UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Course Timetabling

Building an application that optimizes the university's bachelor class schedule

Dissertation presented by
Hoa PHAM QUYNH

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Pierre SCHAUS

Reader(s)
Pierre SCHAUS, Francois GLINEUR , Yves DEVILLE

Academic year 2015 -2016

Abstract

For any academic programme, achiving an optimized course schedule has always been a challenging task. Such a challenge is known as the course timetabling problem, which is one the most widely studied optimization problems and is classified into the NP-Complete category. A course timetable normally involves lots of hard constraints and events, which makes solving the timetabling problem highly complicated and time-consuming, especially with manual scheduling. As such, various algorithms have been designed and developed into software tools in order to assist administrative staffs in approaching a high-quality timetable. For years, the Louvain School of Engineering (EPL) has been using an automatic scheduling tool based on the MIP model developed by Professor Francois Glineur. Among many approaches suggested by researchers, Constraint Programming (CP) has been considered highly effective in tackling the mentioned problems while providing a high level of declarative language that allows users to model problems and describe the properties of solution. Clearly, the expressiveness of CP reduces the amount of work required by the modeler. This thesis work aims at providing a CP-based implementation of an automatic scheduling tool that solves the course timetabling problem. Additionally, the tool is designed in such a way that it can be easily modified when the requirements of the problems change, making it an efficient and flexible solution.

Acknowledgements

I would like to express my gratitude to my promoter, Professor Pierre Schaus for his guidance and support all along this academic year.

Furthermore, I would like to thank to Professor Francois Glineur for his support. Without his help, I possibly met many difficulties when identifying the requirements of the course timetabling problem of EPL.

We would also like to give a special thank to my friends and my family who encouraged and supported me during two years of study.

Contents

List of Figures

Introduction

A timetable is a kind of schedule that sets out times at which specific events are intended to occur [1]. An event is a combination of resources (e.g people, rooms and items of equipment), which are specified by problems, and some of which must be allocated as a part of solution. Timetabling problems have a wide range of applications in education, hospital, and logistics. Every two years, the International Series of Conferences on the Practice and Theory of Automated Timetabling (PATAT) is held as a forum for both researchers and practitioners of timetabling to exchange ideas [19]. Clearly, timetabling is an active research topic which attracts many scientists.

The course timetabling is one of the most widely studied problems which is classified into the NP-Complete category and it cannot be solved in polynomial time. Solving the course timetabling problem manually consumes a lot of time especially when the number of students increases. Therefore, the university administrators usually require development of computer tools which schedule courses automatically. Each year, EPL needs to build the course timetable used in a semester. The current automatic scheduling tool used in EPL is based on the MIP model which has been implemented by Professor Francois Glineur [15].

Problem

Course timetabling problems occur in an educational context whereby a set of university courses have to be scheduled weekly in timeslots [2]. The solution to the problem needs to avoid conflicts in the student schedule and satisfies a number of side-constraints. The side-constraints can be the requirements of resource assignment, room capacities and so on. These requirements vary between different universities.

In the course timetabling problem of EPL, there is no requirement of resource assignment or room capacities. However, finding a feasible solution becomes difficult because of the course structure. Each course includes theory lectures and an exercise part. While fully attending lecture sessions is mandatory for all enrolled students, they just need to take one out of multiple exercise sessions of the same courses. The enrolled students need to be divided equally into different exercise sessions of the same course. This requirement causes a number of issues. In students' schedule, there are many courses which need the separation of students in exercise parts, and sessions of different subjects possibly overlap each other. An interesting question is how to split students into exercise sessions to avoid the conflicts in the student schedule as well as ensure the balance of the number of students in the exercise part. Besides satisfying the side-constraints, the solution to course timetabling problem also has to optimize several objectives which capture the requirements of all stakeholders of the problem. It needs to minimize the total conflicts in student schedule as well as maximize the preference of professors and the quality of schedule.

Motivation

The course timetabling problem is an active research topic. There are numerous papers studying about this topic in the PATAT conference series [14, 6, 16]. These papers present various algorithms which are implemented with different techniques such as MIP, local search, constraint programming. The effectiveness of algorithms are evaluated through the experiments with a standard set of benchmark data. However, there is still a gap between research and practice because the dataset can be simplified to reduce the complexities of real world course scheduling [3]. Furthermore, different institutions have a range of constraints so it is difficult to capture all these requirements when modeling this problem. Solving the course timetabling problem of EPL attracts me because I can gain the scientific knowledge as well as understand the complexity when modeling a real world problem.

Constraint Programming (CP) is one efficient technique used to solve combinatorial problems such as planning and scheduling. Its idea is to use constraints to reduce the search space by removing inconsistent values, which do not belong to any solution.

Approach

The course timetabling problem of EPL is modeled by using CP technique on top of *OscaR* solver [20]. Constraint Programming provides a high level of declarative language and allows users to model problems and describe the properties of solution. The expressiveness of CP reduces the amount of work required by the modeler. The choice of framework implementation does not restrict users to only CP technique. The libraries in the *OscaR* framework allows the modeler to experiment with other optimization techniques such as Large Neighborhood Search (LNS) and Variable Objective Large Neighborhood Search (VO-LNS).

Contribution

This master thesis presents a tool to solve course timetabling problem of EPL automatically. The application has succeeded in finding a solution which optimizes all the required objectives in a convenient time. Several optimization techniques has been experimented in combination with CP. This master thesis presents an additional approach to solve the course timetabling problem of EPL besides the current MIP model. All code implemented of this application can be accessed at the *bitbucket* repositories[1].

The structure of the thesis

The remainder of this master's thesis is structured as follows: the first chapter outlines some features of course timetabling problem. After that, the basic concepts and definitions of CP are presented. The formal model of the course timetabling problem of EPL is described in the third chapter. The next chapter presents the implementation of the model in *OscaR* solver and the differences between two approaches, CP and MIP. The fifth chapter synthesizes the results of experiments on the real data of EPL and compares with the results of the MIP model. A conclusion outlines the main contributions of this master thesis.

[1] *https* : *//bitbucket.org/hoapq*123*/scheduling_timetabling/src*

Chapter 1

Course Timetable Problem

This chapter outlines some features of course timetabling problem and list several approaches used to solve this problem

1.1 University timetabling

A timetable is a kind of schedule that sets out times at which specific events are intended to occur [1]. An event is a combination of resources (e.g people, rooms and items of equipment), which are specified by problems, and some of which must be allocated as a part of solution. Timetabling problems have a wide range of applications in education, hospital, and logistics. The university timetabling is one of the most widely studied problems.In *A Survey of Automated Timetabling*, Andrea Schaerf classifies it into two categories [2]:

- *Course timetabling*: The weekly scheduling for all the lectures of a set of university courses, minimizing the overlaps between lectures of courses having common students
- *Examination timetabling*: The scheduling for the exams of a set of university courses, avoiding the overlap between exams of courses having common students, and spreading the exams for the students as much as possible.

When modeling the course timetabling problem, it can be considered as a variance of the examination timetabling problem. However, two problems can be distinguished by the two following features. First, there are many stakeholders involved in the process of building university course timetabling. Therefore, much more work is required to understand the issues and it is important to consider the requirements and objectives of all stakeholders. In contrast, in the examination timetabling problem, the maximum of comfort of students, including no conflict in their exam schedules and the spread of the exam, is usually defined as the only goal. Another difference is that events in the exam are mostly organized at the module or course level. For example, each subject usually has one exam. However, in the course timetabling problem, subjects need to be broken into smaller parts such as lectures, seminars, exercise sessions and laboratory classes. Different types of events have specific requirements and principles. The following section presents some of the associated issues of course timetabling.

1.2 Introduction Course Timetabling

As mentioned above, there are many stakeholders involved in course scheduling such as administrators, instructors and students. These people have different requirements and expected goals. This part outlines the main features when identifying requirements and defining objectives of a real world problem. These attributes can help to distinguish course scheduling from exam timetabling problem.

1.2.1 Common requirements

There is a diverse variety of university timetabling problems. Each educational institution has a different scale as well as specific requirements. As stated in Edmund Burke's summary [5], the requirements of course timetabling can be classified into four categories:

- *Resource Assignment*: Each event needs to be assigned a set of resources. For example, exercise parts need to take place in a computer room. The specific demands of courses are defined by department or the responsible lecturers.
- *Time Assignment*: An event needs to be scheduled at a particular time. It is used to preassign a specific time to an event. In contrast, there are some restrictions of time assignment, e.g. a teaching unit should not be timetabled on given time slots because of the teacher's unavailability.
- *Time Constraints between Meetings*: There are precedence requirements between some courses or their subparts. For example, a lecture needs to take place before its corresponding exercise section.
- *Room Capacities*: The number of students in a room can not exceed the room's capacity.

The course timetabling problem can be the combination of resource assignment, time assignment and room usage problems, so finding its solutions becomes more complex in comparison with subproblems.

1.2.2 Different goals of stakeholders

Another factor contributing to the difficulty of course timetabling problem starts from the fact that there are many people affected by its result. There are three main stakeholders involved in the schedule and they have different measures to evaluate the quality of timetable:

- *Administrators*: The administration sets the minimum standards that the timetable must conform to [5]. For example, they require that timetable has to minimize the conflict in the schedule of students.
- *Instructors*: They want the schedule to maximize their preferences for courses. For instance, some teachers prefer to compress their teaching into one or two days while others can specify their favorite time slots.
- *Students*: From the students' perspectives, the timetable will be only restricted to the enrolled courses. Many students prefer not to have lectures in the end slots of day.

To model a timetable, a scheduler has to consider all involved parties and define the priority of different goals.

1.2.3 Course structure

Besides the difference of requirements and stakeholders' goals, course timetabling problems become more difficult because of different ways to model the course structure. A module can have different subparts such as lectures, seminars and exercise sessions. It is possibly required to divide enrolled students into subparts of course such as exercise sessions or laboratory parts. In practice, a number of issues are explored. In students' schedule, there are many courses which need the separation of students in exercise parts, and sessions of different subjects possibly overlap each other. An interesting question is how to split students into exercise sessions to avoid conflicts in student schedule and what are the best combination of students to be timetabled in which slots.

In some cases, it requires the number of students in each session to be strictly equal, while sometimes this constraint is too hard and needs to be relaxed. In the second situation, an objective is introduced to improve the balance between exercise parts. The splitting students into exercise sessions can be more complex in combination with resource assignment or room usage. Some approaches cluster the students before scheduling and try to avoid possible conflicts in timetables. For example, Michael W. Carter presents the way to group students into multi-session exercise in [6].

1.2.4 State of art

With the increasing number of students, educational institutions need the tools to automate timetabling. There are various methods to solve course timetabling:

- *Graph coloring*: In the early research, course timetabling is reduced to graph coloring problem. In graph theory, events (e.g lectures) are represented by nodes, between two events, which have same the teacher or enrolled students are linked by an edge. Time slots are considered as different colors. The goal is to color the graph in which no adjacent nodes have the same colors [7].
- *IP/LP method (Integer programming/Linear programming)*: In linear-programming models, variables are continuous and the requirements are represented by linear relationships. In contrast, IP has all or some variables with non-negative integer domains. It is referred to as called mixed integer program (MIP), when only some variables are integer. IL/LP is a complete method which is a branch of Operational research methods. The size of problem has a great influence on the execution time of MIP model. For large problems, it is sometimes difficult to obtain the optimal solution. Professor Francois Glineur uses MIP model to solve the course timetabling model of EPL and its formulation is presented in the appendix.
- *Constraint programming*: CP is used to solve the course timetabling problem in EPL and the definition is described in chapter 2.
- *Other approaches*: Besides the mentioned operational research methods above, other approaches such as local search, simulated annealing, and genetic algorithms have also been successfully used to tackle with the timetabling problem.

In this chapter, we have seen the difference between examination timetabling problem and course scheduling as well as the typical features when modeling course timetabling problem. This information will help to build the constraints and objectives of course timetabling problem of EPL.

Chapter 2

Constraint Programming

Constraint Programming (CP) is one efficient technique used to solve combinatorial problems such as planning and scheduling. This chapter presents the basic concepts and definitions of CP.

2.1 Constraint Programming

Constraint Programming (CP) is a programming paradigm developed from Constraint Logic Programming in which problems are described with constraints. The key idea of CP is that it provides a high level of declarative language that allows users to model problems and describe the properties of solution. The solver is responsible to execute specific steps of embedding algorithms and find the solution.

2.1.1 Constraint Definition

A *constraint* is simply a logical relation among several *variables*. The constraint restricts the possible values that variables can take in their domains [8]. The formal definition of constraint is given as the follows :

Definition 2.1. [1] A *constraint* c$(x_1, ..., x_r)$ is a logical relation among the *variables* $x_1, ..., x_r$. The constraint has a *scope* scp(c) $= (x_1, ..., x_r)$ which is a set of variables impacted by c with r called the *arity* of the constrain. Each variable x_i has its own domain D_i. The *relation* rel(c) represents value combinations accepted by constrain c in the search space $D_1 \times .. \times D_n$.

Example 2.1. Given a problem containing two variables x,y with domains D(x) = {1,2,3,4,5}, D(y) = {3,4,5,6}. Constraint $c \equiv x + y \leq 5$ restricts the domains of x and y to D(x) = {1,2} and D(y) = {3,4}

2.1.2 Constraint Satisfaction Problem

The basic idea of Constraint Programming is to use constraints to model and solve problems. These problems are called Constraint Satisfaction Problem (CSP).

[1] The definition is inspired from the note of Constrain Programming course [11]

Definition 2.2. Constraint Satisfaction Problem (CSP) is a mathematical problem represented by a triple *(X,D,C)* [2]:

- X is a set of n variables: X = $\{x_1, ..., x_n\}$
- D is a set of domains corresponding to each variable: D = $\{D_1, ..., D_n\}$
- C is a set of constraints defined over the variables $x_i \in X$: C = $\{C_1, ..., C_e\}$

A solution to a CSP is an assignment of variables to values in their domains such that all the constraints are respected.

2.1.3 Progagation

The goal of propagation is to reduce the search space by removing inconsistent values, which do not belong to any solution. After the propagation process, we find an equivalent CSP with reduced search space that does not lose any solution. A CSP is unsatisfiable if propagation detects a variable with an empty domain, *i.e.* all the values of this variable are inconsistent regarding the partial assignment that are already built.

Definition 2.3. (Propagation) [3] Consider CSP *P = (X, D, C)*. Propagation will return a new CSP *P' = (X, D', C)* such that:

- $D' \subseteq D$
- *(X, D', C)* is equivalent to *(X, D, C)*
- D' is non-empty partial solution

If no such CSP exists, propagation will return failure.

The level of removing inconsistent values depends on the level of consistency of each constraint and the communication between them.

Level of Consistency

Filtering all the inconsistent values of a problem is called enforcing its *global domain consistency* [11]. However, it is an NP-Hard problem. Therefore, propagation has to relax the level of consistency and it usually considers each constraint in isolation. There are different levels of consistency implemented with different filtering algorithms such as the bound consistency (BC) and the domain consistency (DC). A constraint c respects domain consistency (DC) if and only if all the values of its impacted variables are supported. While DC checks all values of variables, bound consistency (BC) only considers bounded values (lower and upper frontier of the domain). BC is a relaxation of DC and it requires less computation in comparison with DC. To choose suitable level of consistency for each constraint, users need to consider the trade off between the speed and the effectiveness of pruning.

[2] The definition is inspired from the note of Constrain Programming course [11]
[3] The definition is extracted from Lecture 3 of Constraint Programming course [11]

Propagation

In a CSP problem, there are many constraints. After calling a propagator of a constraint c, c will become consistent with regards to its chosen consistency level because all unsupported values of variables in its scope are removed. However, this reduction of the domains might violate the consistency of other constraints. Therefore, propagation needs to ensure communications between the local filtering algorithm of each constraint. Propagation calls each constraint successively until a fixed point is reached. The fixed point is a state of propagation process, where no constraint can perform any changes in the domain of variables.

Example 2.2. To see the communications between constraints through the domain, we add a constraint $c_2 \equiv x \geq 1$ to the example 2.1. The problem CSP has two variables x and y with the corresponding $D(x) = \{1, 2, 3, 4, 5\}$ and $D(y) = \{3, 4, 5, 6\}$. Two constraints c_1, c_2 are called successively to filter the domains of two impacted variables.

$$
\begin{array}{rl}
\mathrm{D(x)} = \{1,2,3,4,5\} & \mathrm{D(y)} = \{3,4,5,6\} \\
\downarrow & c_1 \equiv x + y \leq 5 \\
\mathrm{D(x)} = \{1,2,\cancel{3},\cancel{4},\cancel{5}\} & \mathrm{D(y)} = \{3,4,\cancel{5},\cancel{6}\} \\
\downarrow & c_2 \equiv x \geq 1 \\
\mathrm{D(x)} = \{\cancel{1},2\} & \mathrm{D(y)} = \{3,4\} \\
\downarrow & c_1 \equiv x + y \leq 5 \\
\mathrm{D(x)} = \{2\} & \mathrm{D(y)} = \{3,\cancel{4}\}
\end{array}
$$

Figure 2.1: Propagation process example

In the example 2.2, the propagation will give the same result even if two constraints are bound consistency or domain consistency. Bound consistency is called less in comparison with domain consistency so in this case applying the first level is more effective than the later.

2.1.4 Search

Besides performing propagation to prune the search space, Constraint Programming needs to explore the search space to find the solution. During the exploration, the search tree is constructed. The topology of tree is decided by the branching strategy and the exploration strategy.

Branching strategy

Branching strategy describes how to expand a node with only one requirement for the completeness, the union of the children CSPs is equal to the parent CSP. In general, branching strategy includes selecting a variable and then considering the values in its domain to define the CSPs of the child nodes [4].

[4]It is inspired from the note of Constrain Programming course [11]

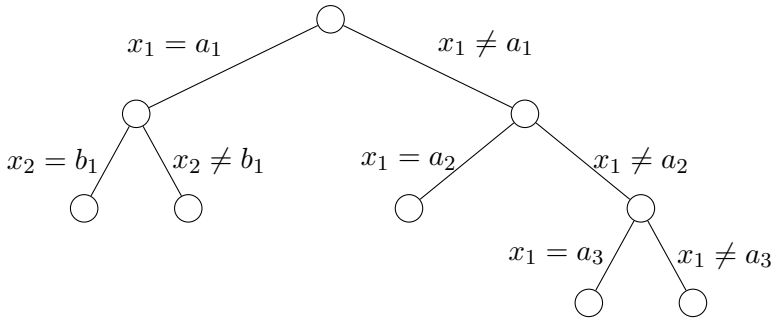

Figure 2.2: Example of tree search of binary labeling strategy

One of the most popular branching procedures is *binary labeling*: at non-leaf node, it creates two branches: one assigns a value to an unbounded variable and the other removes this value from the domain of variable.

A good variable and value heuristics have a large impact on tree size and choosing suitable heuristics can help to find a solution in fewer number of nodes. There are some general guidelines to help build good heuristics [5]:

- *First-Fail principle.* First-Fail principle is a variable heuristic. Its idea is that unsatisfiable CSP can be detected without the assignment to all variables. First-Fail principle considers *first a variable* that is *more likely to fail*, so the failure can be detected early and the further search can be avoided.
- *Best-First principle.* First-Fail principle is a value heuristic which selects first a value that is more likely to succeed. Obviously, choosing values that do not belong to a solution is pointless.

2.1.5 Constraint Optimization Problem

The goal of Constraint Satisfaction Problem is to find a valid solution which satisfies all the constraints or to prove no solution exists. In case there are many satisfied solutions, some are preferred than others by using the objective function to evaluate. This problem is called Constraint Optimization Problem (COP) and its definition is given in the following paragraph.

Definition 2.4. Constraint Optimization Problem (COP) is a specific case of Constraint Satisfaction Problems with an *Objective Function O* and it is fully described with *(X,D,C,O)*. The *objective function O* gives a numerical value to each solution of the problem $O : sol \rightarrow R$. It allows to compare solutions and to find the best one. A solution of COP *(X,D,C,O)* is a solution of CSP *(X,D,C)* which optimizes *O*. [6]

In order to turn a CSP into a COP, a variable *obj*, which represents the value of objective of solution, is added to the problem. This value is maintained in the model by using a constraint: *obj = O(sol)*. There are two approaches to find the optimal solution of a COP. As of now, without loss of generality, we only consider constrained minimization problems. The most commonly used techniques for finding the optimal solution in a search tree are "Branch and Bound" and "Iterative Optimization". Both of these techniques are introduced in the next paragraph.

[5] The heuristic principles are inspired from the note of Constrain Programming course [11]
[6] The definition is inspired from the note of Constrain Programming course [11]

1. **Branch and Bound**

   - Find the first solution having v as the value of objective variable. After that, the solver adds a constraint enforcing the objective variable to be strictly smaller than value v: $c_{opt} \equiv obj < v$.
   - Update value v each time a solution is found

   In this approach, the last solution found is the best.

2. **Iterative Optimization**

   - Calculate the *lower bound (lb)* of the objective variable. After that, add a constraint restricting the value of objective variable: $c_{opt} \equiv obj \leq lb$.
   - If a solution is found, it is the optimal. Otherwise, we increase the *lower bound (lb)* and restart. In the end, the solution found is the best one.

   With this approach, solver searches only one solution which is the optimality . If the search space is too large, iterative optimization possibly does not find any solution within a reasonable period of time.

2.2 Global Constraint

One of strength of Constraint Programming is to provide global constraints which are used to express more complex properties than simple binary constraints. It is stated in [12]: *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.* This part focuses on four global constraints used in the model of course timetabling problem of EPL.

2.2.1 AllDifferent

AllDifferent is a constraint used to express that all variables in its scope must be assigned different values. In semantic, it can be decomposed to a set of inequalities binary constraints like in example 2.3.

Example 2.3. Constraint $AllDifferent(x, y, z)$ is semantically equivalent to the three inequalities: $x \neq y$, $y \neq z$, $x \neq z$.

Although having the equivalent semantic, prunning on global constraint *AllDifferent* is stronger than considering set of binary constraints in isolation. *AllDifferent* Constraint relies on a maximum matching algorithm so it exploits its larger scope to detect more information about the unsatisfiable values of its variables. For instance, consider variables in example 2.3 with domains $D(x) = D(y) = \{1, 2\}$,$D(z) = \{1, 2, 3\}$. By considering these variables all at once, the dedicated algorithm detects that variable z can not take values 1, 2 otherwise it would lead to an infeasible problem. In contrast, these invalid values can not be detected in the decomposition of constraints. Each value of z has a support value in x when considering the constraint $x \neq z$ i.e. when z gets value 1, it is possible to assign x to 2 and vice versa. Similarly, constraint $y \neq z$ is also satisfiable with if z gets value 1 or 2. *AllDifferent* constraint detects invalid values because it expands the satisfiability to all variables in its scope not just considering pairs of variables separately.

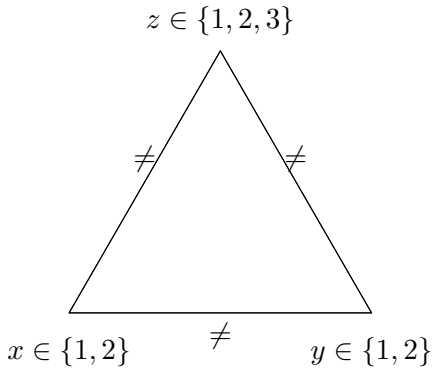

Figure 2.3: Decomposition of *AllDifferent* into set of binary constraints does not detect invalid values of variable z

2.2.2 Global Cardinality Constraint (GCC)

A generalization of the *AllDifferent* constraint is the global cardinality constraint. This constraint forces values v_i occur between l_i an u_i times in all variables in its scope. Its syntax is presented by the following equation:

$$GCC([x_1, ..., x_r], [v_1, ..., v_k], [l_1, ..., l_k], [u_1, ..., u_k])$$

Example 2.4. Let consider a course that has two exercise sessions. There are four enrolled groups (g_1, g_2, g_3, g_4) and each group needs to attend only one session out of two. For each group, there is one corresponding variable representing the exercise session. The domain of each variable is a set of sessions in the exercise part e.i {0,1}. It requires that there is at least one group in each session so there are maximum 3 groups in each slot. The requirement is modeled by the GCC constraint:

$$GCC([x_0, x_1, x_2, x_3], [0, 1], [1, 1], [3, 3])$$

The constraint means that in the array of four variables $X \equiv x_i | \forall i \in 0, ..., 3$, value 0 and 1 appear minimum one time and maximum 3 times. The bounded number of appearances are displayed in the array of $l_i \equiv [1, 1]$ and the array of $u_i \equiv [3, 3]$.One of possible assignments of GCC constraint is illustrated in the figure below.

x_0	x_1	x_2	x_3
{0,1}	{0,1}	{0,1}	{0,1}
↓		Possible assignment	
0	0	0	1

Table 2.1: Possible assignment of GCC constraint.

2.2.3 Bin-Packing Constraint

Bin-Packing Problem is one of the most popular problems in computer science. In this problem, there are n indivisible items, each one has a non-negative weight. There are m bins with their corresponding capacities. The question is that whether we can pack all items into the bins such

that the sum of the weights of items in any bin does not exceed the capacity of bin. Bin-Packing Constraint is used to model this problem. The syntax of constraint is described in the following formula:

$$binPacking([x_1, ..., x_n], [w_1, ..., w_n], [l_1, ..., l_m])$$

In this formula, x_i variable represents the index of bin which item i is packed into. The weights of items are stored in array $W \equiv w_i | \forall i \in 1, .., n$ and array L is the loads of bins. This constraint enforces the relation $l_j = \sum_i (x_i = j) * w_i, \forall j$. It makes the link between n weighted items (item i has a weight w_i) and the m different capacitated bins in which they are to be put [13].

Example 2.5. There are five items which have the weights: 1, 2, 3, 3, 4 in respectively. There are four bins, each has the capacity is 6. One possible assignment is to put three first items into bin one and the fourth and fifth are in bin two and three respectively:

$$binPacking([1, 1, 1, 2, 3], [1, 2, 3, 3, 4], [6, 3, 4, 0])$$

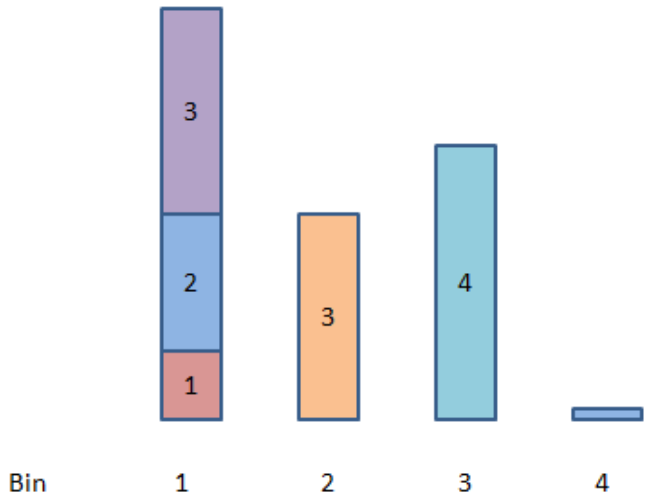

Figure 2.4: Bin Packing

2.2.4 Table constraint

Table constraint can represent any constraint, event it has no intentional definition [10]. Table constraint gives explicitly (in extension) the possible solutions for variables in its scope.

$$table([x_1, .., x_r], tab) \tag{2.1}$$

In this constraint, parameter tab lists all accepted combinations of variables in its scope [11]. The assignment of impacted variables is valid if and if only in the table.

Example 2.6. In some courses, its sessions need to occur consecutively. This requirement can be described by using table constraint to list all accepted slot patterns. The following formula models this constraint and the value of *tab* is displayed in the right table.

$table([session_1, session_2], tab)$

s1	s2
0	1
2	3
4	5
6	7

Chapter 3

Model

This chapter describes the data of course timetabling problem of EPL as well as the implemented model. The data was collected by Professor Francois Glineur when he used MIP technique to solve this problem. The data structure was slightly modified to fit easier with the Constrain Programming framework.

3.1 Problem description

UCL offers a 3 year bachelor program, including one and a half year in common core followed by 3 specialized semesters. At EPL, there are specializations in various fields such as Applied Chemistry and Physics, Constructions, Electricity, Information, Applied mathematics, Mechanical, Biomedical [4]. While the schedules in semester 1, 2 and 3 (S1, S2, S3) are fixed, each year EPL faculty has to reschedule for semester 4, 5 and 6 (S4, S5,S6). The reason is that from the second year, all students are supposed to choose a major and a minor option. The choice of specializations implies a fixed set of required courses, which the students have to take in semester 4 to 6.

Before going into detail, I will give an overview of the data problem and the relationship between entities. Each *student group* has a *program*, which contains a set of *courses* they have to take. For each course, there is a mandatory lecture part while the exercise part is optional. Each part is given by one or several professors. In some cases, the responsible professor of course is unknown at the scheduling time, so the lecturer will be empty. Each student group can be divided into one or several subgroups. Entity Relationship of course timetabling problem is illustrated in the figure 3.1. The following section describes the entities and their attributes.

3.1.1 The semester

There are two terms in each academic year and the detail of bachelor program is shown in table 3.1. It can be seen that S1, S3 and S5 occur in the first part of university year, so they need to be considered together when building timetable. The main reason is that some professors are responsible for courses in several semesters. By scheduling them together, timetable avoids the schedule conflict for professors. If a professor is responsible for courses in S1 and S3, he is considered to be unavailable in some fixed slots. This data is the input when scheduling semester 5, then it can be built independently. Similarly, from the schedule of S2, we have the busy time

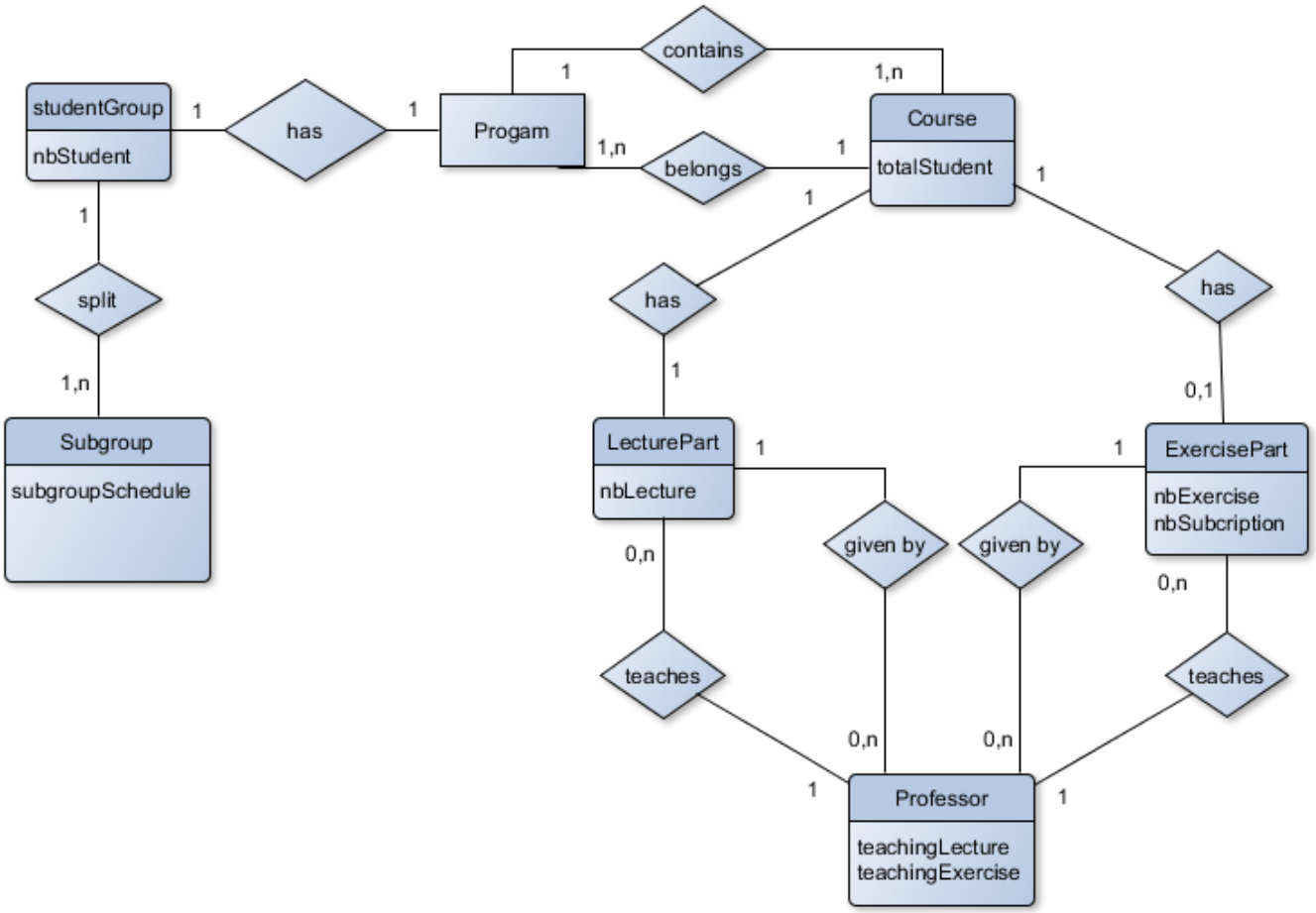

Figure 3.1: Entity Relationship of course timetabling problem

of professors who are in charge of semester S1 and S4 or S6. As a result, only S4 and S6 need to be scheduled together.

	September - December	January - June
Bachelor 1	S1	S2
Bachelor 2	S3	S4
Bachelor 3	S5	S6

Table 3.1: List semesters in 3 years of bachelor program

3.1.2 The group students and program

Definition 3.1. (Student Group). A student group is a group of people who have the same major and minor choice in the same year. A group is identified by a triple of major, minor and semester. All student groups are stored in *Groups* parameter.

Example 3.1. Group **EL4in4** includes all students choosing Electricity and Information and they study in the semester 4. Meanwhile, students who have the same choice but study in semester 6 are in group **EL6in6**

Definition 3.2. (nbStuPerGroup) *nbStuPerGroup* is a map representing the number of students

of each group. Notice that, at the scheduling period, the administrator has not received the enrollment of students. Therefore, the number of students in each group is calculated based on the statistic data of previous years.

Definition 3.3. (Programs). A *Program* is a set of courses that a student group has to take. These subjects depend on the students' options. All programs in the scheduled semester are in *Programs* parameter

Example 3.2. Program EL4in4 has 6 courses which students choosing Electricity and Information have to study in the semester 4.
Programs(EL4in4) = {LFSAB1106,LFSAB1803,LELEC1101,LELEC1370,LSINF1225, LSINF1252}

3.1.3 The course

As mentioned above, each program contains a set of courses, and each course can belong to several programs. A course includes theory lectures and it possibly has an exercise part. While fully attending lecture sessions is mandatory for all enrolled students, they only need to subscribe partially to multi-session of exercise part of some courses. In order to present information of courses, we used three parameters:

- (NbLecture) *NbLecture* is a map representing numbers of lecture sessions of all courses.
- (NbExercise) *NbExercise* is a map representing numbers of exercise sessions of all courses.
- (NbSubcriptionExercise) *NbSubcriptionExercise* is a map representing numbers of mandatory exercise sessions for enrolled students of all courses.

Definition 3.4. (NoExercise, FullSubscriptionExercise and PartialSubscriptionExercise) Each course belongs exclusively to one of three following categorizes: NoExercise, FullSubscriptionExercise and UniSubscriptionExercise

- (NoExercise) *NoExercise* contains courses which have no exercise part. These courses have: $NbExercise_{course} = 0$
- (FullSubscriptionExercise): In *FullSubscriptionExercise* courses, student are required to attend all exercise sessions. These courses have:
  $NbExercise_{course} = NbSubcriptionExercisecourse$ AND $NbExercise_{course} > 0$
- PartialSubscriptionExercise: In *PartialSubscriptionExercise* course, there are several exercise sessions but enrolled students need to take only one of them. These courses have:
  $NbExercise_{course} > NbSubcriptionExercise_{course}$ AND $NbExercise_{course} > 0$

Definition 3.5. (LectureSession, ExerciseSession) *LectureSession* is a teaching unit of theory part of a course and it takes one slot in the timetabling. Similarly, *ExerciseSession* represents for a teaching unit of exercise part of a course. Each session is identified by two properties: its course and its index session.

Notation 3.1. (LectureSession, ExerciseSession notation) Either lecture session or exercise session is denoted as $courseLabel^{index}$, where *courseLabel* is the name of the course and *index* is the position of this session in the lecture part.

Each lecture and exercise session has a specific number of attending students. Two parameters are used to store that information

- *TotStudentPerCourse* is a map representing the number of registered students in each course. It is also the number of student attending lecture sessions of each course.
- *AvgStudentPerExercise* is a map representing the average number of students in each exercise session.

Example 3.3. Course LAUCE1152 has two lecture sessions and two exercise slots, but it requires students to attend only one exercise session out of two. Following the definition 3.4, LAUCE1152 is a partial subscription exercise course. The total number of registered students in this course is 88, so on average each exercise session has 44 people. This information is stored in parameters:

- $NbLecture_{LAUCE1152} = 2$
- $NbExericse_{LAUCE1152} = 2$ and $NbSubcriptionExercise_{LAUCE1152} = 1$
- $TotStuLectureSession_{LAUCE1152} = 88$
- $AvgStuPerExerciseSession_{LAUCE1152} = 44$

LectureSession and *ExerciseSession* notation is used to distinguish different lecture and exercise sessions. All the information of LAUCE1152 course is illustrated in the following table.

Course	Subparts	Sessions		Number students	Note
LAUCE1152	Lecture	LectureSession:	$LAUCE1152^1$	88	Attend 2 sessions
			$LAUCE1152^2$	88	
	Exercise	ExerciseSession:	$LAUCE1152^1$	44	Attend 1 out of two
			$LAUCE1152^2$	44	

Table 3.2: Example of data of a course

3.1.4 Subgroups

As mentioned above, students in the same group can have different exercise slots of partial subscription course. Therefore, a group needs to be divided into smaller units, in which students have the same schedule in both lecture and exercise sessions.

Definition 3.6. (Subgroup) *Subgroup* is a set of one or several students who are in the same group and take the same exercise sessions, including both full subscription or partial subscription exercise course. (Subgroups) *Subgroups* is a set containing all subgroups. Each subgroup is represented by two properties, the name of group and its position in this group.

The division of a student group into smaller units also helps to arrange these blocs into different exercise sessions of the same course. As a result, the number of students in these exercise slots becomes more balanced than when using group unit.

Definition 3.7. (enrolled) *enrolled* is a set of pairs (subgroup,course) such that the subgroup belongs to a group having this course in their program.

$$\begin{aligned} &\forall course \in Courses, \forall group : course \in Programs_{group} \\ &\forall subgroup \in group : (subgroup, course) \in enrolled \end{aligned} \tag{3.1}$$

Example 3.4. Course LAUCE1152 in the example 3.3 has two enrolled groups, **ME6gc6** and **GC6me6** and the number of students of two groups is 30 and 20 respectively. Each group is divided into two smaller groups. The figure above describes one possible arrangement of subgroups into different exercise sessions.

Group	# student Group	Subgroup	# student SubGroup	Exercise session	
				$LAUCE1152^1$	$LAUCE1152^2$
ME6gc6	30	$ME6gc6^1$	15	x	
		$ME6gc6^2$	15		x
GC6me6	20	$GC6me6^1$	10	x	
		$GC6me6^2$	10		x

Table 3.3: Example of arranging subgroups into exercise sessions

It can be seen in the example that the numbers of students attending exercise session are equal. This can not be achieved if students in the same group are arranged to take the same exercise session. For example, if students in group **ME6gc6** take one exercise session and group **GC6me6** takes the other, then the deviation of number of students between two exercise sessions is 10.

Clearly, organizing students in groups leads to the imbalance between exercise sessions. Therefore, student groups need to divide into smaller units to improve the balance between exercise sessions.

3.1.5 Professors

Definition 3.8. (Professors) *Professors* are the set of all teachers giving at least a course in the scheduled semesters. Each professor is responsible for a set of courses and this information is stored in two parameters, *LectureGivenByProf* and *ExerciseGivenByProf*:

- *LectureGivenByProf* is a map containing courses in which professors have lecture sessions.
- *ExerciseGivenByProf* is a map containing courses in which professors are in charge of exercise parts.

Parameter 3.1. (UnavailableSlots) UnavailableSlots is a map representing slots at which professors can not teach. (For example, the professor has other cousers in semester S1,S2, and S3)

Parameter 3.2. (NonPreferredSlots) NonPreferredSlots is a map representing slots at which professors can teach but they prefer to avoid. This information is used to optimize the preference of professors

3.1.6 The slots

The slots of any course timetabling problem are characterized by the number of slots per day and the number of days studying each week. The range of possible slots per week (*rSlots*) is the product of two former parameters.

```
nbDays          // Is the number of days studying per week
nbSlotPerDay    // Is the number of slots in each day. In the schedule of EPL, each time
                // slot always composes of two consecutive hours, and each day has four time slots
nbSlots         // Is the number of slots in a studying week
                // = nbDays * nbSlotPerDay
rSlots          // range of valid slots
```

3.1.7 The problem parameters

This part describes parameters used to define some constraints and objectives.

Definition 3.9. (fixedLecture,fixedExercise) Some lecture or exercise sessions are preassigned to specific slots due to external reasons. Fixed sessions are stored into two parameters: *fixedLecture* containing pairs of (lectureSession,slot) and *fixedExercise* containing pairs of (exerciseSession,slot). Those values represent some sessions which are fixed to specific slots in the timetable.

Definition 3.10. (Margin) *Margin* is an input parameter which defines the degree of balance of the number of students in different exercise sessions of a course. In the equation 3.2, *NbStudentInExerciseSession* is a map representing the number student attending each exercise session.

$$\begin{aligned} &\forall course \in PartialSubscriptionExercise \\ &\quad \forall session \in ExerciseSession_{course} : \\ &\qquad NbStudentInExerciseSession_{session} \geq Margin * AvgStudentPerExercise_{course} \end{aligned} \tag{3.2}$$

The *Margin* parameter gets the value from 0 to 1 ($Margin \in [0, 1]$). It defines the lower bound of number of students in each exercise session.

- If $Margin = 1 \implies NbStudentInExerciseSession_{session} = AvgStudentPerExercise_{course}$
  It means the number of students in each exercise session of the partial subscription course is strictly equal.
- If $Margin = 0 \implies NbStudentInExerciseSession_{session} \geq 0$
  It means there is no requirement of the number of students in each exercise session so distributing subgroups into exercise sessions is not constrained.

Definition 3.11. (QualitySlots) *QualitySlots* is an array which associates each slot with a number representing the expectation degree to have a teaching unit in this slot. In general, slots in the middle of day have higher quality than slots at the beginning or at the end of day. This parameter is used for maximizing the quality of timetable.

Slot Hour	Monday	Tuesday	Wednesday	Thursday	Friday
8:30 - 10:30 am	0.9	0.9	0.9	0.9	0.9
10:45 - 12:45 am	1	1	1	1	1
2:00 - 4:00 pm	1	1	1	1	0.5
4:15 - 6:15 am	0.5	0.5	0.5	0.5	0

Table 3.4: Quality of slots

Definition 3.12. (ContinuousSlots) *ContinuousSlots* is a set of pairs of consecutive slots. The two continuous slots are in a half of day, in the morning or in the afternoon.

	Monday	Tuesday	Wednesday	Thursday	Friday
Morning	0	4	8	12	16
	1	5	9	13	17
Afternoon	2	6	10	14	18
	3	7	11	15	19

Table 3.5: Continuous Slots

In the table 3.5, the two slots in the same day and having the same color are continuous. This information is used to schedule continuous courses which require its lecture and its exercise sessions occur consecutively. These courses are stored in *ContinuousCourse* parameter

3.2 Variables and Domain

Variable 3.1. *slotLecture* is a set of variables representing slots scheduled for each lecture session. *slotLecture* associates all lecture sessions with a slot in the corresponding course timetable.

$$\forall session \in LectureSession : slotLecture_{session} \in rSlots \tag{3.3}$$

Variable 3.2. *slotExercise* is a set of variables representing slots scheduled for each exercise session. *slotExercise* associates all exercise sessions with a slot in the corresponding timetable.

$$\forall session \in ExerciseSession : slotExercise_{session} \in rSlots \tag{3.4}$$

Notation 3.2. $slotLecture_{course}$ is an array of all lecture sessions of a course, while all exercise sessions of a course are presented in $slotExercise_{course}$

Variable 3.3. *professorSchedule* is a set of variables representing slots of both lecture and exercise sessions that a professor is in charge of.

$$\begin{aligned} &\forall p \in Professors : \\ &\quad professorSchedule_p = slotLecture_{lectureCourses} \cup slotExercise_{exerciseCourses} \\ &\qquad with: \quad \forall lectureCourses \in LectureGivenByProf_p \\ &\qquad and \quad \forall exerciseCourses \in ExerciseGivenByProf_p \end{aligned} \tag{3.5}$$

Variable 3.4. *subGroupExerciseIndex* is a set of variables representing the index of session which subgroups take for their exercise part of the partial subscription course. *subGroupExerciseIndex* associates to each pair of (subgroup, partialSubscriptionExercise course). Notice that, only the variables of subgroups for partial subscription exercise course are introduced. In the full subscription exercise course, all registered students have to attend all sessions so it is not necessary to define a specific variable for each subgroup.

The domain of a variable of *subGroupExerciseIndex* is the possible sessions of its course, ranging from 0 until the number of exercises of the corresponding course

$$\begin{aligned} &\forall (subgroup, course) \in enrolled : \\ &\quad subGroupUniExercise_{subgroup,course} \in 0, .., NbExercise_{course} - 1 \end{aligned} \tag{3.6}$$

Variable 3.5. (SubGroupSchedule) *SubGroupSchedule* is a set of variables representing slots scheduled for a subgroup, in which there is at least one lecture or exercise session of their enrolled courses. *SubGroupSchedules* is a map containing the timetable of all subgroups.

The curriculum of each subgroup is composed of two components: the lecture sessions and the exercise slots of both the full subscription and the partial subscription exercise courses.

3.3 Constraints

After introducing the basic variables, I will present the constraints holding on previously defined variables.

3.3.1 Constraint 1: No conflict in sessions of each course

Both the lecture and exercise sessions of each course do not overlap such that students can fully attend the theory lecture and the exercise part. This requirement also allows students to choose to study different sessions in partial subscription exercise course.

$$\begin{aligned} &\forall course \in Courses : AllDifferent(courseSessions) \\ &\quad with \quad courseSessions = slotLecture_{course} \cup slotExercise_{course} \end{aligned} \tag{3.7}$$

3.3.2 Constraint 2: No conflict in the schedule of professors

The schedule of each professor must not conflict because he cannot teach two courses at the same time.

$$\forall p \in Professors, AllDifferent(professorSchedule_p) \tag{3.8}$$

3.3.3 Constraint 3: No teaching session in unavailable slots of professors

Each professor has several unavailable slots. Therefore, course sessions are not allowed to be assigned to slots at which the responsible professors are busy.

$$\forall p \in Professors, \forall session \in professorSchedule_p : sesion \notin UnavailableSlots_p \tag{3.9}$$

3.3.4 Constraint 4: Maximum capacity for each slot

The number of classes occurring at the same time is restricted because of the classroom limitation. (Default is 50 activities).

$$\forall s \in rSlots : \sum_{session \in slotLecture} session == s + \sum_{session \in slotExercise} session == s \leq Capacity \tag{3.10}$$

3.3.5 Constraint 5: Assignment sessions

Some lecture and exercise sessions are preassigned to specific slots. Those pre-assignments are declared in *fixedLecture*, *fixedExercise*.

$$\begin{aligned} &\forall (session, slot) \in fixedLecture: \quad slotLecture_{session} = slot \\ &\forall (session, slot) \in fixedExercise: \quad slotExercise_{session} = slot \end{aligned} \tag{3.11}$$

3.3.6 Constraint 6: Breaking symmetry between sessions

In a course having two lecture sessions, swapping the values of the two slots does not change the schedule of course. However, the search detects it as a new solution because the variables of lecture sessions of this course are assigned to new values. Therefore, a new constraint is added to avoid symmetry between variables of lecture or exercise sessions. Notice that, this constraint only applies to non-fixed sessions.

$$\begin{aligned} &\forall course \in Courses, \forall i \in [fixed, .., NbLecture_{course} - 1]: \\ &\qquad slotLecture_{course^i} < slotLecture_{course^{i+1}} \quad with \quad fixed = |fixedLecture_{course}| \\ &\forall course \in Courses, \forall i \in [fixed, .., NbExercise_{course}]: \\ &\qquad slotExercise_{course^i} < slotExercise_{course^{i+1}} \quad with \quad fixed = |fixedExercise_{course}| \end{aligned} \tag{3.12}$$

3.3.7 Constraint 7: Courses having continuous sessions

There are two cases of continuous courses. First, if the course has two lecture sessions, then these sessions belong to the slot pattern of *ContinuousSlots* parameter. Secondly, if the course has one lecture and at least one exercise sessions, then the lecture and the first exercise session belong to the slot pattern of *ContinuousSlots* parameter.

$$\begin{aligned} &\forall course \in ContinuousCourses: \\ &\qquad \{session1, session2\} \in ContinuousSlots \\ &\quad with \quad if \quad NbLecture_{course} == 2: \\ &\qquad\qquad session1 = slotLecture_{course^1},\ session2 = slotLecture_{course^2} \\ &\qquad if \quad NbLecture_{course} == 1 \text{ AND } NbExercise_{course} \geq 1: \\ &\qquad\qquad session1 = slotLecture_{course^1}, session2 = slotExercise_{course^1} \end{aligned} \tag{3.13}$$

3.3.8 Constraint 8: No conflict in student schedule

One of the main requirements of course timetabling problem is to avoid conflict in student schedule because the students can not attend more than one teaching sessions at the same time.

$$\forall subgroup \in SubGroups: AllDifferent(SubGroupSchedules_{subgroup}) \tag{3.14}$$

However, this requirement makes the problem over-constrained in some instances, hence it was considered 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 [21]. The number of conflicts in each subgroup schedule is stored in *SubGroupViolation*, which associates a pair of

(subgroup,number violation). The first objective is to minimize the total cost violation of all subgroups.

3.3.9 Constraint 9: Balance between exercise sessions

The numbers of students in exercise sessions of the same course need to be balanced. These numbers must be strictly equal when *Margin* parameter is set to one. However, this requirement is often too strong for realistic instances and makes the problem become over-constrained. For example, a course has two exercise sessions and students are required to attend one of them. There are an odd number of students registering this course. In MIP model, the number of students can be split equally into two sessions because the variable can get a real number. However, this requirement cannot be satisfied in CP model, in which the domain of variables is an integer. In addition, even if it is possible to split students equally in exercise sessions of some courses, finding an exact way to arrange subgroups takes a long time. Therefore, this requirement is relaxed by introducing a new parameter *RelaxMargin*, which is smaller than *Margin*. The ratio of students attending in each exercise sessions over the average of number of students is at least equal to *RelaxMargin*. Thus, an objective is introduced to minimize the deviation between exercise sessions.

$$\begin{aligned}&\forall course \in PartialSubscriptionExercise\\ &\quad \forall session \in ExerciseSession_{course} :\\ &\qquad NbStudentInExerciseSession_{session} \geq RelaxMargin * AvgStudentPerExercise_{course}\end{aligned} \tag{3.15}$$

3.4 Optimization

In this part, the definition of objective functions of course timetabling problem is introduced. These objectives cover the requirements of all stakeholders, namely the administrators, the professors and the students. These formulations are inspired from the model of Professor Francois Gilneur [15].

3.4.1 Objective 1: Minimizing the total conflicts in student schedule

The students are organized into subgroups, so the total number of conflicts in student schedule is calculated based on the violation of subgroup schedule. *SubGroupViolation* is a map containing the number of conflicts of each subgroup. *NbStudentSubgroup* is a map representing the number of students in each subgroup and it is calculated when dividing groups into smaller units.

$$TotalConflict = \sum_{sg \in SubGroups} SubGroupViolation_{sg} * NbStudentSubgroup_{sg} \tag{3.16}$$

The total conflicts in student schedule is the first objective because the solution of the timetable needs to maximize the comfort of students globally.

3.4.2 Objective 2: Minimizing the total conflicts in non preferred slots of professor schedule

Each professor has a set of non preferred slots, at which they prefer to avoid teaching. The number of conflicts in non preferred slots of each professor p is calculated in the equation below.

$$NonPreferredConflict_p = \sum_{sesion \in ProfessorSchedule_p} \sum_{s \in NonPreferredSlots} session == s \quad (3.17)$$

After calculating the number of conflicts in non preferred slots of each professor, the total conflicts and the maximum of conflicts of each instructor are computed.

Sub-Objective 2.1 - Minimizing the total conflicts in non preferred slots of all professors

This objective focuses on minimizing the total conflicts of all professors and it does not care about each individual professor.

$$TotalNonPreferredConflict = \sum_{p \in Professors} NonPreferredConflict_p \quad (3.18)$$

Sub-Objective 2.2 - Minimizing the maximum number of conflicts of a professor

Instead of concentrating on the total conflicts, this objective tries to minimize the maximum number of conflict of a professor. It is expected to improve the fairness between professors.

$$MaxNonPreferredConflict = max(NonPreferredConflict_p) \,|\, \forall p \in Professors \quad (3.19)$$

Notice that, the formulations of two sub-objectives are different from the equation proposed in MIP model. In this model, the score of conflicts of each professor is the product of his number of conflicts in non preferred slots and a real number. The real number is computed to ensure the fairness between professors. However, the domains of variables are integer, so the formulation was changed to fit with the CP framework.

3.4.3 Objective 3: Maximizing the quality of schedule

The quality of schedule depends on the value of teaching slots and the number of attending students. Instead of using the real number of students studying in each exercise session, we use the average number of students in one session of this course. There are two main reasons for this choice. First, this computation is faster than using the real number of students because the quality of schedule is computed when all *slotLecture* and *slotExercise* variables are assigned. The quality of schedule does not depend on distributing subgroups to exercise session. Secondly, after optimizing all objectives, I will minimize the deviation of students in each exercise session. At this time, the real number of students attending each session will change.

As mentioned above, in partial subscription exercise, students attend only one exercise session, while they are required to fully subscribe all the lecture parts. Therefore, the number of students in a lecture session is the total students enrolled this course (*totStudentPerCourse* parameter). Meanwhile, the number of students in exercise session is the average number of students in

one session of this course (*AvgStudentPerExercise* parameter). The value of each session is the product of the quality of the slot and the number of attending students. Two intermediate variables are introduced *valueLecture* and *valueExercise*. The total value of lecture part is stored in *valueLecture* variable and *valueExercise* contains the value of exercise part.

$$valueLecture = \sum_{session} \quad qualitySlots(session) * totStudentPerCourse_{course} \\ with \quad course \in Courses, \quad session \in LectureSession_{course} \tag{3.20}$$

$$valueExercise = \sum_{session} \quad qualitySlots(session) * AvgStudentPerExercise_{course} \\ with \quad course \in Courses, \quad session \in ExerciseSession_{course} \tag{3.21}$$

After computing two intermediate variables, the quality of schedule is calculated. In the below formula, *totStudentAllLecture* and *totStudentAllExercise* are the sum of total number students in all lecture sessions and exercise sessions respectively.

$$qualitySchedule = \frac{(valueLecture * \alpha + valueExercise)}{totStudentAllLecture * \alpha + totStudentAllExercise} \tag{3.22}$$

The factor α is added into *valueLecture* to define the priority of the lecture sessions and exercise sessions. In my implementation, α is set to 2 because the improvement the quality of lecture sessions are considered to be more important than the quality of exercise sessions (*valueExercise*).

In the formula 3.22, two parameters *totStudentAllLecture* and *totStudentAllExercise* are constant. Therefore, maximizing the quality of schedule is reduced to maximizing the weighted function ($valueLecture * \alpha + valueExercise$). This function is also defined as the third objective.

3.4.4 Objective 4: Minimizing the total number of students attending the end slot of day

This objective is also used to improve the quality of schedule and it focuses on minimizing the number of students studying in the end slot of day. In the following formula, *valueLectureEndSlots* is the total number of students attending lecture sessions in the end slot of day. Similarly, *valueExerciseEndSlots* stores the total number of students participating exercise sessions in the end slot of day.

$$valueLectureEndSlots = \sum_{session} \quad endSlots(session) * totStudentPerCourse_{course} \\ with \quad course \in Courses, \quad session \in LectureSession_{course} \tag{3.23}$$

$$valueExerciseEndSlots = \sum_{session} \quad endSlots(session) * AvgStudentPerExercise_{course} \\ with \quad course \in Courses, \quad session \in ExerciseSession_{course} \tag{3.24}$$

endSlots is an array representing whether a slot is in the end of day. Therefore, the indices of the end slots receive value 1 while the other slots get value zero.

Slot Hour	Monday	Tuesday	Wednesday	Thursday	Friday
8:30 - 10:30 am	0	0	0	0	0
10:45 - 12:45 am	0	0	0	0	0
2:00 - 4:00 pm	0	0	0	0	0
4:15 - 6:15 am	1	1	1	1	1

Table 3.6: The end slots of day

Based on the value of two intermediate variables, the quality of schedule in the end slots is computed in the following equation:

$$qualityEndSlots = \frac{(valueLectureEndSlots * \alpha + valueExerciseEndSlots)}{totStudentAllLecture * \alpha + totStudentAllExercise} \tag{3.25}$$

Based on the formula 3.25, the denominator is a constant. Therefore, minimizing the total number of students attending the end slots of day can be reduced to minimizing the function $(valueLectureEndSlots * \alpha + valueExerciseEndSlots)$. This is also the function of the fourth objective.

3.4.5 Objective 5: Minimizing the total deviation of number of students exercise sessions

In each partial subscription exercise session, the difference between the real number students attending and the average number of students in a exercise session is computed. The last objective is to minimize the total deviation of number of students in exercise sessions. By doing this, the distribution of students in exercise parts becomes more balanced. In the equation 3.26, *NbStudentInExerciseSession* contains the number of students attending each exercise session and it depends on they way of the arrangement subgroups into different exercise slots. The purpose of the objective can be visualized in the figure 3.2.

$$totalDeviation = \sum_{session} |NbStudentInExerciseSession_{session} - AvgStudentPerExercise_{course}| \\ with \quad course \in Courses, \quad session \in ExerciseSession_{course} \tag{3.26}$$

In order to evaluate the quality of the fifth objective, the ratio of the total deviation of number of students attending exercise sessions over the total number of students in all exercise sessions is computed.

$$percentageOfDeviation = \frac{totalDeviation * 100}{totStudentAllExercise} \tag{3.27}$$

In summary, this chapter describes all constraints of the course timetabling problem and defines its objective functions. The differences between CP model and MIP model also have been laid out.

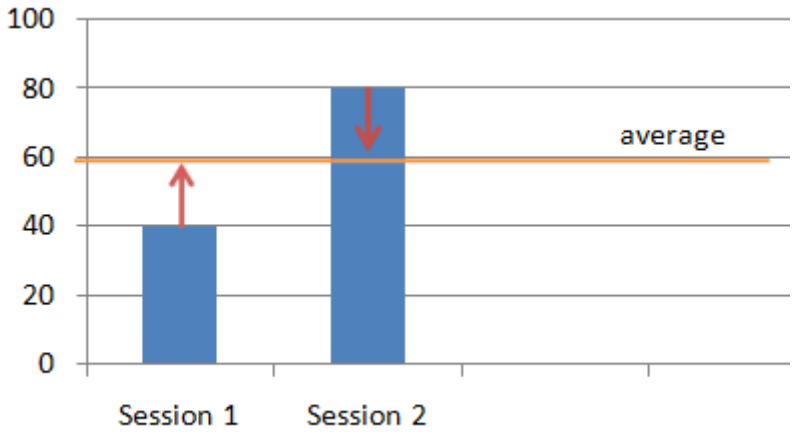

Figure 3.2: The deviation of number of students in exercise sessions of one course

Chapter 4

The Solver

The previous chapter described the details of model, including the variables, their domains and all the objectives of the problem. This chapter presents the way this model is implemented in *Oscar solver*.

4.1 The variables

```
/* Course sessions */
 val slotLecture = Map[String, Array[CPIntVar]]()
 val slotExercise = Map[String, Array[CPIntVar]]()

/* Professor schedule */
 val professorSchedule = Map[String,Array[CPIntVar]]()

/* Subgroup schedule */
val subGroupSchedule = Map[(String,Int), Array[CPIntVar]]()
```

All the variables are CPIntVar type and their domains are possible slots of a week (the *rSlots* range of the previous chapter).

Each course in the programs has two parts: lecture theory and exercise sessions. **Lecture theory** information is stored in *slotLecture* map with the courseID as key and an array of the variables as value. The size of each array is the number of lecture sessions of its corresponding course. Similarly, courses having **exercise sessions** are represented in *slotExercise* map with courseID as key and exercise sessions stored in an array of the variables as value.

professorSchedule represents the teaching schedule for each professor. It is a map with professorID as key, and an array of the variables as value. This array represents all teaching sessions of a professor, including both lecture and exercise sessions. To get the schedule of each professor, we just refer to the created variables of his responsible courses.

subGroupSchedule represents the weekly studying timetable for several students having same schedule. This is a map with key composed of the group name and the index of the subgroup in the parent group. Value of the map represents all variables of courses which the subgroup enrolled. In lecture sessions and full subscription exercise sessions, all registered students have to fully attend, so we refer to created variables (*slotLecture*, *slotExercise* variables). However, if

a course has the partial subscription exercise part, a new variable is created to represent the index of the exercise session at which the subgroup attends. All these variables are stored in the *subGroupExerciseIndex* map. The domain of these variables is the possible exercise sessions of the course which subgroups could take (range from 0 until $NbExercise_{course}$).

4.2 Constraints

4.2.1 Constraint 1: No conflict in sessions of each course

We set *allDifferent* constraint on all variables of a course.

```
/*
 * C1: Lecture and exercise sessions of the same course are in different slots
 */
 for(c <- courses){
     val allSessionOfCourse = slotLecture(c) ++ slotExercise.getOrElse(c, Array())
     if(allSessionOfCourse.size > 1)
         add(allDifferent(allSessionOfCourse))
 }
```

4.2.2 Constraint 2: No conflict in the schedule of professors

We also use *allDifferent* constraint on all variables of a professor schedule to ensure that he is only in charge of one teaching unit for each slot.

```
/*
 * C2: A professor can not teach 2 different sessions at the same time
 */
 for(p <- professors){
     val sessionOfProfessor = professorSchedule.getOrElse(p, Array())
     if(sessionOfProfessor.size > 1){
         add(allDifferent(sessionOfProfessor))
     }
 }
```

4.2.3 Constraint 3: No teaching session in unavailable slots of professors

We iterate over all variables of a professor schedule and make sure that they are not in his unavailable slots. These invalid values of variables of teaching sessions can be removed in the initialization of variables. However, we decided to model this requirement separately so that the code can be modularized.

```
/*  C3: No lecture or exercise occurs in the unavailable slots of the responsible
    professors */
 for((p,unvailableSlot) <- data.unavailableTimeOfProf){
     val sessionOfProfessor = professorSchedule.getOrElse(p, Array())
     for(sessionVar <- sessionOfProfessor)
         unvailableSlot.map(un => add(sessionVar != un))
```

```
}
```

4.2.4 Constraint 4: Maximum capacity for each slot

For each slot, the number of teaching sessions does not exceed the given maximum number of activities. *GCC* constraint is used to model this requirement

```
/*
 * C4: Limitation of number of teaching units occurring at the same time
 */
val allSessions = ( slotLecture.values ++ slotExercise.values).toArray
add(gcc(allSessions.flatten, rSlots,0,data.maxActivities),Weak)
```

4.2.5 Constraint 5: Assignment sessions

This constraint assigns each fixed slot to the teaching session. We also ensure that the number of fixed slots does not exceed the number of lecture sessions of the course. The fixed exercise sessions are processed in the same way.

```
/*
 * C5: fixed lecture and exercise slots of courses
 */
for((course,fixedSlots) <- data.fixedLectureOfCourse){
    for(i <- 0 until math.min(fixedSlots.size,data.nbLecture(course)) ){
        add(slotLecture(course)(i) == fixedSlots(i))
    }
}
```

4.2.6 Constraint 6: Breaking symmetry between sessions

In a course having two lecture sessions, if the values of the two slots are swapped then the search considers it a new solution although the schedule of course does not change. To avoid variable symmetry, we added a constraint that forces the lecture slots to be strictly increasing. Notice that, this additional constraint is only enforced on non-fixed variables. Breaking symmetry of exercise variables is similar so we do not present the code here.

```
/*
 * C6: Break symmetry between lecture sessions or exercise sessions of the same
     course
 */
for((course,lectureSessions) <- slotLecture){
    val from = data.fixedLectureOfCourse.getOrElse(course, Array()).size
    // all the non-fixed lecture sessions of course
    if( lectureSessions.size - from > 1)
        for(i <- from until lectureSessions.size -1){
            add(lectureSessions(i) < lectureSessions(i+1))
        }
}
```

4.2.7 Constraint 7: Courses having continuous sessions

This requirement is modeled by using *Table* constraint. In this constraint, *continousSlots* is a table listing continuous slot patterns. There are two cases of continuous slots: the course has two lecture sessions or the course with one lecture and at least one exercise session. Two cases are checked and the corresponding variables are set.

```
/*
  * C7: Continuous sessions
  */
 for(course <- data.continuousCourses){
   // course has one lecture and at least one exercise session
     if(data.nbLecture(course) == 1 && data.nbExercise(course) >= 1){
         add(table(slotLecture(course)(0), slotExercise(course)(1), continousSlots))
     }
   // course has two lecture sessions
   else if(data.nbLecture(course) == 2){
         add(table(slotLecture(course)(0), slotLecture(course)(1), continousSlots))
   }
 }
```

4.2.8 Constraint 8: No conflict in student schedule

One of the most important requirements of the course timetabling problem is that there is no conflict in student schedule. However, with some data instances, this requirement makes the problem over-constrained so it is considered as a soft constraint. The violation of each subgroup schedule is stored in *subGroupViolation* variable

```
val subGroupViolation = Map[(String,Int),CPIntVar]()

 /* C8: Calculate the conflict schedule of each subgroup */
 for(group <- groupStudent; i <- 0 until nbSubGroups(group)){
     add(softAllDifferent(subGroupSchedule(group,i), subGroupViolation(group,i)) )
 }
```

4.2.9 Constraint 9: Balance between exercise sessions

The numbers of students in exercise sessions of the same course need to be balanced. However, this requirement is often too strong for realistic instances so it was relaxed. It only requires that the ratio of students attending in each exercise sessions to the average of number of students is at least equal to *RelaxMargin*.

This requirement is described by using a *binPacking* constraint since the arrangement of subgroups into exercise sessions is similar to put items into bins. The minimum number of students attending each exercise session of a course is calculated in line 8. This number is also the lower bound of the bins (line 11). The exercise variables of all subgroups attending this course are stored in *varOfExerciseSubGroup* array. The number of students of each subgroup is stored in *nbStuInSubGroup* array and it is considered as the weight of each item in the bin packing problem (line 16).

```
/*
  * C9: Balancing between exercise session :
  *     Require the minimum number of students attending each exercise sessions of
      a course
  */
for(course <- partialSubscriptionExercise){
   ...
   // the minimum number of students
     val minStudentPerSlot = (avgStuPerExerciseSession(course) * RelaxMargin).toInt

   // the number of students in each sessions are stored in loads array
     val loads = Array.fill(nbExercise)(CPIntVar(minStudentPerSlot to
         maxStudentPerSlot))

   // all variables of subgroup in partial subscription exercise sessions
     val varOfExerciseSubGroup : Array[CPIntVar]()
   // the number students in the corresponding subgroup
     val nbStuInSubGroup : Array[Int]()

     ...

     add(binPacking(varOfExerciseSubGroup, nbStuInSubGroup, loads))
 }
```

4.2.10 Constraint 10: Additional constraint

The previous constraint ensures that the number of students in each partial subscription exercise reaches the minimum. However, *binPacking* constraint is only trigged when the variables of the index of subgroup in exercise session are assigned (*subGroupExerciseIndex* variables). However, in some cases where a exercise session is possibly assigned to a slot at which there is not enough number of attending students.

Example 4.1. Course A has two sessions but it requires each registered student to attend one of them. There are four groups (g1,g2,g3,g4) taking this course. In slot 2, g1 has a lecture of course B while g2 takes course C 4.1. Only two groups, g2 and g4 can participate the exercise of course A, if its session is scheduled to slot 2. Parameter Margin is 0.8, so the exercise session of course A needs at least 16 students. (0.8 * 20 = 16). Therefore, the variable of exercise sessions of course A should remove value 2 because only 7 students can attend this slot.

Slot	0	1	2	3	4	..	19
g1: 20 stu			Lec: B^1				
g2: 3 stu			v				
g3: 13 stu			Lec: C^1				
g4: 4 stu			v				

Table 4.1: Example of the availability of groups. The variables of exercise sessions need to remove the slots at which do not have enough attending students

In order to remove the invalid value early, an extra constraint was added. This one removes the values of the variables of partial subscription exercises which do not have enough attending students. It is triggered when all lecture sessions and full subscription exercise are bounded and

before the variables of partial subscription exercise courses are assigned. The detail of the code of this constraint is presented in the appendix.

After the pruning of this constraint, the number of available students in all values of the domain of the exercise variables is greater or equal to the minimum requirement.

4.3 Division student groups

From the previous chapter, we know that student groups need to be divided into smaller units, which contain several students having the same schedule in both lecture and exercise sessions. These units can be arranged into multiple sessions of partial subscription exercise courses. There are two approaches to split a large group. It can be divided into atomic units (individual students) or subgroups (coarse-grained level). In this section, the benefits and drawbacks of two levels of granularity are presented.

4.3.1 Organizing students in subgroups reduces the execution time of finding the solution

In data instances of EPL's problem, each course usually has 1 lecture session and 2 exercise sessions which requires students to attend one of them. Each program has about 6 courses, so in total, 18 sessions need to be arranged for each program. There are 20 slots in the weekly timetable, so the density of sessions is dense. In addition, different programs have several courses in common. As a result, teaching sessions in a same program can overlap each other, especially exercise sessions of different courses. In this case, the scheduling tool needs to find a correct way to arrange subgroups to minimize the total conflict in the student schedule and also ensure the equal number of students attending in exercise sessions of the same course. In this situation, grouping students into subgroups can make the execution time faster than organizing individual students. The reason is that the search algorithm does not need to compute and verify the permutation of the n students in each subgroup.

Example 4.2. Once the schedule of lecture and exercise sessions is planned, subgroups are arranged into suitable exercise slots. Suppose three courses X,Y,Z are partial subscription exercise and their exercise sessions are scheduled as shown in the following table. Assume that subgroup A has 15 students who enrolled on three courses. During the search process, all exercise sessions of group A are assigned to slot 2. The solver detects the conflicts and tries to reduce it. It moves group A to the second session of course Y (slot 5). If students were organized separately then it would need to make 15 times changes to swap students to the second session.

slot	0	1	2	3	4	5	6
Course X			ex: X^1		ex: X^2		
Course Y			ex: Y^1			ex: Y^2	
Course Z			ex: Z^1				ex: Z^2

Table 4.2: Choosing the schedule for subgroup A taking three courses X,Y,Z

Furthermore, the clustering students into subgroups decreases the number of variables and the number of constraints. As a result, the execution time can be improved.

4.3.2 Organizing students in larger units leads to an imbalance of the number of students between exercise sessions

Despite the previously mentioned advantage, gathering students in larger units can provoke an imbalance between exercise sessions.

Example 4.3. A course has two exercise sessions and requires enrolled students to take only one of them. There are 15 people taking this course. As shown in figure 4.1, grouping students by groups of 5 people has the minimum deviation of 5 whereas considering individuals allows to reduce that to 1.

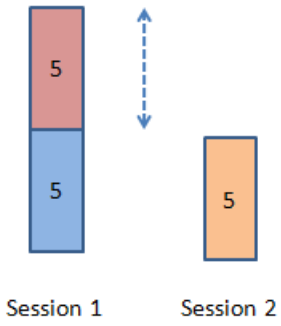

Figure 4.1: The deviation of two exercise sessions when organizing students in subgroups.

It is important to find a suitable strategy to split groups, limiting the drawback of grouping students in large units while still keeping the benefit of that strategy.

4.3.3 How to divide student groups

Students taking the same program often have the same timetable

We know that students in the same group can take different slots of the exercise part of a course. Therefore, if a group has many partial subscription exercises, it possibly has many arrangements for student schedules. For example, considering a group enrolls on n courses, each one has two exercise sessions and only requires students to take one of them. In theory, this group has 2^n combinations of time schedule. However, in practice, students registering for the same program are likely to take the same timetable so each group just has a few number of schedules.

Example 4.4. [1] Suppose 4 students sign up for the course LINMA1691, Discrete Mathematics. This course has two exercise sessions, but students need to attend only one. There are 2 students learning electricity (ELEC) and 2 students in physic (PHY) major. Each group registers for four other courses. The student schedule can be visualized as a graph. In this graph, course sessions are vertices and there is an edge between two vertices if there is at least one student taking both sessions. The course timetabling problem is reduced to the graph coloring problem.

If each group takes in one exercise session, then there are 10 edges for a group in the graph (Figure 4.2). Meanwhile, if we mix students of different groups and put one electricity student

[1] This example is inspired by the paper of Michael W. Carter [6]

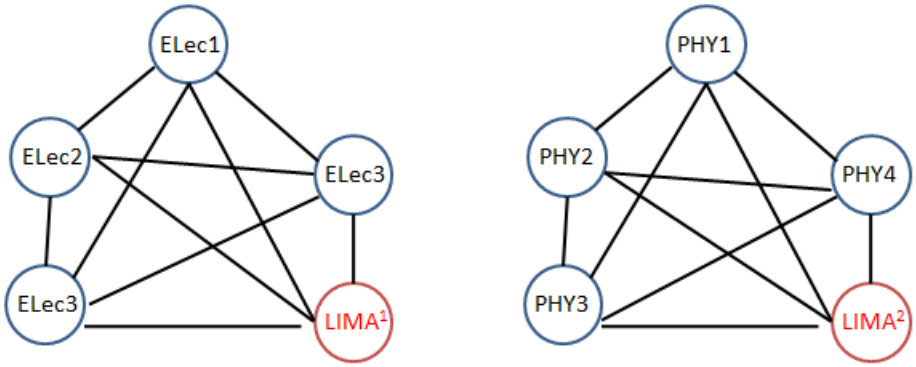

Figure 4.2: When each student group is assigned to separate exercise sessions, each group has 10 edges for their courses, then there the total edges is 20.

and one physic student in each session, then the number of edges doubles. When the number of edges in the graph is increased, it becomes much more difficult to find the conflict-free solution of graph coloring problem. It also means finding a solution to the course timetabling problem which has no conflict in the student schedule becomes more difficult.

In summary, students taking the same set of courses are likely to take the same timetable. Therefore, it is proposed that only large groups need to be divided to ensure the balance between exercise sessions, while students in small groups are kept together.

Proposal of division of student groups

Through the experiment, we suggest one way to divide large groups into smaller units. Overall, groups accounting for the largest percentage of each exercise session will be divided. The algorithm is presented in the upcoming section.

At the beginning, there is one subpart for each group (line 3). *NbSubgroups* is a map storing the number of subgroups of each group. For each course, the ratio of students in each group to the average number of students in an exercise session is computed (line 8). If the percentage is higher than 100 percent, the groups will be divided by 4 (line 9 -11). If the percentage is higher than 50 percent and less than 100 percent, then the group will be split into two subgroups (line 12 -14). The number of subgroups of each group is updated if its current value is smaller than the new one, *nbSubPart* (line 15 -17). After running this algorithm, in each exercise session, there is no subgroup accounting for more 50% in each exercise session. As a result, the arrangement of subgroups into different exercise sessions can help to reduce the deviation between them.

Notice that the decision of splitting a group depends solely on its percentage of its students on the average number of students in each exercise session. There is no limit set on the number of students of each group. For example, if group has more than 15 students then it needs to be divided into smaller units. The reason is that a group can account for a minor part in one course, but it can be the major group in other courses

Example 4.5. There are two courses, LMECA1901 and LELEC1530, in the program of the student group ME5el5. The average number of students per exercise session of each course is 68 and 45 respectively. There are 49 students in the student group ME5el5 and it accounts for 72% of the exercise session of LMECA1901 and 108% for a session of LELEC1530. The group

Algorithmus 1 Calculate the number of subgroups when splitting large groups

```
1: procedure CALCULATE–NUMBER–SUBGROUP
2:     for each group g ∈ Groups do
3:         NbSubgroups_g ← 1
4:     end for
5:     for each course c ∈ Courses do
6:         EnrolledGroup ← EnrolledGroupOfCourse(course)
7:         for each group g ∈ EnrolledGroup do
8:             Percent_{g,course} ← (NbStuPerGroup_g * 100) / AvgStudentPerExercise_course
9:             nbSubPart ← 1
10:            if Percent_{g,course} ≥ 100 then
11:                nbSubPart ← 4
12:            end if
13:            if Percent_{g,course} ≥ 50 AND Percent_{g,course} ≤ 100 then
14:                nbSubPart ← 2
15:            end if
16:            if nbSubPart > NbSubgroups_g then
17:                NbSubgroups_g ← nbSubPart
18:            end if
19:        end for
20:    end for
21: end procedure
```

should be divided by 2 in LMECA1901 and by 4 in LELEC1530 course. As a result, the number of subgroups is the maximum of two values and the student group will be split by 4.

	Avg. students	ME5el5 (49 students)
LMECA1901	68	72%
LELEC1530	45	108%

$\Longrightarrow$

	ME5el5 (# subGroups)
LMECA1901	2
LELEC1530	4

Table 4.3: Example of dividing a group into subgroups

4.4 Optimization techniques

4.4.1 Large Neighborhood Search

As mentioned in chapter 2, Constraint Programming is a complete method, which means it exhaustively explores the search space by using Branch and Bound Depth First Search. However, in hard Constraint Optimization problems, this method possibly experiences scalability issues. For example, a huge search tree cannot be explored entirely within a given duration and all the time is spent to explore a small region of the search space. Because of this, CP can be stuck in a small part of the search tree. This case is illustrated in the figure 4.3[2].

Large Neighborhood Search (LNS) is an incomplete method, which hybridizes Local Search (LS) and Constraint Programming (CP) to solve large-scale COPs. Its idea is that when the

[2]The figure is inspired from the note of Constrain Programming course [11]

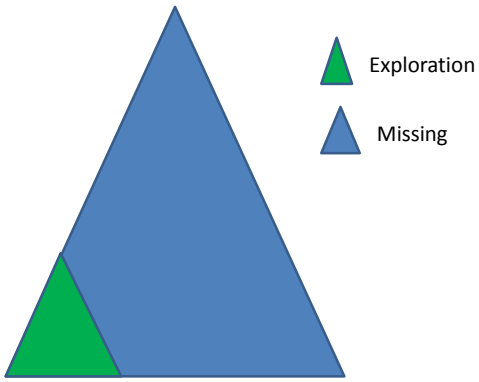

Figure 4.3: The search space of Constraint Programming.

search is stuck for a long time, it restarts and explores different regions of search search space. The process of LNS can be outlined in the following steps:

1. Find the first solution and consider it as the current best solution.
2. Generate the neighborhood by relaxing a fragment of the variables to their initial domain.
3. Explore the neighborhood by CP with a search limit.
4. If a solution is found, update it to the current best one. When the search limit is reached, restart and repeat step 2

One advantage of LNS is that it tackles the scalability issue of CP and improves the diversification by exploring different neighborhoods. Moreover, it is quite easy to implement when the user is provided a relaxation procedure and a search limit [3].

- Relaxation procedure defines the neighborhood around the current solution by relaxing a fragment of the variables. Other variables are fixed to their current value by adding temporary constraints to the problem.
- Search limit helps to avoid spending too much time on one region of search space. When the limit is reached, a restart is triggered and the search "jumps" to new a neighborhood. Its search space is similar to the figure 4.4 [4].

Standard LNS only optimizes one objective function. Meanwhile, there are five objectives in the course timetabling problem of EPL. These objectives can be optimized together by introducing a weighted sum function including all of them. However, the sum constraint in CP leads to weak filtering on lower bounds (based on [17]). Therefore, it was decided to optimize the objectives sequentially and their orders correspond to their priorities in the problem. In [18], Pierre Schaus presents Variable Objective Large Neighborhood Search which allows to change the objective functions along restarts.

[3] It is extracted from [18]
[4] The figure is inspired from the note of Constrain Programming course [11]

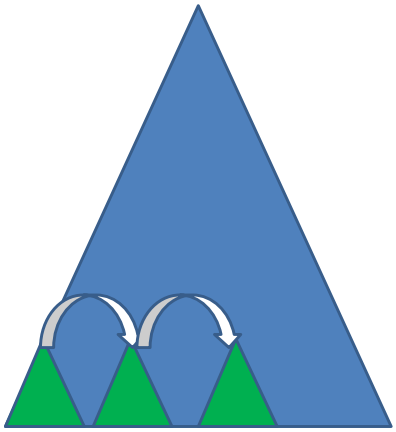

Figure 4.4: The search space of Large Neighborhood Search.

4.4.2 Variable Objective Large Neighborhood Search (VO-LNS)

Variable Objective Large Neighborhood Search is an extension of LNS which allows to optimize several objectives at the time and to change the objective function dynamically along the iterations. It has two benefits [18]:

- It provides a full control on the prioritization among the different objectives
- It allows stronger pruning during the branch and bound than using the weighted sum method.

The constrained optimization problem needs to minimize several objectives which can be included in a weighted sum function as follow:

$$\begin{array}{ll} Optimize & z = obj_1 + obj_2 + ... + obj_m \\ Subject\ to & constraints \end{array}$$

To turn this optimization model into VO-LNS, the objectives can be stated as the following equation.

$$\begin{array}{ll} Optimize & obj = \{obj_1, obj_2..., obj_m\} \\ Subject\ to & constraints \end{array}$$

The difference is that several objectives are stated in VO-LNS model instead only one in the weighted function. For each iteration, each objective is assigned to one out of three different filtering modes which affect the pruning of the search space [18]:

- *No-Filtering* deactivates the objective so it has no influence at all.
- *Weak-Filtering*: If a solution is found the bound of the objective is updated. The value of the objective in the next solutions have to be better or equal to its current value.
- *Strong-Filtering*: If a solution is found the bound of the objective is updated. The value of the objective in the next solutions have to be strictly greater than its current value.

Different configurations of the filtering modes can be set up to the objectives of course timetabling problem when they are optimized successively. For example, when minimizing the

first objective, it is assigned to *Strong-Filtering* mode while the others are set to *No-Filtering* (line 7 - 9). Once the value of the first objective reaches its minimum or the stop condition is met, the search moves to the next objective by changing the variable of the total conflicts in student schedule to *Weak-Filtering* mode and set the second objective to *Strong-Filtering* (line 15 - 17). These configurations do not allow the degradation the value of the first objective during the search.

```
1  // VO-LNS configuration
2  val allObjectives = Array[CPIntVar](objective1,
3                        objective2,objective3,objective4,objective5)
4  solver.minimize(allObjectives)
5
6  // minimizing the objective1
7  solver.obj(objective1).tightenMode = TightenType.StrongTighten
8  for(obj <- allObjectives; if obj != objective1)
9     solver.obj(obj).tightenMode = TightenType.NoTighten
10
11 ...
12 // Restarts
13 while (condition) {
14      lastSolutionStat = startSubjectTo(failureLimit = limitedBacktrack){
15       // change to the optimization of objective2
16       if(changeObjectiveCondition){
17          solver.obj(objective1).tightenMode = TightenType.WeakTighten
18          solver.obj(objective2).tightenMode = TightenType.StrongTighten
19       }
20    }
```

4.5 Optimization Objectives

Five objectives in the course timetabling of EPL are minimized sequentially. Through the experiment of optimizing all objectives, *dom/deg* variable heuristic was used. In this heuristic, the variable having the smallest ratio between its domain size and its number of constraints is be chosen first. The value assigned to a variable is selected randomly (except when optimizing the second objective).

4.5.1 Objective 1: Minimizing the total conflict in student schedule

The most important goal of course timetabling problem is to find a solution which minimizes the total conflict in student schedule. The schedule conflict of each subgroup is modeled by *softAllDifferent* in the 8th constraint 4.2.8. Based on this information, the total of conflict in the schedule of all students is calculated.

As mentioned in chapter 2, there are two approaches to solve constraint optimization problem. Both ways to optimize the first objective have been implemented and their results are presented in the next paragraphs.

Branch and Bound

In this approach, the *Oscar* solver is responsible for finding the optimality value of the objective which was declared. Every time the solver finds a solution with value v, it updates the constraint of objective variable to enforce the value of next solution strictly smaller than value v. The final solution is considered as the best one.

Relaxation

Branch and Bound method is combined with Large Neighborhood Search (LNS) to improve the diversification of the search and avoid the heavy tail problem. In LNS exploration, a relaxation strategy needs to be implemented in order to define the neighborhood when restarting a new search. Two different relaxation strategies has been implemented:

1. Relaxing all variables of courses of each program in which its subgroups has the timetable clash
2. Relaxing all variables of courses which overlap with the sessions of other courses having common students.

In order to distinguish between two relaxation strategies, let us consider the following example:

Example 4.6. Subgroup $EL4in4^1$ has one conflict in its schedule then the variables of courses in its program will be relaxed.

- The first strategy relaxes all variables of lecture and exercise sessions of 6 courses in the program of group $EL4in4$ to their initial domain. (These courses are listed in example 3.2).
- The second strategy detects courses which cause the conflict in the schedule. For example, the lecture sessions of two courses, *LSINF1252* and *LSINF1225*, occur at the same time, then only the variables of two courses will be relaxed

Experiments with two relaxation strategies have been carried on and the results are shown in table 4.4. In these tests, the student groups are split into subgroups.

	Semester 5 - 2013			Semester 4 and 6 - 2013	
#	Best value	Avg. Runtime		Best value	Avg. Runtime
Strategy 1	0	30 s		649	240 s
Strategy 2	0	22 s		583	240 s

Table 4.4: Comparison between two relaxation strategies.
Experiment with data of semester 5 in 2013.

The result of the second relaxation strategy dominates that of the first strategy. The reason is that when the total number of conflicts is large, most subgroups have the timetable clash. According to the first strategy, the variables of many courses need to be relaxed because they belong to a program in which its subgroups have the schedule conflict. As a result, finding the next optimal solution in the large neighborhood consumes much more time than relaxing only courses having conflicts in the second strategy.

Splitting student groups into smaller units

There are two ways of splitting a student group into smaller units: dividing them into individual students (atomic units) or subgroups (Algorithm 1). The following figure displays the result of the experiment implemented with data of semester 5 in 2013. It can be seen that the splitting student groups into subgroups obtains the optimal objective in 21 seconds. In contrast, the other approach could not yield the optimal result even after two minutes (Figure 4.5). The reason is that if a group is split into atomic units, students having different programs can be mixed in the same exercise session. As a result, there are many invalid combinations and the total conflicts in student schedule decreases slowly. The same experiment was done with data of

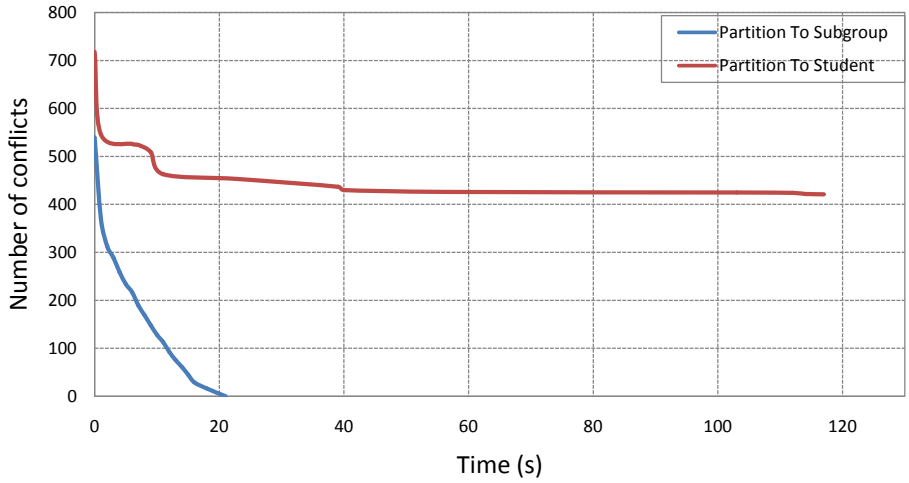

Figure 4.5: Comparison between splitting group into subgroups and students. Experiment with data of semester 5 in 2013.

semester 4 and 6 in 2013 and figure 4.6 illustrates its result. When the size of data increases, the branch and bound approach can not find the optimal objective in 4 minutes

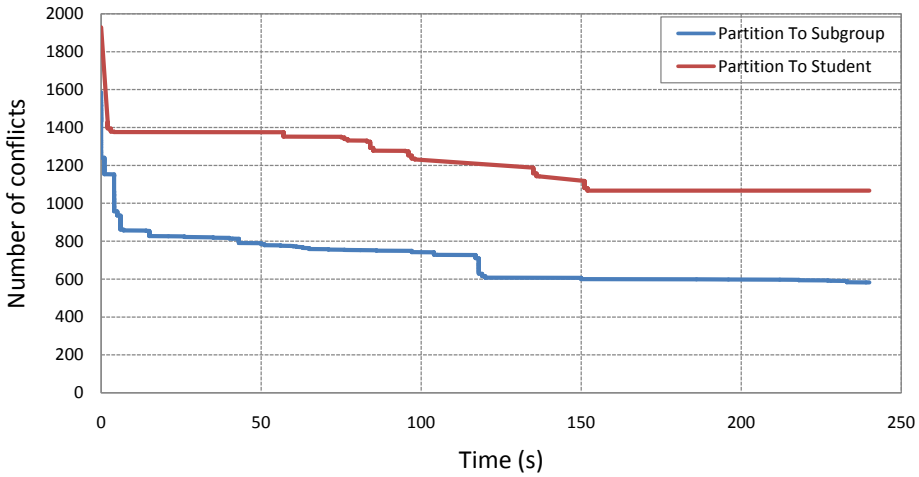

Figure 4.6: Comparison between splitting groups into individual students and subgroups. Experiment with data of semester 4 and 6 in 2013.

In summary, through these experiments, it can be seen that splitting groups into subgroups delivers better result and performance than splitting it into individuals. These results are suitable with the explanation and observation given in the section 4.3. Besides that, although this approach can find the optimal solution for semester 5 (S5), it was unable to identify the optimality when the size of the data increases (S4S6).

Iterative Optimization

Practice revealed that the total conflicts in student schedule for each data instance is quite small. Otherwise, the solution of timetable will not be accepted when it causes many clashes in the student schedule. Therefore, an iterative approach was tested, starting from the lower bound of the objective and gradually increasing it. For each search, a new constraint is added to enforce the total conflict is equal or less than the lower bound (line 7). Adding this constraint prunes the search space and removes parts not containing the optimal solution. If the search reaches the limit and solver has not found any solution, the lower bound increases by 1 (line 11 - 13).

```
1  // start from the lower bound
2  var lowerBound= totalConflictSchedule.min
3
4  // stop searching when finding one solution or the time execution runs out
5  while(!isFound && java.lang.System.currentTimeMillis() - startTime < maxTime){
6     val stat = startSubjectTo(nSols = 1, failureLimit = limitedBacktrack){
7        add(totalConflictSchedule <= lowerBound)
8     }
9     isFound = stat.nSols == 1
10    // increase the lower bound when not found the solution
11    if(!isFound){
12       nbRestart += 1
13       lowerBound += 1
14    }
15 }
```

The implementation was 100 times for each instance of data in 2013. On average, it took 0.8 s to find the first solution of semester 5 in 2013 (table 4.5) , while data of semester 4 and 6 in 2013 needs 3.2 s (table 4.6). From the result of the experiments, the total conflicts in student schedule is found faster by using iterative optimization method than by branch and bound approach. Even when the size of data increases, this method can solve the problem in reasonable time. However, in some tests, the iterative optimization approach can not find the best objective. For example, with the data of semester 5 in 2013, the optimal value is found only 56 times out of 100 runs. Meanwhile, the result of the semester 4 and 6 in 2013 is worse, only 37 times out of 100. The reason is that even the search space is reduced by adding the constraint to restrict the value of objective, it is still too large for an exhaustive exploration. Therefore, the search can stop when reaching the limit number of backtracks.

Found Value	Number times (out of 100 tests)
0	56
1	26
2	8
3	5
4	1
5	3
The Optimal value : 0	
Average time: 0.8 s	

Table 4.5: Result of optimization semester 5 in 2013 by using iterative method

Found Value	Number times (out of 100 tests)
54	37
55	25
56	11
57	10
58	8
59	2
60	1
62	2
65	2
71	1
72	1
The Optimal value : 54	
Average time: 3.2 s	

Table 4.6: Result of optimization semester 4 and 6 in 2013 by using iterative method

Combining both approaches

Through the analysis above, one sees that the advantage of iterative optimization is finding the solution in short time even with large-scale data. Meanwhile, branch and bound approach can find the optimal objective. Therefore, we suggest to use both methods to leverage their strong points. First, iterative optimization is used to find the first solution. The experiment was run 100 times with the data of 2013 and its average result is displayed in table 4.7. Clearly, the found value is quite close to the optimality. The average value of objective in semester 5 is 0.89 while it is 56.05 in semester 4 and 6. Notice that, the runtime of first step includes the initialization of variables in the model.

	Semester 5		Semester 4 and 6
Runtime	0.90 s		3.8 s
The optimal value	0		54
Avg. Value of objective	0.89		56.05

Table 4.7: The average result of first solution found by iterative optimization. Testing with data of 2013.

After having the first solution, the optimal objective is searched by branch and bound approach with Large Neighborhood search (LNS). The experiment was run with two relaxation strategies which are described in Branch and Bound approach (section 4.5.1).

1. Relaxing all variables of courses of each program in which its subgroups has the timetable clash
2. Relaxing all variables of courses which overlap with the sessions of other courses having common students.

Branch and bound approach can find the optimal objectives with both strategies (The total conflict is 0 in semester 5 and it is 54 with S4S6). However, the average of execution time of the

	Semester 5 - 2013			Semester 4 and 6 - 2013	
#	Best value	Avg. Runtime step 2		Best value	Avg. Runtime step 2
Strategy 1	0	68 ms		54	3250 ms
Strategy 2	0	51 ms		54	2490 ms

Table 4.8: Comparison branch and bound approach with two different exploration strategies

second relaxation strategy is also shorter than the first one. Relaxing only courses which take the same slots with other courses having common students reduces the neighborhood and also gives the flexibility to re-optimize the current solution.

The search moves to the next objective iff two conditions are met: either the current solution is equal to the lower bound of the objective variable or the duration from the last solution reaches the limit. The found objective can be non-optimal if the stop condition falls into the second case. However, through the experiment with all instances of data (S5 in 2011 - 2013 and S4S6 in 2013) the best objective is always found.

4.5.2 Objective 2: Minimizing the conflicts in non-preferred slots of professors

Generate non-preferred slots of each professor

The second objective of the problem is to minimize the conflicts in non-preferred slots of professors. However, there is no information about the non-preferred slots of each professor in the data of three years 2011, 2012 and 2013. Therefore, an instance generator was created to produce artificial data for testing purpose. The non-preferred slots of each professor are generated such that do not overlap his unavailable slots. The number of non-preferred slots depends on the given percentage.

Example 4.7. Professor A has 3 unavailable slots out of 20 slots. Thus, he can teach in 17 left slots. If the percentage of non-preferred slots is 30%, then the generator will create 5 slots at which the professor prefers to avoid teaching (17 * 0.3 = 5.1).

Calculate the conflicts in non-preferred slots

To compute the number of conflicts in non-preferred slots of each professor, each of his teaching session is checked whether it is in his unfavored slots (line 5-6). After that, the maximum conflicts of a professor and the total conflicts of non preferred slots of all professors are computed.

```
1  /* Calculate the conflict in non-preferred slots of each professor */
2  for(session <- professorSchedule){
3     ...
4     /* check whether the session is in non-preferred slots */
5     val isInNonPreferredSlot = sum(0 until nonPreferredSlots.size)
6                                    (i => session === nonPreferredSlot(i)))
7     ...
8  }
```

The solution requires to ensure the fairness between professors so the total conflicts in unfavored slots should not compress on one person. Therefore, the following priority was set between objectives:

1. The maximum number of conflicts in non preferred slots of a professor (subObjective1)
2. The total conflicts in non preferred slots of all professors are computed (subObjective2)

This requirement can be modeled by a weighted sum function. The formulation needs to ensure that the solution with smaller value of subObjective1 is considered better than the other solution. Therefore, the value of subObjective1 is multiplied with the upper bound of the subObjective2.

```
// weighted sum
val objective2 = subObjective1 * UBsubObjective2+ subObjective2
minimize(objective2)
```

Search

Besides using the same variable heuristic like other parts, a specific value heuristic for this objective was devised. For each teaching session, an array containing the number of conflicts in non-preferred slots is created. When choosing the value of the variable of each teaching session, a value which is in its domain and has the smallest number of conflicts is chosen first. If there are many values having the same number of conflicts, one of them is chosen randomly.

Example 4.8. The lecture session of course A has the array of number of conflicts in non-preferred slots like the table 4.8. The size of the array is equal to the number of slots per week (*nbSlots*). This array is computed before the search beginning.

The value of the fourth element in the array $NbConflict$ is 2. It means that if this lecture session is assigned to slot 4, it causes two conflicts in non-preferred slots of professors. Assume the current domain of variable of this lecture sessions has three values (1, 7, 4). Slots 1 and 7 have the lowest number of conflicts (1 conflict) and one of them is chosen randomly.

NbConflict	0	1	2	0	2	0	1	1	...	0
Index	0	1	2	3	4	5	6	7	...	19

Table 4.9: An example of the array of number of conflicts in non-preferred slots of a lecture session.
Based on this array, a value is chosen for its variable.

With each given percentage, the experiment is run 10 times and its average result is illustrated in table 4.10. With both heuristics, the maximum number of conflicts of a professor obtains the optimal result. The score of $objective2$ is better with the specific value heuristic than with the random, especially when the percentage of non-preferred slots increases. However, in some tests, the optimal objective is not found The reason is that the search moves to the next objective if the duration from the last solution exceeds the time limit.

	Random value heuristic			Specific value heuristic			Optimal Result	
Percentage	Max Con.	Objective2		Max Con.	Objective2		Max Con.	Objective2
10	1	40		1	40		1	40
20	0	0		0	0		0	0
30	1	40		1	40		1	40
40	2	87.1		2	83		2	83
50	2	84		2	84		2	84
60	1	45		1	45		1	45
70	2	93.6		2	91.6		2	91
80	2	99.7		2	96.2		2	96
90	2	103.4		2	100		2	100
100	3	156		3	156		3	156

Table 4.10: Comparing the results of two heuristics with the optimal value.
Testing with data of semester 5 in 2013

Relaxation

If a lecture session or exercise session occurs in non-preferred slots of their responsible professors, then the variable of this session is relaxed. Meanwhile, a portion of teaching sessions having no conflict in non-preferred slots is fixed.

4.5.3 Objective 3: Maximizing the quality of schedule

Definition of the third objective

In the equation computing the quality of schedule (equation 3.22), the denominator is constant. Therefore, the maximization the quality of schedule is reduced to the maximization of the weighted function ($valueLecture * 2 + valueExercise$). It is presented in dual way by minimizing the negative value of the weighted function. Therefore, the third objective function is defined like the following code snippet:

```
1  // objective3 : Minimizing the negative value of lecture and exercise sessions
2  val objective3 = - valueLecture * 2 - valueExercise
```

Relaxation

If a teaching session is scheduled in the low quality slots (smaller than one), then its variable is relaxed. Meanwhile, a portion of lecture and exercise sessions occurring in the middle of day will be fixed to the schedule of the current solution.

Result

The experiments with the data of semester 5 from 2011 to 2013 were run and their results are displayed in figure 4.7. The quality of schedule in 2012 and 2013 takes a long time to improve and their results are 81.6% in 2012 and 82.9% in 2013. In contrast, the optimization of *objective2*

with the data of semester 5 in 2011 takes a shorter time (about 20s) to reach the best value (97%).

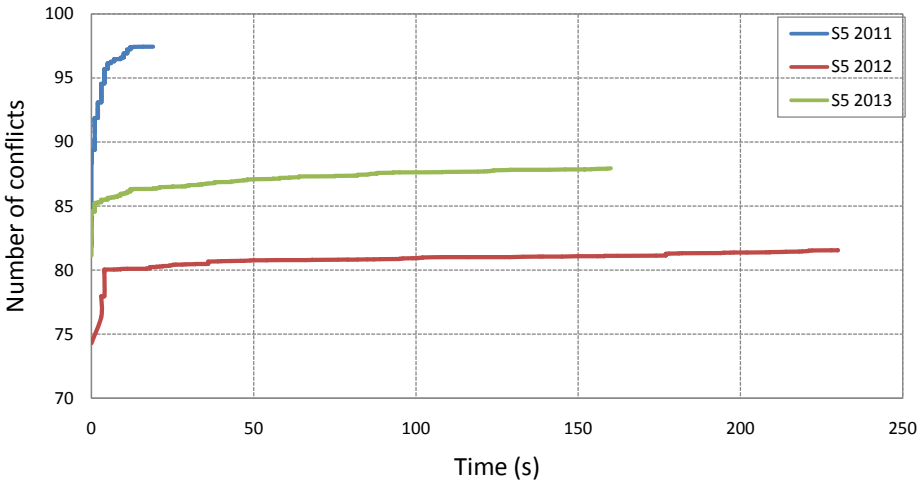

Figure 4.7: The quality of schedule in semester 5 from 2011 to 2013.

4.5.4 Objective 4: Minimizing the total number of students attending the end slot of day

Definition of the fourth objective

As mentioned in the third chapter, the minimization of the total number of students attending the end slot of day is equivalent to the minimization of the expression $(valueLectureEndSlots * 2 + valueExerciseEndSlots)$. The following code snippet demonstrates the function of the fourth objective.

```
1  // objective4 : Minimizing the value of lecture and
2  //              exercise sessions in the end slots
3  val objective4 = valueLectureEndSlots * 2 + valueExerciseEndSlots
```

From the equation 3.25, the quality of the schedule in the end slots is calculated easily by dividing the value of $objective4$ by the value of the expression $totStudentAllLecture * 2 + totStudentAllExercise$.

Relaxation

If a teaching session is scheduled in the end slot of day, then its variable is relaxed. Meanwhile, a portion of lecture and exercise sessions not occurring in the end slot of day will be fixed to the schedule of the current solution.

The reason to introduce the fourth objective

Clearly, the end slot of day has the lowest value in the *qualitySlots* array so the minimization of the total number of students attending the end slot of day leads to the increase in the quality of schedule. It can be seen that the fourth objective has the same goal of improving the quality of schedule. An interesting question is that why an extra objective is introduced while it is included in the third one. The first reason is that by using LNS, users have no proof of optimality for the found value of the objective because the search stops when the search limit is reached. For example, the third objective moves to the next goal when the duration from the last solution to the current exceeds the given time. In addition, when focusing on the end slot of day, only variables of teaching sessions occurring in the last time of day will be relaxed. As a result, this relaxation strategy defines a smaller neighborhood in comparison with the strategy when maximizing the quality of schedule. It is expected to find a better solution when increasing the intensification of the search. In fact, the quality of schedule still increases when minimizing of the total number of students in the end slots. The results are demonstrated in the table 4.11

	Result of optimizing objective 3			Result of optimizing objective 4	
Year	Quality of schedule	Quality of end slots		Quality of schedule	Quality of end slots
2011	96.7%	1.62%		97.04%	1.53%
2012	82.2%	19.6%		82.28%	19.33%
2013	87.6 %	14.8 %		88.7%	12.9%

Table 4.11: The improvement of the quality of schedule after optimizing objective 4

It can be seen that the quality of schedule is improved after optimizing the fourth objective in two years 2011 and 2013, while it does not change much in 2012.

4.5.5 Objective 5: Minimizing the total deviation of number of students in exercise sessions

One of requirements of the problem is to arrange students in exercise sessions equally. However, this requirement is often too strong for realistic instances so it is relaxed. Therefore, the last objective is to minimize the total deviation of number of students in exercise sessions.

The number of students in exercise sessions is restricted by the *binPacking* constraint (In section 4.2.9). In this constraint, the load of each bin is also the number of students in each exercise session. From this, the deviation of number of students in each session with the average number of students is computed (line 5-6). The total deviation is the sum of the deviation in all exercise sessions.

```
1  // binPacking restricts the minimum number of students in exercise sessions of a
       course
2  add(binPacking(varOfExerciseSubGroup, nbStuInSubGroup, loads))
3  ...
4  // the deviation of exercise sessions of a course
5  val diffOfSessions = loads.map { x =>
6                              absolute(x - avgStuPerExerciseSession(course))}
```

Relaxation

The schedule of lecture and exercise sessions is fixed to the current solution. Meanwhile, if the ratio of the deviation of real number of students attending each session to the average number of students of this session is greater than *Margin* percent, then the variables of index of subgroups in this exercise are relaxed.

To recall, the ratio of the deviation of number of students attending each session to the average number of students of this session is computed as the following equation.

$$\begin{aligned} &\forall course \in PartialSubscriptionExercise \\ &\quad \forall session \in ExerciseSession_{course} : \\ &\qquad ratioDeviationStudentInSession_{session} = \frac{NbStudentInExerciseSession_{session}}{AvgStudentPerExercise_{course}} \end{aligned} \tag{4.1}$$

Result

Table 4.12 illustrates the results of the experiments with data of semester 5 in three years 2011 to 2013. In the experiments, *Margin* parameter is set to 0.8 and *RelaxMargin* is set to a smaller value, 0.7. In general, the percentage of deviation of number students in exercise sessions (computed by equation 3.27) is smaller than 20%.

Year	The total deviation	Percentage of deviation
2011	49	9%
2012	294	15%
2013	91	10%

Table 4.12: The total deviation of number of students in exercise sessions

4.5.6 Relaxation

The LNS technique is used to optimize objective functions and there are several relaxation procedures implemented to define the neighborhood

Worst Solution Relaxation

When each objective is optimized, a specific relaxation procedure is implemented in which the variables preventing the improvement of objective function are relaxed.

- Optimizing objective 1: If a teaching session of a course overlaps the sessions of other courses having common students, then its variable is relaxed.
- Optimizing objective 2: If a lecture session or exercise session occurs in non-preferred slots of their responsible professors, then the variable of this session is relaxed.
- Optimizing objective 3: If a teaching session is scheduled in low-quality slots (the quality value of this slot is smaller than one), then its variable is relaxed.

- Optimizing objective 4: If a teaching session is scheduled in the last slot of day, then its variable is relaxed.
- Optimizing objective 5: If the ratio of the deviation of number of students in a session is greater than *Margin* percent, then the variables of index of subgroups in this exercise are relaxed.

By relaxing variables that prevent the objective improvement, the probability of finding better solution increases. Notice that, when optimizing objectives 1 to 4, if the variables of exercise sessions of a course are fixed to the value of the current solution, then the arrangement of subgroups in these exercise sessions does not change. The fixing subgroups in these exercise sessions decreases the time to find the next solution. This is because the arrangement of subgroups in each exercise sessions must not increase the total of conflicts in student schedule as well as to reach the minimum of number students attending each exercise. It is difficult to satisfy this requirement when optimizing the quality of schedule because teaching sessions are scheduled in a few slots having the high quality. Consequently, some exercise sessions of courses in the same program can occur in the same slot, so the search needs to find the exact way to arrange subgroups. One case of this situation is demonstrated in example 4.2.

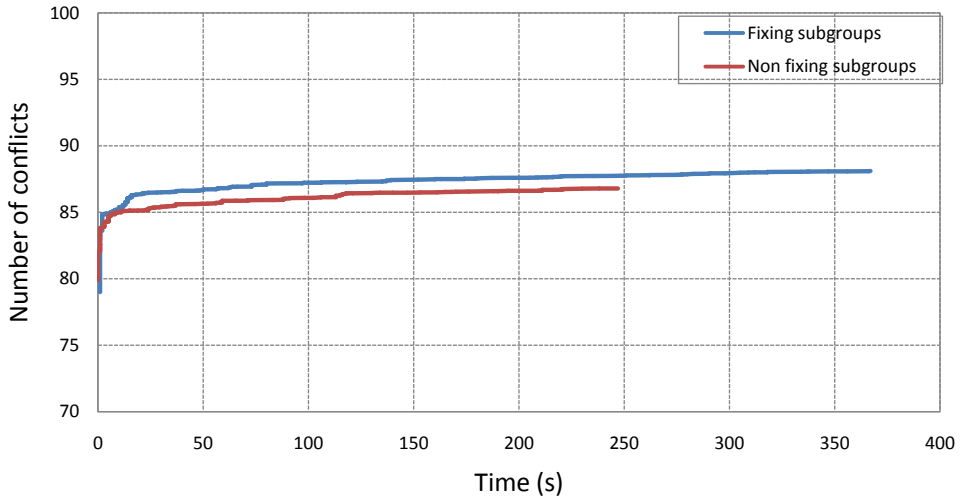

Figure 4.8: Comparing the improvement of the quality of schedule when fixing the arrangement of subgroups and relaxing them. Testing with data of semester 5 in 2013.

The figure 4.8 illustrates the results of the quality of schedule improvement when fixing the arrangement of subgroups and relaxing them. Clearly, the quality of schedule gets better result when fixing the arrangement of subgroups.

After relaxing the variables of a specific set of courses which prevent the improvement of the objective function, 60% of the remaining variables will be fixed to their value of the current solution. The 60% is chosen randomly. Relaxing random variables is important especially in case of restarts without solution.

Adaptive Neighborhood Relaxation

Besides relaxing random variables, LNS is extended with adaptive neighborhood to make the exploration vary in restarts. This approach uses the information collected during the search to

	Result
Total conflict in schedule	5
Quality Schedule	78.2%
Quality in the end slots	24.6%
Percentage of deviation	12%

Table 4.13: Value of objectives when using dom heuristic.
Testing with the data of semester 5 in 2013

define the neighborhood of new restarts. If no solution is found during a search, the neighborhood should be enlarged by increasing the fragment of relaxation. If the search is completed, the neighborhood should be much more enlarged. Finally, if at least one solution is found, the neighborhood should be narrow. After relaxing all the variables which prevent the improvement of objective function, the portion of the remaining variables are fixed in a new restart. This portion is modified as the following:

- In initialization, 60% of the variables are fixed.
- If no solution is found, it decreases by 3%
- If the neighborhood is fully explored, it decreases by 10%
- If a solution is found, the portion is set back to 60%

In summary, random relaxation and adaptive neighborhood relaxation improve the diversification the search.

4.5.7 Search

Relaxation procedure defines the neighborhood around the current solution by relaxing a fragment of the variables. The search is responsible for exploring the neighborhood and finding better solutions. A good variable and value heuristics have a large impact on tree size so choosing suitable heuristics can help to find a solution in fewer number of nodes. Two variable heuristics have been experimented.

Binary First Fail - Dom heuristic

Its idea is to select a variable with the smallest domain first. It is a dynamic heuristic which respects the first-fail principle [5]. Table 4.13 displays the value of objectives when using dom heuristic. The value of total conflicts in student schedule is 5 while the optimal value is 0. In our model, the variables of index of subgroups in exercise sessions have the smallest domain which is the number of exercise sessions of the corresponding course (usually is 2). Therefore, these variables are chosen first. However, arranging subgroups into exercise sessions before scheduling teaching sessions causes many conflicts in the student schedule.

[5]The is inspired from the note of Constrain Programming course [11]

Binary First Fail - Dom/deg heuristic

The variable having the smallest ratio of its domain size over its number of constraints is selected first. It also respects the first-fail principle. This variable heuristic is used for all the experiments in this chapter. It successes in finding the optimal of value of the total conflicts in student schedule. Other objectives are quite close to the optimal values.

In summary, this chapter describes the implementation of the model of the course timetabling problem in *Oscar* solver. When optimizing objectives, several optimization techniques are experimented and presented.

Chapter 5

The experiments

Although several tests have been reported in the previous parts, this chapter offers the synthesis results of the experiments with the data of course timetabling problem of EPL.

5.1 Data size

The experiments have been run with the data of semester 5 of three years from 2011 to 2013 and semester 4 and 6 in 2013. The detail of each attribute of data is presented in the table 5.1. The size of data of semester 4 and 6 in 2013 is the largest because two semesters are scheduled together. On average, each group has approximately 12 sessions out of 20 slots per week in all instances excepting the data of semester 5 in 2011.

#	S5			S4S6
	2011	2012	2013	2013
Courses	22	22	21	39
Lecture Sessions	24	43	39	63
Exercise Sessions	22	50	35	72
Professors	35	29	32	52
Students	263	362	322	644
Groups	41	51	44	88
Avg. sessions per group	9.41	12.51	11.66	11.55

Table 5.1: The attributes of data instances

5.2 Results

The solution found by the CP model is checked in the MIP model to verify whether it satisfies all constraints. Through the experiments, two models have different values in the total conflicts in student schedule with the same timetable schedule. The reason is that in the CP model, the number of students is converted to an integer number. As a result, some groups which have the small number of students are rounded down to zero. For example, two groups *GBe2* and *GCe2* had 0.4 students in semester 5 in 2012. However, the numbers of students of two groups are

rounded down to zero in CP model (table 5.2). When the total conflicts in student schedule are computed by the equation 3.16, the conflicts in the schedule of two groups are ignored. In CP model, the solution of semester 5 in 2012 has no conflicts in student schedule while it gets 0.8 conflict in the MIP model.

	Estimated number of student (In MIP model)	Rounded number of student (In CP model)
GBe2	0.4	0
GCe2	0.4	0

Table 5.2: The difference in the number of students in two models.

In order to consider groups with a few students, the numbers of students of all groups are multiplied with a ratio (e.g 10 or 100). Only *objective1* is affected by this changes. The total of conflicts in the student schedule is calculated by divided the value of *objective1* by the ratio. In my tests, the ratio is 100 and the results of experiments are illustrated in two tables 5.3 and 5.4.

	S5 2011		S5 2012	
	CP result	The optimal value	CP result	The optimal value
Total conflict in schedule	0	0	0	0
Quality of schedule	96.55%	97.45%	82.12%	82.28%
Quality in the end slots	1.54%	1.51%	19.6%	19.1%
Percentage of deviation	8%	-	14%	-

Table 5.3: Result of experiments with the data of semester 5 in 2011 and 2012

	S5 2013		S4S6 2013	
	CP result	The optimal value	CP result	The optimal value
Total conflict in schedule	0	0	53.5	53.5
Quality of schedule	88.5%	91.8%	84.08%	90.04%
Quality in the end slots	10%	8.26%	15.82%	8.7%
Percentage of deviation	8%	-	12%	-

Table 5.4: Result of experiments with the data of 2013

Clearly, the total conflicts in student schedule reach the optimal value with all data instances. The quality of schedule is quite close to the optimal value. Notice that, in the real data of EPL in 2011 to 2013, there is no information about the non-preferred slots of professors so the second objective is always equal to zero. In the MIP model, the numbers of students in exercise sessions of the same course are strictly equal because these number students can get real numbers. In contrast, the CP model only allows the number of students of each exercise session to be an integer. Therefore, it does not make sense to compare the deviation of students in exercise sessions between two models.

In summary, this chapter presents the result of the experiment in the CP model which is quite close to the optimal value. The difference of type format of the CP model and MIP model leads to the difference of the results of two models and a simple trick was introduced to reduce the gap.

Conclusion

This master thesis presents a tool to schedule course timetabling problem of EPL automatically. The application has succeeded in finding a solution which optimizes all the required objectives in a convenient time.

In the modeling process, we also proposed an algorithm to split student groups into small units. The division of groups into subgroups improves the balance when arranging students in exercise sessions of the same course. Meanwhile, it also reduces the execution time of finding solutions in comparison with splitting groups into individual students. When optimizing total conflicts in student schedule, several techniques are implemented such as Branch and Bound and Iterative Optimization and different relaxation procedures are also experimented. As a result, the performance of the model is improved and the optimal value of total conflicts in student schedule can be found in a few seconds even when the data size increases (section 4.5.1).

When modeling the course timetabling problem of EPL, we have considered all stakeholders and defined the priority of their goals. These objectives are optimize successively by using different optimization techniques such as Large Neighborhood Search (LNS) and Variable Objective Large Neighborhood Search (VO-LNS). As a result, a high-quality timetable is found within a reasonable time.

This master thesis presents an additional approach to solve the course timetabling problem of EPL besides the current MIP model. In doing so, we are able to present and analyze the differences between CP and MIP with regards to the formulation and modeling of the given constrained optimization problem.

Bibliography

[1] Wikipedia. *Timetable . https : //en.wikipedia.org/wiki/Timetable_(disambiguation)*

[2] Andrea Schaerf. *A Survey of Automated Timetabling.* 1999. ARTIFICIAL INTELLIGENCE REVIEW. pages 87–127

[3] Barry McCollum *University Timetabling: Bridging the Gap between Research and Practice.* Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling. pages 15–35. 2006

[4] UCL. *Major and minor of bachelor at UCL. http://www.uclouvain.be/cps/ucl/doc/fsa/documents/FSA1BA-Maj-min.pdf*

[5] Edmund Burke, Kirk Jackson, Jeff Kingston, Rupert Weare *Automated University Timetabling: The State of the Art* 1997. *http : //comjnl.oxfordjournals.org/content/40/9/565.short*

[6] Michael W. Carter *A Comprehensive Course Timetabling and Student Scheduling System at the University of Waterloo.* Practice and Theory of Automated Timetabling III: Third International Conference, PATAT 2000.

[7] D. de Werra *An introduction to timetabling.* Euro J. Oper Res. pages 151 - 162. 1985. *http://www.cs.uu.nl/docs/vakken/stt/De-Werra.pdf*

[8] Roman Barták. *Constraint Programming: In Pursuit of the Holy Grail.* In Proceedings of the Week of Doctoral Students. pages 555 - 564. 1999

[9] Régin, Jean-Charles. *Generalized arc consistency for global cardinality constraint.* Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1. pages 209–215. 1996.

[10] Demeulenaere, Jordan and Hartert, Renaud and Lecoutre, Christophe and Perez, Guillaume and Perron, Laurent and Régin, Jean-Charles and Schaus, Pierre. *Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets.* CP2016. 2016.

[11] JB Mairy and Yves Deville. *The Slide of course Lingi2365: Constraint programming. https : //moodleucl.uclouvain.be/mod/page/view.php?id* = 539874

[12] Christian Bessière, Pascal Van Hentenryck. *To Be or Not to Be ... a Global Constraint. In Principles and Practice of Constraint Programming – CP 2003* pages 789-794

[13] Pierre Schaus, Jean-Charles Regin. *Cardinality Reasoning for Bin-Packing Constraint: Application to a Tank Allocation Problem.* Principles and Practice of Constraint Programming. pages 815-822.

[14] Cambazard, Hadrien and Hebrard, Emmanuel and O'Sullivan, Barry and Papadopoulos, Alexandre. *Local search and constraint programming for the post enrolment-based course timetabling problem.* Annals of Operations Research. pages 111–135. 2012.

[15] Francois Glineur. *MIP Model to schedule Bachelor Program of EPL*. 2013.

[16] Cambazard H, Hebrard E, O'Sullivan B, Papadopoulos A *Local Search and Constraint Programming for the Post Enrolment-based Course Timetabling Problem*. Journal: Annals of Operations Research, pages 111 - 135. 2012.

[17] J.C. Regin and T. Petit. *The objective sum constraint*. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pages 190–195, 2011.

[18] 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.

[19] PATAT Conference. *http://www.patatconference.org/*

[20] OscaR Team. OscaR: Scala in OR, 2012.
Available from *https://bitbucket.org/oscarlib/oscar*.

[21] Petit, Thierry and Régin, Jean-Charles and Bessiere, Christian. *Specific filtering algorithms for over-constrained problems*. International Conference on Principles and Practice of Constraint Programming. pages 451–463. 2001.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**