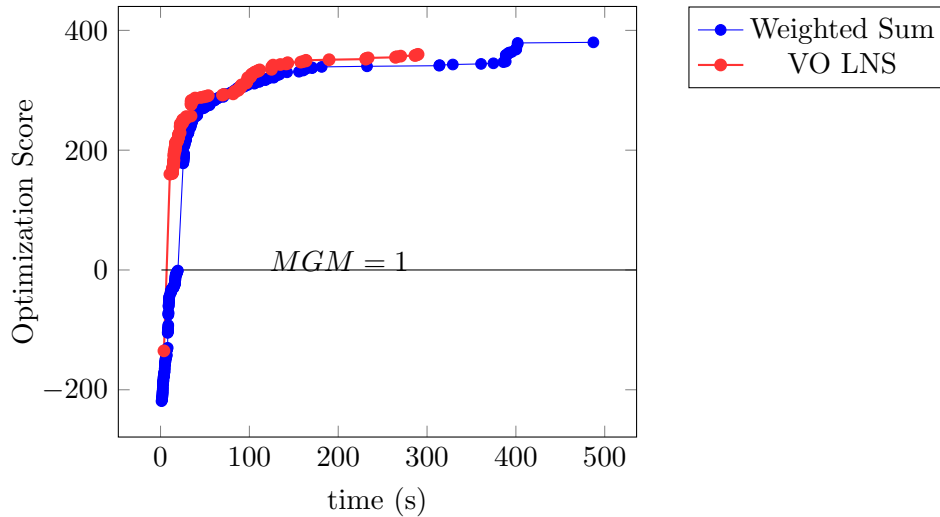


Figure 3.13: Evolution of objective score with a random relaxation on a variable neighbourhood.

We can see that, once again, VO LNS is better at the start; after that, its performance is quite similar to that of the weighted sum method.

3.5.4 Worst Solution & Adaptive Neighbourhood Relaxation

The adaptative neighbourhood and the worst solution relaxation both improved the best solution search. But using one doesn't mean that we can't use the other. On

the contrary, we could use both together. Figure 3.14 shows the evolution of the objective function using both methods together. This time, there is no doubt, the VO LNS search is much better. It finds better solutions sooner than the weighted sum objective function.

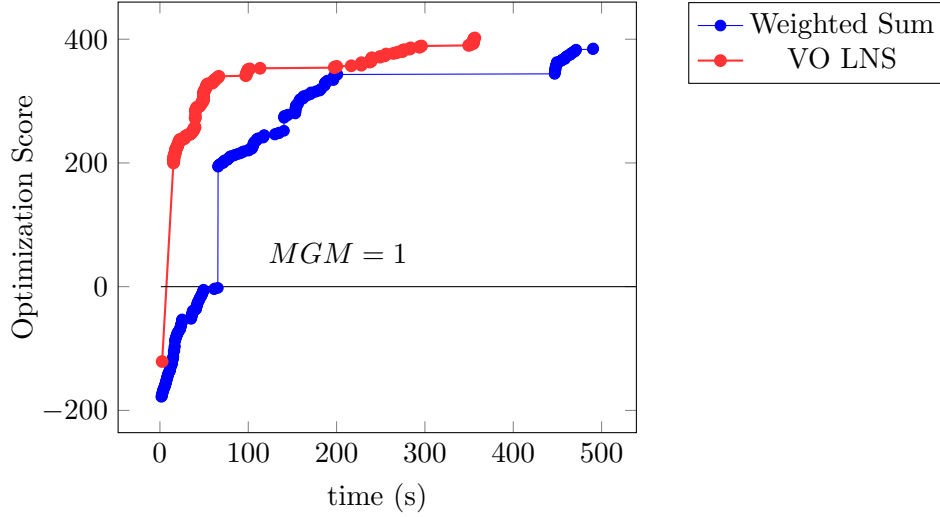


Figure 3.14: Evolution of objective score with a relaxation on worst variables and adaptive on the remaining neighbourhood.

We saw that in all the cases, the VO LNS search was much better during the optimization of objective1. Since our sample data has a best theoretical value for objective1 at 1, the solver using VO LNS is better until objective1 reaches this value. But we think that the difference of performances between VO LNS and weighted sum would be much more significant if applied on sample data with a higher theoretical value for objective1. Furthermore, VO LNS allows to better control objectives that we want to minimize, and permits introducing new ones easily.

3.6 Comparison between the previous three methods

In all the different methods studied previously, VO-LNS performs better or equally to weighted sum optimization. But we didn't try to find which method is better for VO-LNS itself. Results of previous methods are shown in a same graph at Figure 3.15.

We observe that :

- Methods using adaptative neighbourhood find a solution faster at the beginning,
- Methods using worst solution relaxation continue to find solutions when longer running times are allowed.
- Combining both methods gives the best results.

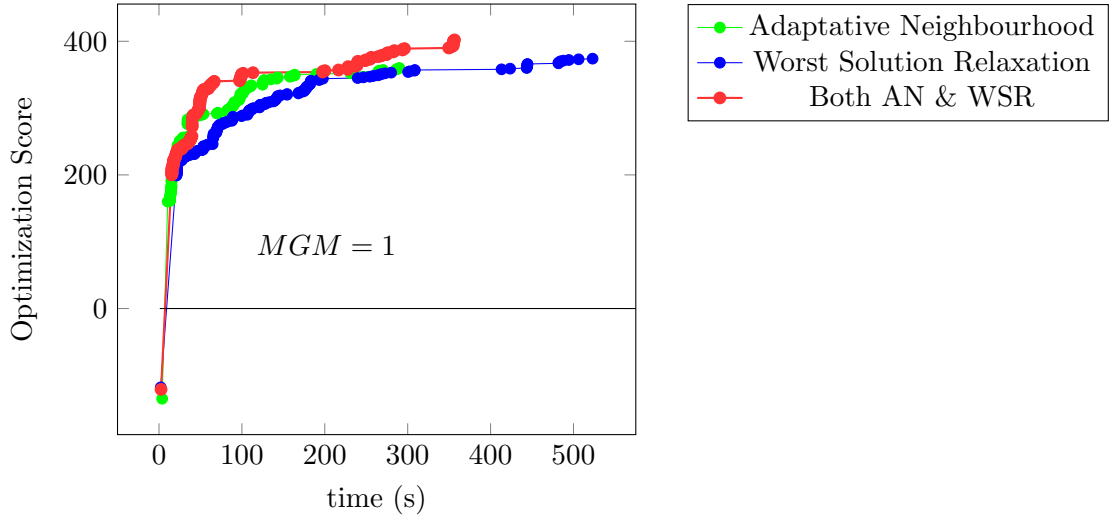


Figure 3.15: Evolution of objective score for the three different relaxation methods.

3.7 Search

We saw that with Local Neighbourhood Search (LNS), we jump in the search tree to explore different parts of it. Each time we jumped, we recreate a search tree corresponding to the exploration of the actual neighbourhood. The exploration of such search tree can also be optimized by choosing next variables and values to be labelled. Choosing the order of assignation in a set of variable is done using the *variable heuristic*, and choosing the order of assignation in a set of values is done thanks to the *value heuristic*. We will present here below some variable heuristics that could be interesting in order to optimize our solver.

3.7.1 Binary First Fail

The binary first fail heuristic is an heuristic widely studied and well known in the domain of constraint programming. Haralick and Elliott shows in [16][17] the efficiency of first fail principle while exploring a tree search in a Constraint Satisfaction Problem. The main idea is to choose first variables that are the most likely to yield failures. A quick detection of failures will minimize the search cost. And in the case the problem is over constrained, this will detect it much more quickly.

3.7.2 Conflict Ordering Search

Gay, Hartert, Lecoutre and Schaus presented a new guided search [18], which is also based on the principle of labelling first variables that are the most likely to fail. Conflict Ordering Search (COS) needs an initial variable heuristic, but remembers the variables

that were implied into conflicts at failures. Priorities are modified at each failure, and conflicting variables get an higher priority. COS will thus quickly get rid of the initial variable heuristic, and have its own based on conflicting variables. It has furthermore been shown that COS approach works better in a restart framework.

3.7.3 Comparison

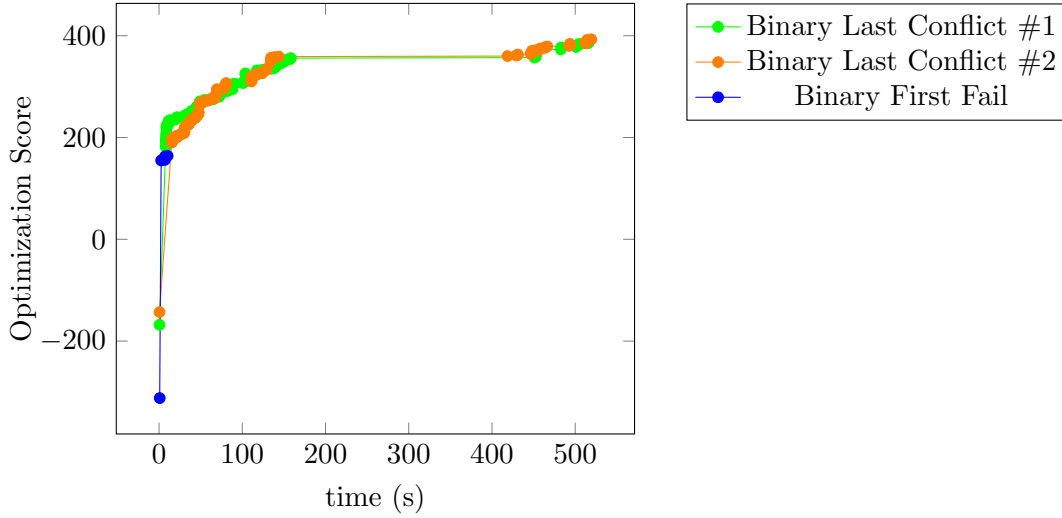


Figure 3.16: Comparison between variable ordering strategies.

Three different variable ordering methods have been tested. The first one is the traditional First Fail Principle. The second one (Binary Last Conflict #1) is the binary last conflict search using variable having minimal values in their domains first as initial heuristic. The last one (Binary Last Conflict #2) is also using COS, but the initial heuristic tends to label variables representing oral examination first. All results are shown on Figure 3.16.

We can observe that both searches using Binary Last Conflict achieve good results, whereas the famous First Fail search quickly stay stuck without succeeding in optimize further.

3.8 Generated Instances

In order to test our solver on other instances that only the one of the June session at EPL, we decided to create an instance generator. The generator will create instances with given number of courses, number of students, number of professors, the mean number of enrolment per student and the mean number of examination per professor.

Instance	Opti. Score	Max. Enrolment	Obj.1	Best Theoretical Obj.1
Generated 1	283	12	1	1
Generated 2	299	11	1	1
Generated 3	293	12	1	1
Generated 4	290	11	1	1
Real Data	374	11	1	1

Figure 3.17: Statistics on generated instances and real one

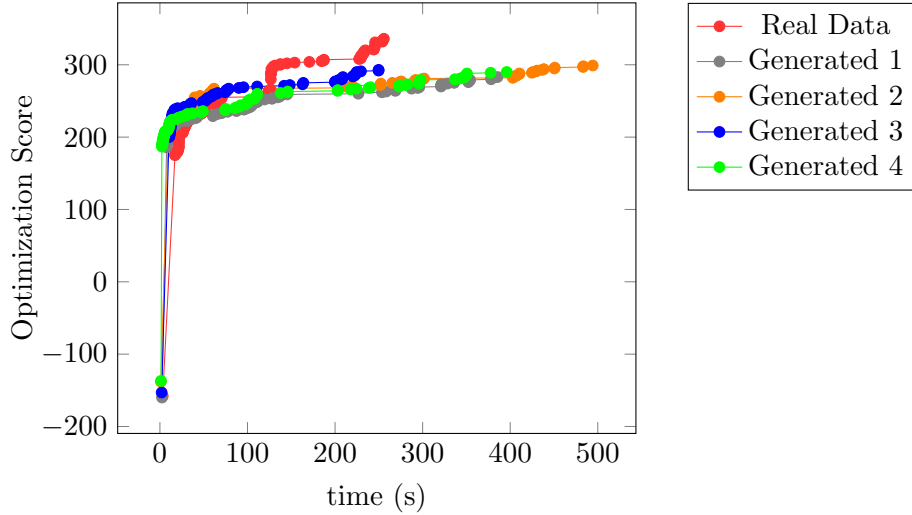


Figure 3.18: Running solver on generated instances

From Figure 3.17 and Figure 3.18, we can observe that solver works better on real instance data. Results on generated one are still good since the solver succeeds in avoiding clash for all students, and further, reach the best theoretical possible value for Objective1. But results are a little bit lower than those of the real instance. A possible explanation of that results is that a generated instance has difficult to correctly simulates a real instance. Indeed, in a real instance, a student from a specific department (e.g. computer engineering department) will have lots of course in common with other students of this department. But he will have no or few common courses with a student from another department (e.g. biomedical department). Our generator can hardly express this cluster behaviour in generated instances.

Server

Now that the solver part of our application has been extensively explained, let us move to the second part of our objectives: the usability of the application. This chapter divides itself into three main parts: first, we explain our architectural choices and motivate our choices of framework/technologies then we move on to the REST api developed to shield the solver, explain the mechanics of the web application. We will wrap up this chapter by summarizing the interactions between these two parts of the application. During the reading of this chapter, if you want to play with the current version of the program, you can access it at the url: <http://wime.cloudapp.net>. If need be, you can find a user guide in the appendixes of this master's thesis.

4.1 Architectural choices

First of all, it is necessary to understand the architectural choices that the computation power needed to solve the instance (to an interesting level of optimum) in a quick time is above the average of a desktop computer. With this fact in mind, two main architectural choices are possible: a (distributed) desktop application or a web application. It is interesting to notice that, in each case, a protocol for the communication between the solver and the application is essential. We will keep the choice of the protocol for the last of this section since this choice will be heavily influenced by the choice of the application.

Now let us weight up the pros and cons of each possibility. For the distributed desktop application, the main advantage is the knowledge already acquired in the languages and techniques used. As a matter of fact, having both followed either de "Distributed Applications Design" or its successor "Cloud Computing", we were already possessors of the know-how required for the implementation of such an application. Furthermore, this choice would allow to more easily control the access to the application (via the simple control of the binaries) where a web application would basically be more "open" to the world. On the opposite side, the choice of a desktop application would force us to handle problems such as firewalls, internet connectivity, computer compatibility, etc.

The second choice available to us was the web application. The main advantage of the web application choice is the ease of integration with the already existing services of the university. Furthermore, this choice would allow the secretary and the student

	Pros	Cons
(distributed) Desktop	Knowledge already in our possession Control over the binaries Design flexibility Speed of processing	Installation required on each device Update and Maintainability Too much freedom in UI Design Network specificity (VPNs, firewalls)
Web	Ease of integration Guidelines for UI dev. Centralized app deployment basic Http protocol Accessible anywhere	slow performances cross browser incompatibilities browser limitations internet required multi-language development

Figure 4.1: Table presenting the pros and cons of the two main architectural choices

representatives to try and run a few simulations on the application without having to endure the burden of an installation on the device they want to use. From a technical point of view, the knowledge of web applications had been already developed by one of us while the other one was eager to try and learn new ways of building applications. Nowadays a lot of mainstream industries are taking their software to the cloud, SaaS (Software as a Service) allowing for more interconnected solutions becomes (or has become) a main component of the IT industry. The principal disadvantage of this alternative is the lack of power the web applications have on the frontend. Indeed, one of the functions of the application will be to parse the file(s) representing the university scheduling problem. Such parsing can be rather CPU intensive when executed in the browser.

As you might have guessed, our choice went to the latter of the two alternatives: we chose to build a web application. (table 4.1) presents the main points we thought about when making this selection. With the broad variety of framework available to build web applications, we needed to trim down the tree of possibilities. With the help of the criteria formulated in table 4.2, we arrived to a threefold choice: AngularJS versus MeteorJS versus Play. Each one of these framework answered to enough of our criteria to be considered: Angular and Meteor are javascript framework working in the browser while the play framework is a more classical approach to website building but is compatible with AngularJS and is written in Scala. All these framework would allow us to build complete web applications and benefited from an active and growing community. And in the end, two main arguments were retained in the favour of Meteor: its real-time rendering and its modularity. Indeed making this choice forced us to completely separate the code bases of the solver and its shield from the web application, allowing a greater modularity and resilience in the application.

Having found our web application framework, we still needed to define the protocol that would be used to communicate between the two parts of the application.

The REST protocol imposed itself to us as the standard of the web protocols. The Spray framework was retained for the implementation of the api thanks to its low level methods leaving more room for the optimization and its actor based concurrency.

4.2 The REST Interface

With the architectural choices being made, let us delve deeper into the REST part of our application. We first explain the choice of the Spray framework, then move on to the encapsulation of the solver against concurrency mayhem and finish with the REST api in and of itself.

4.2.1 The choice of Spray.io

In Scala, two main libraries are available for RPC/REST implementation: the Spray (soon-to-be Akka-http) and Finagle frameworks. The latter is built by Twitter while Akka is maintained by TypeSafe. The main differences between them can be summarized in two fields: the concurrency and communications models. Finagle (built in Scala and Java) relies on Futures to handle the concurrency and supports a lot a different protocols and services, it seems more flexible for heterogeneous systems. Spray (also built in Scala and Java) proposes a concurrency model based on message passing and actors: each actor has its own light-weight thread shielded from the rest of the application. The communication model of Spray is message passing (between

<i>Categories</i>	<i>Criteria</i>
Security	possibility of encryption fast creation of user accounts smallest amount of data stored on the client
Usability	testing Single page application possibility to create webhooks real-time database updates important amount of usable packages
Performances	integration with Scala scalability
Miscellaneous	rendering performances learning curve Maintainability Modularity (possibility to replace it easily with something else) community

Figure 4.2: criteria for the choice of a web application framework

the internal actors) and the HTTP web protocol (with the rest of the world).

While both choices are valid ones, we have found that the Spray way of designing and maintaining the application allowed us for more modularity and resiliency. Indeed, each instance of the problem could run in its own actor, allowing us to recover gracefully from an unforeseeable error in the solving of the instance. Furthermore, it allowed us to strictly define an internal interface between the solver and the REST api parts of the code: the solver encapsulation.

4.2.2 The solver encapsulation

The need for an encapsulation of the model comes from two different parts: the protection of the data structures against concurrent accesses and requirement of consistency in the multiple maps and sets that compose the model. Furthermore, as mentioned here-above, it allows the overall application to be more resilient and to recover gracefully from an error or a too long search in the solver. First let us analyse the concurrent threat.

The concurrent threat

The Spray.io is a low level framework built for the efficiency of the application rather than for the ease of use. It relies on the concurrency model of the akka framework (framework that is also building block of Play) to handle the HTTP request. It furnishes a few different methods regrouped in packages: the http, can and routing packages are the ones we are interested in. These packages provide the necessary building blocks of the REST api we will define here after. What interests us now is the fact that, since these packages are built on actors, they inherently live in the concurrent world, allowing for efficient asynchronous handling of the requests. This asynchronous-ity is what causes such mayhem to the solver's model: imagine a request for the creation of a professor interleaved with the request linking said professor to a course. If the order of operations is not guaranteed, the result of these two actions become chancy at best. We need to define a bottleneck for each instance that will guarantee the order of the requests.

The consistency of the model

More than just the mayhem of the concurrency world, another danger for the solver is the inconsistency of the model. Indeed, the solver in and of itself does not provide for a clear cut CRUD interface: it is simply composed of the different data structures, parameters and variables used to solve the instance. This means that, as is, the solver can represent incoherent data according to the EPL's examination schedule (i.e. a course given by a professor whom does not exists in the professors set). We need to define an interface that will guarantee the validity of the data. The definition

also allows us to define a new constraint: the "professors unavailability" which maps directly to a set of courses unavailability constraints via the interface. The latter will be implemented using the actor based system of akka, dealing with both points of concerns presented.

Definition of the message passing protocol

The definition of an internal protocol being explained, we explicit now the different parts of this protocol. It is composed of two main files, namely the **Encapsulated Model Protocol** and its pendant, the **Encapsulated Model**; the former defines strictly the messages allowed to flow from an actor to the (encapsulated) model while the latter implements such messages. As explained in chapter 2, the problem's data is structured in different pieces: the general parameters, the students, courses and professors, the links between them and then the constraints. We follow here the same flow to present the different pieces.

The general parameters are the simplest ones: the only possible operations are the creation or the retrieval of the information. As you can see in the first listing, the messages are defined as case classes (for simple recursive decomposition in the pattern matching).

```
object EncapsulatedModelProtocol {  
  case class setNumberOfDays(*...*)  
  case class getNumberOfDays()  
  
  case class setNumberOfSessionPerDay(*...*)  
  case class getNumberOfSessionPerDay()  
  
  case class setFirstDay(*...*)  
  case class getFirstDay()  
  /*...*/  
}
```

The students are quite a bit more complicated. Indeed, two complications make their entrance with these resources. First, the update or removal of a student impacts more than just one data structure: imagine you update the student "a" name from $student_a$ to $student^a$, since the solver keeps the names of the students, courses and professors until the lowest level, you will need to update the map of enrolments as well as the set of students.

Moreover, the conventions of REST interfaces define the access to a specific resource via an integer identifier rather than via a string name. This forces us to

extend the current model by adding data structures linking a specific student/professor/course/constraint to an ID. In order to separate the business logic, this structure is kept in the akka actor itself rather than pushed onto the model. We define then the removal and update of the student accordingly and create a new message `getStudentOfId` exposing the newly created data structure to the end user.

```
object EncapsulatedModelProtocol {  
  /*...*/  
  case class addStudent(\*...\*)  
  case class removeStudent(\*...\*)  
  case class updateStudent(\*...\*)  
  case class getStudentOfId(\*...\*)  
  case class getStudents()  
  /*...*/  
}
```

The professors -linked messages have to handle one implication more than the students. Of course, the same problem intervenes: updating the name of a professor require modifications in the *teachesWritten* and/or *teachesOral* maps. But, since there exists a constraint for the professors unavailability, the update of a professor must not only keep the previously mentioned data structures coherent but must also handle the update of all the constraints linked to this professor. The signatures of the messages being very similar to the students ones, we refer you directly to the code.

The courses messages are very similar to the students and professors ones. Indeed, the management of the courses implies the same considerations that the one of the professors: at the update/removal of a course, the *enrolment*, *teachesWritten*, *teachesOral*, etc. must be update consequently. The constraints data structures must also be updated to reflect the modification. Even if more structures are impacted by the operations on a course, the same logic than for the professors applies.

The links between the professors/students and the courses have to be formally defined as well. Messages need to be defined to support the creation, retrieval and deletion of those links are defined such as:

```
object EncapsulatedModelProtocol {  
  /*...*/  
  case class addStudentFollowingCourse(\*...\*)  
  case class removeStudentFollowingCourse(\*...\*)  
  case class getStudentsFollowingCourse(\*...\*)  
  case class getCoursesFollowedByStudent(\*...\*)  
}
```

```

case class addProfessorGivingCourse(\*...\*)
case class removeProfessorGivingCourse(\*...\*)
case class getProfessorsTeachingCourse(\*...\*)
case class getCoursesTeachedByProfessor(\*...\*)
/*...*/
}

```

The update operation on these structure being handled by the ones on the professors/students/courses, the only operations needed are the ones defined here-above. These operations modify the *enrolment*, *teachesWritten*, *teachesOral* structures accordingly and are defined with respect to the identifiers of the resources rather than their names.

The constraints are handled via two simple messages: the `addESSConstraint` and the `removeESSConstraint`. While the first is sent with the details of the constraint to create, the latter is accessed only with the identifier of the constraint. An internal structure keeps track of the constraints and their IDs with the limitation of maximum one constraint of each type for each course (which represents the real-world use case). Each constraint is validated and then posted in the corresponding data structures.

```

object EncapsulatedModelProtocol {
  /*...*/
  case class addESSConstraint(\*...\*)
  case class removeESSConstraint(\*...\*)
  /*...*/
}

```

4.2.3 The REST protocol

The model being effectively shielded from the concurrent world and the end user, we define now the REST protocol. Before entering the details of the routing and marshallng of the resources, one point still remains unanswered: the multi-instances. Indeed, we frequently have spoken of multi-instances but have never explained how we handle and create them. This level of abstraction is built in the REST server. Indeed, as you will see hereafter, the protocol allows users to create new instances of the problem, each instance running inside its own encapsulation (actor).

The marshalling

The marshalling is the transformation of complex objects to a data format suitable for the transmission "Over the wire". We defined 7 different objects with the purpose of being exchanged between the REST server and its client(s).

```
object OTWModels {  
  case class Student(name: String)  
  case class Course(name: String)  
  case class Session(nDays: Int, nSessionPerDay: Int, firstDay: Int)  
  case class Constraint(Type: String, args: List[String])  
  case class Professor(name: String)  
  case class Callback(address: String, port: Int, relPath: String)  
  case class Answer(solution: Map[String, Array[Int]], stat: Seq[(Int, Int)])  
}
```

The first 5 ones should not come as a surprise, the only interrogation could be: "Why define single property object when a string is directly marshable?". The answer to this question lies in the evolvability of the model: adding a field to the students or the professors will be easier if the main resources have already been defined as objects.

The 2 last ones are specific to the REST interface and their usage will come to light over the next section.

The marshalling of the resources is done with the help of the Spray Json library and its default protocol: case classes being serializable, the last thing required to send such objects on the wire is the specification, to the Spray framework, of the format of each object. This is done via a simple implicit conversion:

```
object OTWModels {  
  implicit val StudentFormat = jsonFormat1(Student)  
  implicit val CallbackFormat = jsonFormat3(Callback)  
  implicit val ConstraintFormat = jsonFormat2(Constraint)  
  implicit val ProfessorFormat = jsonFormat1(Professor)  
  implicit val CourseFormat = jsonFormat1(Course)  
  implicit val SessionFormat = jsonFormat3(Session)  
  implicit val AnswerFormat = jsonFormat2(Answer)  
}
```

The routing

The routing of the requests consists of the handling of the different endpoints defined by the REST interface. We explain now each endpoint and the HTTP methods defined

over them. Each group of endpoint is divided into four different endpoints (aside from the constraint endpoints)

The `"/instance"` group :

- *the `"/instance"` endpoint itself.* It is the only endpoint which does not require the specific header `"X-instance-name"` to be set. Indeed, this header will serve as identifier of the instance for all other requests. The reason why such header is not required here is simple: the two methods defined over the endpoint are the ones creating (or removing) the instance. A request GET on this URI will return a unique random instance name to the user while a DELETE request will clean the data of the instance provided (in the body of the request) and destroy the corresponding actor.
- *the `"/instance/session"` endpoint* is defined for two methods: the POST and GET ones. POST'ing a request to this URI with a Session object as body will set up the *nDays*, *nSlotsPerDay* and *firstDay* parameters of the instance (pointed by `"X-instance-name"`). GET'ing will return the same object as provided before or the default one (which is a session of 24 days, starting a Monday and composed of 2 sessions per day).
- *the `"/instance/callback"` endpoint* allows for the creation and retrieval of the callback address. This address will be the one used to POST the solutions found (and their statistics). Two methods, GET and POST are defined here.
- *the `"/instance/search"` endpoint* is the URI used to launch the solving of the instance. The search only begins if (and only if) a callback has been already posted since it is the only method possible for the reception of the solutions. The URI defines an optional parameter: the maximum time of research for the solution, in milliseconds.

The `"/student"` group :

- *the `"/student"` endpoint* itself defines two HTTP methods: the POST and GET ones. POST'ing with a student as request body will create a new student for the specified instance (header `"X-instance-name"`) while GET'ing returns the set of students of the corresponding instance (*Students*). When POST'ing a student, the server answers with a header `"Location"` of the pattern `"/student/ID"` specifying the newly created student's ID.
- *the `"/student/ID"` endpoint* allows for the retrieval, update or removal of a specific student (of a specific instance). Respectively the methods defined are GET, PUT and DELETE. The second one demand for a student object as request body (and returns the identifier) while the first and last ones only refer to the student with its ID.

- the *"/student/ID/course"* endpoint defines only one method: GET. GET'ing on this URI will return the set of courses followed by the student of identifier ID.
- the *"/student/ID/course/ID"* endpoint defines only POST. POST'ing on this URI creates the entry (*student, course*) in the data structure enrolment via the encapsulated model.

The **"/professor" group** is completely symmetric to the students one (but uses professors instead of students and *teachesWritten, teachesOral* instead of enrolment). We refer you to the code for more information.

The **"/course" group** :

- the *"/course"* endpoint itself, like the *"/student"* endpoint, defines POST and GET. Likewise, GET'ing will return the set of courses of the specified instance while POST'ing will create the provided (as request body) course and return its newly created ID.
- the *"/course/ID"* endpoint allows for the retrieval (GET), update (PUT) and deletion (DELETE) of a course via its identifier. The request is validated and then forwarded to the encapsulated model. The PUT method require a course as a request body and returns the identifier.
- the *"/course/ID/student"* endpoint permits the retrieval of the set of students following the specified course via the HTTP method GET.
- the *"/course/ID/professor"* endpoint, symmetrical to the previous endpoint, allows for the retrieval of the set of professors teaching the specified course via the HTTP method GET.

The **"/constraint" group** :

- the *"/constraint"* endpoint itself defines only one method: POST. POST'ing to this URI with a constraint object as request body will forward the request to the encapsulated model and, upon completion, return the ID of the newly created constraint
- the *"/constraint/ID"* endpoint defines the DELETE method allowing for the removal of the constraint of identifier ID.

The role of each endpoint (aside from the instance creation/deletion), is to validate and (un)marshall the datas passing trough and forward the message to the corresponding encapsulated model actor (stored in a map data structure having for key the instance name and for value, the instance actor reference). Each method always returns with an http status code between 200 and 299 upon completion and an error code (> 300) with the details of the error as body if an error happened

A simple guide of the REST interface

The REST server can be launched via the file `Boot.scala` which can be found in the "launcher" package (*sbt run and then choose `Boot.scala`*). Note that, in order to work, all the subprojects of Oskar have to be published locally. A script name "updateOscar.sh" can be found at the root of the repository and handles such task. The script asks for the Oskar directory as argument.

The REST interface as defined allows for the usage of the solver by any application capable of sending and receiving simple HTTP methods. A typical usage of said interface follows the following scheme (with each request written as

$$METHOD :: URI [- header] [- requestbody] \rightarrow RESPONSE$$

- GET :: /instance $\rightarrow$ instance-name
- POST :: /instance/session - instance-name - {"nDays": 23, "nSessionPerDay": 2, "firstDay": 0}
- POST :: /course - instance-name - {"name": "LINGI2990"} $\rightarrow$ 0
- POST :: /student - instance-name - {"name": "Martin Crochelet"} $\rightarrow$ 0
- POST :: /student/0/course/0 - instance-name
- POST :: /professor - instance-name - {"name": "Pierre Schaus"} $\rightarrow$ 0
- POST :: /professor/0/course/0 - instance-name
- POST :: /constraint - instance-name - {"Type": "fixed", "args": ["LINGI2990", "0"]}
- POST :: /instance/callback - instance-name - {"address": "localhost", "port": 1234, "relPath": "example-instance-webhook" }
- POST :: /instance/search?maxTime=300000 - instance-name

The solutions and their respective statistics will then be POST'ed to "http://localhost:1234-/example/instance/webhook" each and every time a (temporary) solution is found by the solver and this during 5 minutes ($5 * 60 * 1000 = 300000ms$). Any of those request can be made with your favourite REST client, from *cURL* to the chrome extension "Advanced REST Client"¹

And example of utilization of the REST interface can be found in the file `testingREST.scala` of the package "launcher". If you want to retrieve the results, do not forget to launch the simple reception server `receiver.scala`.

¹see: https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddfdnphfgcellkdfbfjeloo?hl=en-US&utm_source=ARC

4.3 The Meteor Application

Having a REST interface on top of the server is helpful but does not meet the objectives we had given ourselves at the beginning of the master thesis. Our objective, concerning the application part was:

[...] to build an accessible application easing the usage of the solver by providing a clear and convenient way to deal with the exam session scheduling. This application should provide an efficient way to enter and modify the session specifications (compiled data) as well as administer means of browsing and measuring the solved timetable.

With this in mind, we present now the WIME web application. We begin by a rapid explanation of the key concepts of Meteor, then move on to the data structure of the application, followed by the security concerns. We conclude this section and chapter by a brief user guide to the application itself.

4.3.1 The Meteor way

Meteor is a "complete open source platform for building web and mobile apps in pure JavaScript" [8]. It is based on reactive and event driven programming models allowing for fast and real time rendering. It uses the Distributed Data Protocol (DDP) to communicate between the clients (frontends) and the server (backend) over websockets. This allows the Meteor applications to bypass standard VPNs and firewalls. Furthermore, Meteor implements Full Stack Database Drivers (currently on MongoDB (production-level) and Redis (prototype-quality)[8]) which allow for latency compensation and offline working.

Meteor also defines reactive programming on all levels: from the UI to the livequeries supported by the (mini)MongoDB, Meteor handles the database dependencies of what is called computations. This will be helpful when implementing the managers (see here-after).

Definition 4.1. (Computation). A computation in the Meteor framework is an "[...] object [which] represents code that is repeatedly rerun in response to reactive data changes." [19]

A Meteor frontend application is composed of three main parts: the routing (which is classical for any web application), the templates and the "managers". As explained in [20], the managers are not strict controllers in the MVC sense. We will then continue to use the term manager. The managers (which are basically customized computations) are linked to the templates via the blaze rendering system and their role is twofold: first, they define the data dependencies of the template and lastly, they format the said data in a way that is usable by Blaze.

A last important thing to know about Meteor is that basically all files are shared "as is" between the client (leading to an important overload at the starting of the application) and the server. There is nonetheless a few exceptions to this rule:

- the client folder (and all its sub-directories) are only interpreted by the client
- the server and private folders (and all its sub-directories) are only interpreted by the server (and thus not sent to the client)
- the public folder contains the static resources and is accessed in a classical way.

The routing

The routing in Meteor can be handled by a variety of packages (or even by hand). The most common router in Meteor is `iron:router` [21] which enables us with client side and server side routes. The client side routes are captured in the browser and simulated via the `html5 pushState` function while server side routes are classical ones requiring a server round trip.

The routing of the application is defined in two separated files: `lib/router.js` and `server/router.js`. The first one defines the client side routes and their dependencies toward (some parts of) the database (the router displays the loading template until all the dependencies have been cached in miniMongo) while the latter defines the webhooks that will be used when receiving solution answers from the REST api.

4.3.2 The data structure

Since Meteor does (currently) only work with the MongoDB database, the data structure is composed of collections, each collection itself composed of (schema-less) documents. This flexibility is harnessed by the application in the case of the students (see hereafter).

The WIME data model is structured in 6 collections:

```
/*collections/collections.js :: created both in the client and the server*/
Instances = new Meteor.Collection('instances');
Students = new Meteor.Collection('students');
Courses = new Meteor.Collection('courses');
Professors = new Meteor.Collection('professors');
Constraints = new Meteor.Collection('constraints');

BulkFiles = new Meteor.Collection('bulk-files');
Results = new Meteor.Collection('results');
```

The "Instances" collection stores the start and end dates of the session as well as the number of sessions per day. Each instance is created for a specific tuple (month, year) with the month being one of January, September or June. Lastly, the instance document stores the state of the instance which can be either "idle", "syncing" or

"solving". Only instances in the "idle" status will be allowed to create a REST instance and solve it. For security purposes, the "userID" of the creator is also added.

The "Students" collection is the more flexible one. Indeed, in addition of the required fields "name" and "surname", the students documents can, per instance, store variable fields (see the user guide for more informations on how to choose the fields to store). Each student is linked to an instance via its "_id" which is a field enforced by mongoDB and that is guaranteed unique. Each student is also linked to a user via its "userID" field. To represent the *enrolment* links between a student and the courses, the student disposes also of a "follows" field defined as a set of course identifiers: for each course followed by the student, the corresponding id is pushed on to the set.

The "Courses" collection stores the courses with two fields: its "tag" (i.e. LINGI2990) and its name (i.e. "Master Thesis"). The "userID" and "instanceID" fields are also automatically added.

The "Professors" collection keep the professors as documents composed of the "name" and "surname" fields as well as the "userID" and "instanceID" fields. Like the students, the professor documents now possess a field "teaches" (a set) containing all the course identifiers taught by the professor.

The "Constraints" collection harnesses also the flexibility of the schema-less documents by changing its structure accordingly with the constraint type (field "constraint-Type").

- Fixed and Unavailable constraints are composed of the courseID (field "_id" of the corresponding course document) and the slots field (array of Strings, each one representing a slot)
- Orals constraints are constructed with the identifier of the course, the number of slots and the number of students per slot.
- Multislots constraints compose themselves similarly to the Orals one without the number of students per slot field
- Professor Unavailable constraints are similar to Unavailable Constraints with the exception of the course identifier which is now the professor identifier.

The "BulkFiles" collection is a temporary collection only used to forward the documents from the frontend to the backend in a reactive way. Indeed, when the parsing of a file is asked by the end user, once the parsing itself is completed, the interpreted data structure of the file is directly logged into the BulkFiles collection (see subsection 4.3.4 for more details)

The "Results" collection is a collection representing the results propagated from the REST api to the (backend part) of the web application. Each document is composed of two main fields: (aside from the classical "userID" and "instanceID") the "results" and "stats". The former represents a map between the courses tags and their corresponding sessions while the latter represents the statistics displayed in the statistics page: for

each possible gap in the session, the number of students having said gap as minimal for their whole session.

4.3.3 Security

Concerning the security of the application, a few considerations have been taken into account. First, only the documents linked to a user are sent to said user. This is done via the publications mechanism of Meteor: the server only publishes to a client the documents that are allowed for said client; in our case, the server only publishes to the client the documents whose "userID" matches its own.

Then, the user is forced to log into the application to access its components. The log in is currently the vanilla version of the package user-accounts built by the Meteor team. The encryption of the traffic is possible with Meteor [22] and we would advise to use the nginx solution in order to gain the possibility of load balancing easily. However this not being our main focus, we leave this for further developments.

4.3.4 Interesting points of the Web Application

Aside from the data structure and the security, a few points of the web application deserve to be mentioned namely: the file import, the webhook mechanism and the REST api calls; we invite you to refer to the user guide for a more extensive presentation of all the mechanism of the web application.

The file import works as two step mechanism :

- first, the file is provided by the user from the instance page creation button "parse File". When a change in the pointed file is detected, the file is parsed with the help of the library "Papa Parse"[23]. This latter allows us to analyse the file in the browser very efficiently and, after validation by the user, the parsed data structure is written in a "BulkFiles" document. The use of "Papa Parse" library also allows us to easily handle different formats of csv files as long as the file begins with a header line. An example of such a file is

```
fac;course;profs;null;id;student;studOptions;inscribed>null
EPL;LFSA2995;Oestges;INFO22MS/G;3;Crochelet, Martin;optinfo2m/ch;I;
```

which will be interpreted as the bulk-file:

```
{
  userID: "7yvsFSzqnt7vbmG6s",
  instanceID: "8Mo8Di2Ncgy5HhLCa",
  results: {
    results: [{
      fac: "EPL",
```

```
        course: "LFSA2995",
        profs: "Oestges",
        id: "3",
        student: "Crochelet, Martin",
        studOptions: "optinfo2m/ch",
        inscribed: "I",
        null: ""
      }]
    meta: {
      fields: ["fac", "course", "profs", "id", "student",
        "studOptions", "inscribed", "null" ],
      /*...*/
    }
  }
}
```

- Then, the computation observing the BulkFiles collection is triggered at the addition of a new document and intercepts it. The computation reads the parsed data structure and, since each line of the file corresponds now to an object in the *results.results* field of the document, the computation parses the said array and create the documents needed to represent the resources and links written in the object.

Leaving the webhook mechanism for the end, let us look more deeply in the api calls made by the application (or, more precisely, how those calls are made). It seems trivial that the application will, for each resource composing the instance, send the corresponding REST call as defined in the previous section. What could be more trickier to understand is the fact that these calls have to be made from the server and not from the client. Indeed, since the REST api is built behind a proxy only available to the backend of the web application (in the current setting), only the server is able to reach the interface. The creation of the REST instance and the search starting is done from a Meteor method. Meteor method are basically Remote procedure calls made over the DDP protocol. They are registered using the `Meteor.method` call and called via the `Meteor.call` method. An example of such definition can be found in the file "server/serversideMethods.js"

The webhook mechanism is the system that allows us to receive the answers from the REST api as explained before. This apparatus is build with the help of the `http-methods` package and allows us to define a endpoint `"/api/solution/:instanceID"`. This endpoint is the callback registered via the POST callback of the REST api; using the MongoDB's `"_id"` field as identifiers of the resource. Each time a solution is POST'ed to this URI, we save the solution in the "Results" collection and building on the reactivity of Meteor, this modification is detected by the computations in the frontend

and automatically displayed on the screen. This is this mechanism that allows us to reactively update the schedule (results page) and the statistics (statistics page).

Possible improvements

Now that all the components of the application have been presented, we suggest possible ways of improving and extending it. To continue the same structure, we divide this chapter in three parts: the solver, REST and Meteor parts.

5.1 Improvements for the solver

The current state of the solver allows for the resolution of the basic problem of the university however a few cases still wait to be considered:

- The consideration of the rooms allocation: in the current state of the examination scheduling procedure, the University first creates its schedule and then asks for the rooms.
- The extension of the model to the whole EPL's case (bachelors and masters) and then to the university case: the current model has been built to handle the EPLs masters examination scheduling case and is not certified to work with an instance of the size of the university ($\simeq 30000$ students, $14\text{facs} \sim > 3000\text{courses}$) nor with its specificities.
- The reduction of the model to its smallest version: currently, if two students have exactly the same session, both students will post the same constraints. A type of "student group" gathering all the students having the same session (typically students of the same option in the early years of university)
- The current solver optimizes the global minimal gap for students then, the number of students having this gap as their own minimal gap. We could go one step further by optimizing other gaps of students, depict the fact that the minimal gap cannot be improved anymore.

5.2 Improvements for the REST api

The REST api has been built with the idea of a web application and is therefore not complete. From the specifications of a RESTful api, the still missing components are:

- cacheability : no cache has been considered while building the interface: all caching is currently disabled
- uniform interface: currently, each resource behaves differently than the other one.
- HTTP methods are still missing for some resources (constraints, basic parameters, etc.).
- No messages are currently self descriptive
- the handling of the request could be separated in one actor per endpoint group allowing for a greater load on the server.
- the implementation of an authentication mechanism (the REST server is currently hidden behind a firewall but an authentication mechanism would allow the opening of the REST port and third party applications to connect directly to it.

5.3 Improvements for the Meteor application

The Meteor application has been built in order to sustain the requirements we setted ourselves in the beginning of the master thesis however the application in and of itself is not yet production ready. Here is a list of the possible improvements:

- The removal of the "insecure" package: currently, the application allows all writes (not read) by default by the end user. Meaning that if, by some way, a malicious user could gain knowledge of an instanceID not belonging to him, he could remove/update the instance even if he is not the current owner of the document (or its linked documents (students, courses, etc.)).
- the correction of a few bugs in the UI (modal misplacement, buttons misplacements, etc.).
- the installation and setting of a SSL encryption (mentioned in the security section)
- the correction of a timeout error on Meteor server restart (not critical).

Conclusion

Now that we have developed something a little bit more efficient than the modern age version of a "slip of paper", it is time for us to wrap up our work and summarize our journey. To put in perspective the results of our master's thesis, let us quote back the objectives we had set ourselves:

[...] on one hand, we want to implement a solver using the OscaR library that will be able to work out a valid and practical timetable for the EPL. The solver will be intended to replace the work of the student representatives by providing its solution from the compiled data (students enrolments, professors availability, courses specifications, etc.). On the other hand, we want to build an accessible application easing the usage of the solver by providing a clear and convenient way to deal with the exam session scheduling. This application should provide an efficient way to enter and modify the session specifications (compiled data) as well as administer means of browsing and measuring the solved timetable

This allows us to look back to the work carried out. For the solver part, after having implemented a few different search strategies and chosen carefully the (sub) objectives to optimize, we have succeeded in finding a timetable that seems practical in a convenient time. With less than a second before finding the first solution and the ability to retrieve 130 different solutions in less than 5 minutes, we think that the objective of working out a practical timetable found in an acceptable time has been met. However, as mentioned in the previous chapter, the current state of the solver only allows for the resolutions of the EPL's masters examination schedule which is a reduction of the initial objective.

The development of such a component has given us the opportunity to implement a real world problem in AI and showed us that the optimization of a problem can make the difference between solving it and waiting for an answer long enough to compute the meaning of life, the universe, and everything. More concretely, we learned to model a well known mathematical problem, how to find and analyse different objective functions, how to optimize efficiently the search and the ins and outs of different search strategies.

For the "application" part, we have shown that the current application allows for real time feedback of the search as well as instantaneous visualization of the (temporary and final) solutions. The application, in its current state, can be played with at the

url `http://wime.cloudapp.net` and accounts for the proper functioning of the data structure and communication protocols we established. A lot of work remains still to bring the application to a production level: from the completion of the REST requirements to the correction of the few bugs that cripple the application including the insecure aspects of it (SSL, insecure package, etc.)

The development of the application instructed us in the mayhem of concurrency and the benefits of modularity and clear cut components; it showed us the lay of the land of http requests and status codes, gave us a taste of the marshalling of the objects while also allowing us to try our hand to interface design and specifications, event-driven programming and client-server web specifications.

This concludes our participation in this application and we truly hope that it will be able to help the secretary in its work and the student representatives in their negotiations but more than anything we hope that WIME will be able to help the students themselves by providing a quality solution for one of their worst nightmare in this end of May, a hard examination session.

Bibliography

- [1] schedule. *New Oxford American Dictionary*. Oxford University Press, 2015.
- [2] Altintas Cevriye, Asta Shahriar, Özcan Ender, and Yigit Tuncay. A self-generating memetic algorithm for examination timetabling. In *PATAT*, 2014.
- [3] Michael Eley. Ant algorithms for the exam timetabling problem. In EdmundK. Burke and Hana Rudova, editors, *Practice and Theory of Automated Timetabling VI*, volume 3867 of *Lecture Notes in Computer Science*, pages 364–382. Springer Berlin Heidelberg, 2007.
- [4] Barry McCollum. A perspective on bridging the gap between theory and practice in university timetabling. *Lecture Notes in Computer Science*, 3867(3-23), 2007.
- [5] Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.
- [6] Oscala Team. Oscala: Scala in OR, 2012. Available from <https://bitbucket.org/oscarlib/oscar>.
- [7] Spray.io. URL <http://spray.io>.
- [8] Meteor. URL <https://www.meteor.com>.
- [9] Yves Deville and JB Mairiy. Lingi2365: Constraint programming. course given at the Ecole Polytechnique de Louvain.
- [10] Christian Bessière and Pascal Van Hentenryck. To be or not to be ... a global constraint. In *Principles and Practice of Constraint Programming*, volume 2833, pages 789–794. 2003.
- [11] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. In 419-424, editor, *IJCAI'09: 21st International Joint Conference on Artificial Intelligence*, 2009.
- [12] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94 Proceedings of the twelfth national conference on Artificial intelligence*, volume 1, pages 362–367, 2001.
- [13] Willem-Jan van Hoeve. The alldifferent constraint: A survey. In *6th Annual workshop of the ERCIM Working Group on Constraints*, 2001.

- [14] Emmanuel Hebrard, Daniel Marx, Barry O’Sullivan, and Igor Razgon. Constraints of difference and equality: A complete taxonomic characterisation. In IanP. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 424–438. Springer Berlin Heidelberg, 2009.
- [15] Pierre Schaus. Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 971–978. IEEE, 2013.
- [16] Barbara M Smith and Stuart A Grant. Trying harder to fail first. 1997.
- [17] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.
- [18] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search. 2015.
- [19] Meteor docs. URL http://docs.meteor.com/#/full/tracker_computation.
- [20] Tom Coleman and Sacha Greif. *Discover Meteor, Building Real-time JavaScript Web Apps*.
- [21] Iron router. URL <https://atmospherejs.com/iron/router>.
- [22] Securing meteor. URL <http://www.meteorpedia.com/read/SSL>.
- [23] Papa parse library. URL <http://papaparse.com/>.
- [24] Liam TG Merlot, Natashia Boland, Barry D Hughes, and Peter J Stuckey. A hybrid algorithm for the examination timetabling problem. In *Practice and theory of automated timetabling IV*, pages 207–231. Springer, 2003.
- [25] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.