AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

FACULTY OF COMPUTER SCIENCE, ELECTRONICS AND TELECOMMUNICATIONS

DEPARTMENT OF COMPUTER SCIENCE

Master of Science Thesis

Course Scheduling System for Students

Author: *Adam Furmanek*
Degree programme: *Computer Science*
Supervisor: *Piotr Faliszewski PhD, DSc*

Kraków, 2015

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIA INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Praca dyplomowa magisterska

System przydziału zajęć dla studentów

Autor: *Adam Furmanek*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr hab. inż. Piotr Faliszewski*

Kraków, 2015

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the thesis itself.

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i że nie korzystałem ze źródeł innych niż wymienione w pracy.

I would like to thank my promoter Piotr Faliszewski for his overall help with my work on this thesis.

Contents

1. Introduction

The goal of this thesis is to describe some possible ways of using Integer Linear Programming in solving a Students Placement Problem. We formulate logic operators, mathematical functions, and a sorting procedure in terms of ILP constraints. We then use them to formulate and solve the problem of assigning students to the classes on a typical university. Finally, we study the performance of the implemented system and compare different ILP solvers.

1.1. Students Placement Problem

Before each semester, the students of Computer Science at AGH University of Science and Technology in Cracow, Poland need to select classes they want to attend. Every student can declare subjects that he or she is interested in and wants to study during the next semester. Based on these preferences, the authorities of the Department of Computer Science create the timetable of the classes. Such a timetable contains a number of classes for each course, the upper limit of people allowed to attend each particular class and the assigned lecturers.

This creates a way for the students to select the classes they want most. They are not assigned by the authorities of the University to particular classes, and so they can try to find a schedule which allows them to fulfill both their external duties and their school obligations. For instance, they can try to select the classes occurring in the evening so they can work during the day. Or, they can select classes happening after the lunch break so they can sleep long. We can come out with many other ideas because each student has different duties and therefore might want to attend a different set of classes.

Unfortunately, there are limitations. Every class has a well specified limit of people that can attend that class, so there might be situations when there are too many people willing to attend a given class. When it happens, some of the students must select other classes in order to be able to finish the course. Since most of the students have similar needs, there is a set of classes which are very highly desired. Therefore, there is a need to decide which students are allowed to attend which particular classes when there are more candidates than places. We need to have a method to create an assignment allowing the students to attend the courses and, on the other hand, that respects their preferences as much as possible.

An obvious method is to assign students manually. We can gather students' preferences and ask someone (for instance, the leader of the student representative body) to create the assignment. This method has some clear drawbacks. Depending on the classes' limits, it might be very slow. It might be

unfair, because it is unclear how to assign students in a fair way. Finally, it might not work when there are many students and tight bounds. On the other hand, this method can be very efficient in some cases and is often used in practice.

We can see that a fair method of creating the assignments would be very useful. It would allow the students to utilize their time in a most efficient way and to attend as many external activities as possible. It would ease their work and increase their happiness. It is not surprising that there are many research works that regard the methods of finding such assignment. In this thesis we present the system for solving the Students Placement Problem using Integer Linear Programming that was used at AGH University of Science and Technology in years 2013-2014.

1.2. Review of similar problems and state-of-the-art

Many researchers were tackling different types of assignment problems. One of the best known assignment problems is the timetabling problem in which we need to assign lecturers to courses to classes to rooms. Such problems need to be solved every year on many universities.

There are many approaches to solving the timetabling problem using mathematical programing. For instance, Lawrie [17] presents the generation of so-called "layouts" for different groups of students and then attempts to arrange them in an optimal way. On the other hand, Akkoyunlu [1] solves an assignment problem between a set of courses and the time periods, focusing on conflicts in assignment. De Werra shows that the university timetabling problem is NP-complete and so chooses to use the graph theory approach to solve the problem [10]. Tripathy presents an extension to his work [23], focusing on the case of conflict matrices. Tripathy also presents a heuristic approach for the problem [22].

Breslaw shows a solution for the faculty assignment problem [6], a problem similar to the timetabling problem. The same problem was studied by McClure and Wells [18] and the solution was attempted with the help of mathematical programming. Hultberg and Cardoso [15] formulate the teachers assignment problem as an MIP problem and solve it as a special case of the fixed charge transportation problem. Badri, Davis, Davis and Hollingsworth solve the teacher assignment problem using goal programming [4]. Gosselin and Truchon present linear programming formulation for the classroom allocation problem [13].

There are also successful approaches using techniques for combinatorial problems. Among these, Costa used tabu search [8], whereas Kang and White used a constraint logic programming [16]. Genetic algorithms have also been used in the process of solving assignment problems [19][12].

These assignment problems are quite different than the problem presented in this thesis. They focus on creating the timetable for university, not on assigning students to maximize their happiness. There are no recent research works that regard the Students Placement Problem.

1.3. Organization of the Thesis

Let us now describe the organization of this thesis. In Chapter 2, we present the theory of Integer Linear Programming and algorithms commonly used for finding solutions. We review the known result that solving ILPs is an NP-complete problem and cannot be solved using algorithms working in polynomial time (when $P = NP$).

In Chapter 3, we present the possible ways of using ILPs. We formulate the methods of calculating logical operators and mathematical functions, and we provide a program for sorting numbers. All these elements can be used to describe more sophisticated problems, and, indeed, we use these building blocks to formulate our solution for the Students Placement Problem.

In Chapter 4, we formally define the Students Placement Problem and formulate it using ILP. This is a real-life problem which appears on many universities and needs to be solved in some way. We consider practical aspects of the problem, necessary requirements, and possible consequences of adopted solutions.

In Chapter 5, we describe our actual implementation. We first present elements of the system and the real-life data used during the system's lifetime. We then present results of the tests performed to analyse the usefulness of the implementation.

In Chapter 6, we discuss some possible improvements and elements which can be implemented to increase the power of the system.

In the last chapter we present our conclusions.

2. Necessary Preliminaries of Integer Linear Programming

In this chapter we describe the necessary preliminaries regarding Integer Linear Programming (ILP) and its usage in solving optimization problems. We also show the famous proof that ILP is NP-complete, and discuss some algorithms for solving ILP.

2.1. Decision Problems

A decision problem is a question which can be answered either with a "yes" or a "no", depending on the particular input. It is traditionally defined as the set of all possible inputs and the set of inputs for which the answer is "yes". The inputs are often strings over some finite set of symbols, and the subset of strings, for which the problem returns "yes" is a formal language. Thus it is typical to view decision problems as formal languages of the strings for which the answer is "yes". Formally, decision problems are defined as follows:

Definition 1. *Decision problem $\pi = (D_\pi, Y_\pi)$ consists of the inputs of the problem D_π and the inputs of the problem Y_π for which the answer is "yes".*

One of the most popular examples of a decision problem is the primality test—for any non-negative integer we ask whether the number is prime or not. In other words, we ask whether such a number has any divisor apart from one and itself. Decision problems are related to function problems, in which for a given input we expect an output that is possibly more complex than a simple "yes" or "no".
Decision problems are also related to optimization problems, which seek the best answers to a particular problem. The knapsack problem is one such problem, in which for a given set of items (each with a mass and a value) we need to find the set of items with the greatest value, but with the mass not exceeding some given bound. Such problems are called optimization problems because there are usually many feasible solutions (solutions complying with specified requirements), of which we need to select the best one. Mathematical optimization is often called mathematical programming.

2.2. Programming in Mathematics

An optimization problem is usually a function, which needs to be maximized or minimized within a given set of input values. We are given the domain (or a set of constraints), and we need to find the "best

available" values fulfilling the requirements. Formally, optimization problems are defined as follows:

Definition 2. *An optimization problem is to compute* $\min_x f(x)$ *subject to* $x \in S$*, where* $f : \mathbb{R}^n \to \mathbb{R}$ *is called the objective function,* $x \in \mathbb{R}^n$ *is called the choice variable, and* $S \subset \mathbb{R}^n$ *is called the constraint set.*

It is worth noting that x is an n-vector and can be written as $(x_1, x_2, \ldots, x_n)$.
We usually define optimization problems as minimization ones (hence "$\min$" in the definition). Maximization problems can be easily represented as minimization problem by negating the objective function. In this document we interpret the constraint set as follows:

Definition 3. *Constraint set S is represented as a set of inequality constraints:*

$$g_i(x) \leq 0, \quad i = 1, \ldots, m$$

and equality constraints

$$h_i(x) = 0, \quad i = 1, \ldots, p$$

There are many different types of mathematical programming problems, of which the best known ones are linear programming (where the objective function and the set of constraints are specified using linear equalities and inequalities only), combinatorial optimization (where the set of feasible solutions is discrete), and integer programming in which some or all variables are constrained to have integer values only.

Mathematical programming has many applications. It is used in economics (to find the way of spending money to satisfy as many needs as possible [11]), in rigid body dynamics (to compute contact forces [3]), in operations research (to support decision-making by using stochastic programming [20]), in petroleum engineering (to compute models of oil reservoirs [5]) and many more. Because of its applicability, mathematical programming is a highly developed branch of mathematics and there are many types of constraint specification and many algorithms to solve these problems.

2.3. Integer Linear Programming

An Integer Linear Program is a mathematical program in which some or all of the variables are restricted to be integers, the objective function is linear, and the constraints in the set of constraints are linear (see Definition 2 and Definition 3 in Section 2.2). It is sometimes referred to as Mixed Integer Linear Programming (MIPS) for the case where there are both variables constrained to take integer values and variables without such restriction.

Definition 4. *Having matrix* A *representing the constraints' coefficients, vector* b *representing the constraints' bounds, and vector* c *representing the objective function coefficients, the canonical form of*

integer linear program is expressed as:

$$\begin{aligned} &maximize && c^T x \\ &subject\ to && Ax \leq b, \\ & && x \geq 0, \\ & && x \in \mathbb{Z} \end{aligned}$$

There are reasons why we want to use integer variables. First one is the need to represent quantities that can only be integer—for instance, it is not possible to build 1.5 cars so the number of cars to build must be an integer. Second reason is to represent decisions—we can choose to do something or we can choose not to do something and this decision needs to be represented as either a 0 or a 1.
There are also reasons for using only linear functions and constraints. Many practical problems in operations research, network flow, and microeconomics can be expressed as linear programming problems. There is also a significant number of algorithms designed for solving linear programs effectively [21].

To better understand the idea of ILP, let us consider a simple example.

2.3.1. Example

A company has three projects that it would like to undertake, but not all of them can be selected because of a limited budget. Every project requires some investment and produces a revenue. Costs and revenues of projects are as follows:

Project Name	Cost	Revenue
A	800	5000
B	300	2500
C	400	2500

The company wants to maximize the revenue.
Before we describe the necessary constraints, let us define the variables. Let a, b, c be the binary variables meaning whether the company undertakes project A, B or C respectively. Since these variables represents some decisions, they can only take values from the set $\{0, 1\}$. For instance, $b = 1$ means that the company undertakes project B.
Assuming that the company can spend at most 1000$ for the projects, we can define the constraint for budget limitation:

$$a \cdot 800 + b \cdot 300 + c \cdot 400 \leq 1000$$

This constraint means that the total cost of projects cannot exceed $1000$$. When solving the problem, we need to assure that this constraint is satisfied for the actual values of the variables a, b and c. In other words, we cannot undertake all the projects (because then the left-hand side of the inequality would be equal to 1500). It is worth noting that the trivial solution $a = b = c = 0$ satisfies this requirement.

Now we need to define the objective function. In our example we want to maximize the revenue, so our function is:

$$f(a, b, c) = a \cdot 5000 + b \cdot 2500 + c \cdot 2500$$

We can now try to find some solution for the problem. As we can see, one possible solution is $(0, 1, 0)$ meaning that we undertake only project B and end up with revenue equal to 2500\$. It is a feasible solution but we can easily see that we can do better. There are two optimal solutions: $(1, 0, 0)$ and $(0, 1, 1)$—both of them return the revenue equal to 5000\$. It is worth noting that the latter solution is cheaper (700\$ instead of 800\$) but our objective function does not account for the cost of the projects, only the revenue.

ILP problems often use many variables, sometimes even up to one million. Because of the enormity of the possible problems, we need to use specialized algorithms to find the solutions. Unfortunately, ILP cannot be solved "quickly". In the next sections we describe the characteristics of ILP which entail difficulties in solving ILP problems.

2.4. ILP Is NP-hard

In this section we show a proof that solving ILPs is NP-hard by giving a reduction from the 3-SAT problem. We introduce the necessary definitions but with no extensive explanation. We assume that the reader is familiar with the basics of the standard complexity theory. An extended description can be found in the book of Schrijver [21].

2.4.1. Reduction Preliminaries

Before we reduce the 3-SAT problem to solving ILPs, we need to introduce some definitions from the complexity theory.

Definition 5. *A language L is in P if and only if there exists a deterministic Turing machine M, such that:*

- *M runs for polynomial time with respect to the length of the input,*
- *For all x in L, M outputs 1,*
- *For all x not in L, M outputs 0.*

In other words, a problem is in P if and only if we can find the answer for the problem in polynomial time with respect to the size of the instance of the problem.

Definition 6. *A language L is in NP if and only if there exist polynomials p and q, and a deterministic Turing machine M, such that:*

- *For all x and y, the machine M runs in time $p(|x|)$ on input (x, y)*

– *For all x in L, there exists a string y of length $q(|x|)$ such that $M(x, y) = 1$*

– *For all x not in L and all strings y of length $q(|x|)$, $M(x, y) = 0$*

In other words, a language is in NP if and only if we can verify the solution for the problem in polynomial time with respect to the size of the instance of the problem.

Definition 7. *We say that language L_1 can be reduced in polynomial time to language L_2 if there exists a function f computable in polynomial time such that:*

$$x \in L_1 \iff f(x) \in L_2$$

Definition 8. *A language L is NP-hard if and only if every language $L_2 \in NP$ can be reduced in polynomial time to L.*

Definition 9. *A language L is NP-complete if and only if it is NP-hard and belongs to NP.*

Definition 10. *We say that a logical formula ϕ is in Conjunctive Normal Form (CNF) if and only if it is in the form:*

$$\phi = (p_{11} \vee p_{12} \vee \ldots \vee p_{1k_1}) \wedge (p_{21} \vee p_{22} \vee \ldots \vee p_{2k_2}) \wedge \ldots \wedge (p_{n1} \vee p_{n2} \vee \ldots \vee p_{nk_n})$$

where every p_{ij} is a literal (i.e., either a variable or its negation). We say that formula ϕ is in 3CNF if and only if its every clause (i.e., its every part enclosed within brackets) has at most three literals.

In other words, ϕ is in CNF if it is written as a conjunction of clauses of disjunctions of literals. It is in 3CNF if every clause has at most three literals.

Definition 11. *We say that a logical formula ϕ is satisfiable if it can be made true by assigning appropriate logical values to its variables. The problem of determining whether there exists such an assignment is called Boolean Satisfiability Problem (abbreviated SAT).*

In other words, in the SAT problem we ask if it is possible to assign values "true" or "false" to the variables in such a way that the formula ϕ evaluates to true (which means that all the clauses are true).

Definition 12. *3-SAT Problem is the problem of determining whether a given 3CNF formula is satisfiable.*

Lemma 1. *(Cook's Theorem [7]) 3-SAT Problem is NP-Complete.*

2.4.2. 3-SAT Reduction to ILP

We now present the reduction proving that the ILP is NP-hard.

Theorem 1. *ILP is NP-hard.*

Proof. Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be a 3CNF formula. Each clause C_i is a disjunction of three literals $x_{i1} \vee x_{i2} \vee x_{i3}$, and each literal is either a variable or a negated variable. Let our variables be $v_1, \ldots, v_n$. We now construct an ILP program that has a feasible solution if and only if ϕ is satisfiable.

We introduce an integer variable z_i for every boolean variable v_i. The integer variable is constrained to take values zero or one, representing falsity and truth respectively.

We turn every clause C_i into an arithmetic formula Φ_i by replacing disjunction with addition, each variable v_j with variable z_j and each negated variable $\neg v_j$ with $1 - z_j$.

The ILP is:

$$\Phi_i \geq 1, \quad i = 1, \ldots, m$$
$$0 \leq z_j \leq 1, z_j \in \mathbb{Z}, \quad i = 1, \ldots, n$$

We can see that formula ϕ is satisfiable if and only if the ILP constructed above has a feasible solution. Let us assume that ϕ is satisfiable and let v^* be a satisfying assignment. By setting $z_i^* = 1$ if and only if v_i^* is true, every $\Phi_i \geq 1$. Conversely, if z^* is a feasible solution for our ILP program, then each "clause formula" Φ_i must have an integer value of at least one. We can now find the corresponding assignment which satisfies every clause C_i, which satisfies ϕ.

□

2.5. ILP Is NP-complete

To prove that ILP is NP-complete we need to show that it is in NP. We do it by proving that we can verify the solution in polynomial time. Proof of general form of ILP is difficult and beyond the scope of this thesis, so we assume that variables come from the set $\{0, 1\}$ and show the proof for this simpler type of a problem.

Theorem 2. *ILP with all variables coming from the set* $\{0, 1\}$ *is in NP.*

Proof. Assume that vector x^* with each element equal to zero or one is a proposed solution for the ILP program. We can verify in linear time with respect to the size of the vector that every element of the vector is a non-negative integer. We can verify in linear time with respect to the length of the encoding of the constraints set that the constraints are satisfied. Hence, we can perform the verification in polynomial time. □

By showing that ILP is NP-hard and that it is in NP, we have proven the following:

Theorem 3. *[21] ILP with all variables coming from the set* $\{0, 1\}$ *is NP-complete.*

2.6. Algorithms For Solving ILP Problems

There are advanced algorithms for solving ILP programs. We briefly describe the cutting-plane method, the branch and bound method, the branch and cut method and the branch and price method. More algorithms and detailed descriptions can be found in the books of Wolsey [25] and Wolsey and Nemhauser [26].

2.6.1. Cutting-Plane Method

The cutting-plane method is a term for optimization algorithms which iteratively refine an objective function using linear inequalities called cuts. The cutting-plane method works by solving a non-integer linear program (using the simplex algorithm). Found optimum is tested for being an integer solution—if it satisfies this constraint, the algorithm finishes. Otherwise, there is guaranteed to exist a linear inequality (called cut) that separates the optimum from the convex hull of the true feasible set. After adding the cut to the linear program, the process is repeated until an optimal integer solution is found.

2.6.2. Branch and Bound Method

The branch and bound method requires two tools: a splitting procedure and a tool for bounding the objective function.
The former splits the set of feasible solutions into smaller sets whose union covers the whole set of feasible solutions. This step is called branching because its application defines a search tree.
The latter computes the lower bound and the upper bound of objective value within a given subset of feasible solutions. This step is called bounding. If the lower bound for some tree node is greater than the upper bound for some other node, we may safely discard the node with low upper-bound—this step is called pruning.
By recursive application of the algorithm, we end up with a singleton which is the final solution.

2.6.3. Branch and Cut Method

Branch and cut method involves running a branch and bound algorithm and using cutting planes.
First, we try to find the solution using the simplex algorithm, as in the cutting-plane method. If the solution is not feasible, we select one of the non-integer variables x_i' and branch the problem into two: first with added constraint $x_i \leq x_i'$ and second with added constraint $x_i \geq x_i'$. Next, we repeat the process.

There are many branching strategies, when to branch and which variable to choose. Some popular ones include:

- Most Infeasible Branching—we choose the variable with the fractional part closest to 0.5
- Pseudo Cost Branching—we try to keep track of variable changes in the objective function

– Strong Branching—we test which of the candidate variables gives the best improvement to the objective function

2.6.4. Branch and Price Method

Branch and price is a branch and bound method in which we try to remove and add columns of a matrix representing constraints coefficients during the search through the tree. The key idea is based on the observation that for large problems most of the columns will have their corresponding variables equal to zero in any optimal solution, so these columns can be safely removed to reduce the memory and computational requirements.

Most branch and price algorithms are highly problem specific so the problem must be expressed in a way that allows to perform pricing effectively. Because the pricing problem may be difficult to solve, heuristic and local search methods are used.

3. What Can Be Done With ILPs

In this chapter we consider some ILP constraints which allow us to build more complicated programs. Although ILPs use only additions, multiplications by a constant, and weak inequalities, we can use them to build much more sophisticated operators, such as the conditional operator (if) and a sorting procedure. Belowe we describe how we realize such operators.

3.1. Boolean Algebra

Boolean algebra is a subarea of algebra in which the values of the variables come from the set $\{0, 1\}$, where zero means falsity and one means truth. Main operators within Boolean algebra are conjunction (and), disjunction (or) and negation (not). Boolean algebra is fundamental in the development of computer science and digital logic. As we will shortly see, all of its main operations can be represented within ILPs.

Throughout this entire section, let us assume variables a, b and c are binary, that is, their values come from the set $\{0, 1\}$.

3.1.1. Main Operators

First we define the conjunction operator (and). We note that the conjunction operator can be seen, for example, as a maximum of binary variables, or as their multiplication. We want to define a constraint ensuring that variable c is equal to $a \wedge b$. According to the following table, c is equal to one if and only if variables a and b are both equal to one.

a	b	$c = a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

To achieve this, we need to add the following constraint:

$$0 \leq a + b - 2c \leq 1$$

As we can see, when either a or b is zero, variable c cannot be equal to 1 because the whole sum would be less than 0. Thus, c can be equal to 1 only when a and b are both equal to 1, and in that situation c cannot be equal to 0 because then the whole sum would be greater than 1.

This formula can be easily generalized. Assume that variables $a_1, \ldots, a_n$ are all binary variables and we want to add a constraint forcing variable b to be equal to $a_1 \wedge a_2 \wedge \cdots \wedge a_n$. This restriction can be expressed as:

$$0 \leq a_1 + a_2 + \cdots + a_n - nb \leq n - 1$$

As we can see, when either of the variables $a_1, \ldots, a_n$ is equal to 0, b cannot be equal to 1 because then the whole sum would be less than 0. On the other hand, when all variables a_i for $i = 1, \ldots, n$ are equal to 1, b cannot be equal to 0, because then the whole sum would be greater than $n - 1$.

Disjunction can be expressed similarly to conjunction. This time we want variable c to be equal to 0 if and only if variables a and b are both equal to 0, as presented in the following table:

a	b	$c = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

To achieve this, we need to add the following constraint, which resembles the constraint used to define the conjunction operator:

$$-1 \leq a + b - 2c \leq 0$$

As we can see, when either a or b is equal to one, c cannot be equal to zero (but can be equal to one) because the whole sum would be positive. On the other hand, c can be equal to zero (and cannot be equal to one) only when a and b are both equal to 0.

As before, the above formula can be easily generalized. Let us assume that we want binary variable b to be equal $a_1 \vee a_2 \vee \cdots \vee a_n$. In order to achieve this, we need to add the following constraint:

$$-(n-1) \leq a_1 + a_2 + \cdots + a_n - n \cdot b \leq 0$$

Negation (denoted as $\neg$) is the easiest operation. This time we want variable b to be equal to 1 exactly if variable a is equal to 0, and the other way round, as shown in the following table:

a	$b = \neg a$
0	1
1	0

To implement the negation operator, we use the following constraint:

$$b = 1 - a$$

The equality operator (=) in ILPs can be expressed as a pair of inequalities, so the above formula can be rewritten as:

$$b \geq 1 - a$$
$$b \leq 1 - a$$

This can also be expressed as follows:

$$a + b = 1$$

This expression means that only one of a and b can be equal to 1 at the same time, and also that a and b cannot be both equal to 0.

3.1.2. Other Boolean Operations

There are three other operators which are sometimes classified as basic within Boolean algebra. These are: implication, equivalence (abbreviated as iff) and exclusive disjunction (exclusive or, abbreviated as xor). All these operators can be represented using conjunction, disjunction and negation. Once again we assume that a, b and c are binary variables.

We consider the implication operator first. We want to ensure that $c = a \rightarrow b$ is equal to zero when a is equal to one and b is equal to zero, and that in all other cases variable c is equal to one, as presented in the following table:

a	b	$c = a \rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

We can achieve this effect using the following propositional logic identity:

$$a \rightarrow b = \neg a \vee b$$

Using negation and disjunction we end up with the following constraint to implement the implication operator:

$$-1 \leq (1 - a) + b - 2c \leq 0$$

The if and only if operator is defined as follows: $a \leftrightarrow b$ is equal to 1 when variables a and b have the same value, as presented in the following table.

a	b	$c = a \leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

To represent this operation we can use the following identity:

$$a \leftrightarrow b = (a \wedge b) \vee (\neg a \wedge \neg b)$$

We need two temporary binary variables to represent this constraint:

$$c_{a,b} = a \wedge b$$
$$c_{\neg a,\neg b} = \neg a \wedge \neg b$$

Now we can use our implementation of conjunction, disjunction and negation operators, and express the iff operator with the following constraints:

$$0 \leq a + b - 2 \cdot c_{a,b} \leq 1$$
$$0 \leq (1 - a) + (1 - b) - 2 \cdot c_{\neg a,\neg b} \leq 1$$
$$-1 \leq c_{a,b} + c_{\neg a,\neg b} - 2 \cdot c \leq 0$$

The last operation that we consider is the exclusive or operator. Variable $c = a \oplus b$ is equal to one whenever variables a and b have different values, as presented in the following table:

a	b	$c = a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

To express this constraint we can use the iff operator or we can use the following, direct, identity:

$$a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$$

To implement this idea, we need two additional temporary binary variables defined as follows:

$$c_{a,\neg b} = a \wedge \neg b$$
$$c_{\neg a,b} = \neg a \wedge b$$

The final set of constraints that ensure that $c = a \oplus b$, is as follows:

$$0 \leq a + (1 - b) - 2 \cdot c_{a,\neg b} \leq 1$$
$$0 \leq (1 - a) + b - 2 \cdot c_{\neg a,b} \leq 1$$
$$-1 \leq c_{a,\neg b} + c_{\neg a,b} - 2 \cdot c \leq 0$$

3.2. Arithmetic

There are four basic arithmetic operations: addition, subtraction, multiplication and division. All these operations are binary (which means they have two arguments) and can be performed either on constant values or on the ILP variables.

Addition and subtraction are easy and do not need much effort, because they can be directly represented within the ILP constraints. However, multiplication and division are not that simple.

First, let us consider multiplication. We can multiply two constant values (for instance $2 \cdot 3$), because the result is still a constant. We can also multiply a variable by a constant (for example $a \cdot 5$, where a is a variable), because the result of this operation is still linear and can be directly represented within ILPs. However, we are not allowed to multiply two variables (for instance $a \cdot b$, where a and b are variables), because the outcome of this operation is not linear. This can be better seen in case of multiplying variable a by itself: $a \cdot a = a^2$.
Division goes the same way—we can easily represent division of two constant values, or division of a variable by a constant, but we need much greater effort to divide two variables.

Since we can use linear constraints only, we need to find a way of representing multiplication and division of two variables with only linear inequalities. In this section we express multiplication and division in this way. We also consider the issue of efficiency of our implementation.

3.2.1. Multiplication

To multiply two variables, we need to make an assumption regarding the greatest possible values of these variable. In the next part of this document, we assume that the absolute value of each variable is not greater than $2^n - 1$, where n is some positive integer. First, we show a method for multiplying non-negative numbers and then we quickly argue how it can be generalized to possibly negative ones.

Unsigned magnitude representation

Let us assume that we want to define constraints causing variable c to be equal $a \cdot b$. First, we need to find the unsigned magnitude representations of a and b. This representation is similar to computer's internal representation of integer numbers.
Unsigned magnitude representation of an n-bit number a consists of n bits $a_0, a_1, \ldots, a_{n-1}$, where each bit a_i is binary and is multiplied by its weight, which is equal to 2^i. Sum of all of the bits multiplied

by their weights is equal to a. To sum up, non-negative variable a can be represented by the following formula:

$$a = \sum_{i=0}^{n-1} a_i 2^i$$

For instance, 12 can be represented using 4 bits:

$$12 = 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 = 0 + 0 + 4 + 8$$

We can also represent 12 using 6 bits:

$$12 = 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 0 \cdot 16 + 0 \cdot 32 = 0 + 0 + 4 + 8 + 0 + 0$$

The greatest possible value that can be represented using n bits is equal to $2^n - 1$, when all bits are equal to 1:

$$1 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + \cdots + 1 \cdot 2^{n-1} = 1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$$

We write $\overline{a_{n-1} \ldots a_0}$ to denote the unsigned magnitude representation of variable a. Every non-negative integer can be represented this way. In the next subsection we use this representation to perform multiplication.

Multiplication of non-negative numbers

In this section we want to multiply two non-negative integers a and b. We use the unsigned magnitude representation defined in the previous subsection—each variable is represented on n bits as $\overline{a_{n-1}a_{n-2} \ldots a_2a_1a_0}$ and $\overline{b_{n-1}b_{n-2} \ldots b_2b_1b_0}$ respectively.

We are going to express the algorithm of long multiplication. Using this algorithm, when we calculate $a \cdot b$ we multiply a by each digit of b and then sum the results, remembering about multiplication by consecutive powers of 2. We can do almost the same using ILP constraints because multiplication of two binary digits can be treated as their conjunction, which can be done easily as presented in the previous section.

Before we present the general solution, let us calculate the simple example: we multiply $a = 3$ and $b = 4$, each treated as a 4 bit number ($n = 4$).
First we need to find the unsigned magnitude representation of a and b:

$$a = 3 = a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 4 + a_3 \cdot 8$$
$$b = 4 = b_0 \cdot 1 + b_1 \cdot 2 + b_2 \cdot 4 + b_3 \cdot 8$$

We get: $a_0 = 1$, $a_1 = 1$, $a_2 = 0$, $a_3 = 0$, $b_0 = 0$, $b_1 = 0$, $b_2 = 1$, $b_3 = 0$. Now we perform the long multiplication:

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
\cdot & & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
\end{array}
$$

Using the previously found values we obtain:

$$
\begin{array}{cccccccc}
 & & & & 0 & 0 & 1 & 1 \\
\cdot & & & & 0 & 1 & 0 & 0 \\
\hline
 & & & & 0\cdot 0 & 0\cdot 0 & 1\cdot 0 & 1\cdot 0 \\
 & & & 0\cdot 0 & 0\cdot 0 & 1\cdot 0 & 1\cdot 0 & \\
 & & 0\cdot 1 & 0\cdot 1 & 1\cdot 1 & 1\cdot 1 & & \\
+ & 0\cdot 0 & 0\cdot 0 & 1\cdot 0 & 1\cdot 0 & & & \\
\hline
 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
$$

The result is equal to 12, which comes from $0\cdot 1+0\cdot 2+1\cdot 4+1\cdot 8+0\cdot 16+0\cdot 32+0\cdot 64$. We could write our result using the following expression:

$$
\begin{gathered}
a\cdot b= \\
a_0\cdot b_0+2\cdot a_1\cdot b_0+4\cdot a_2\cdot b_0+8\cdot a_3\cdot b_0+ \\
+2\cdot(a_0\cdot b_1+2\cdot a_1\cdot b_1+4\cdot a_2\cdot b_1+8\cdot a_3\cdot b_1)+ \\
+4\cdot(a_0\cdot b_2+2\cdot a_1\cdot b_2+4\cdot a_2\cdot b_2+8\cdot a_3\cdot b_2)+ \\
+8\cdot(a_0\cdot b_3+2\cdot a_1\cdot b_3+4\cdot a_2\cdot b_3+8\cdot a_3\cdot b_3)
\end{gathered}
$$

Which in our example is equal to:

$$
\begin{gathered}
3\cdot 4= \\
1\cdot 0+2\cdot 1\cdot 0+4\cdot 0\cdot 0+8\cdot 0\cdot 0+ \\
+2\cdot(1\cdot 0+2\cdot 1\cdot 0+4\cdot 0\cdot 0+8\cdot 0\cdot 0)+ \\
+4\cdot(1\cdot 1+2\cdot 1\cdot 1+4\cdot 0\cdot 1+8\cdot 0\cdot 1)+ \\
+8\cdot(1\cdot 0+2\cdot 1\cdot 0+4\cdot 0\cdot 0+8\cdot 0\cdot 0)= \\
0+2\cdot 0+4\cdot(1+2)+8\cdot 0= \\
0+0+4\cdot 3+0=12
\end{gathered}
$$

Since we cannot multiply two variables (because we would get a non-linear constraint), we need to represent every multiplication of two digits (two binary variables) as their conjunction.
To sum up, our algorithm comes as follows: we first find the unsigned magnitude representations of multiplicand and the multiplier, next we calculate conjunction of every digit of the multiplicand and every digit of the multiplier, and finally we form the result by summing the digits multiplied by correct weights.

To summarize, the general formula for multiplication of two non-negative integers a and b (using unsigned magnitude representation described in previous section) is:

$$\overline{a_{n-1}a_{n-2}\dots a_2a_1a_0} \cdot \overline{b_{n-1}b_{n-2}\dots b_2b_1b_0} = \sum_{i=0}^{n-1} 2^i \cdot \left(\sum_{j=0}^{n-1} 2^j \cdot (a_j \wedge b_i) \right)$$

Let's analyze this formula. Every conjunction means simply multiplication of two digits. As we remember, conjunction of variables is equal to one if and only if both arguments are equal to one so it can be treated as multiplication. The inner sum means multiplication of number a by a single digit of b. Every digit a_j must be multiplied by 2^j which means the weight of this digit.
The outer sum means multiplication of a by successive digits of b. In this case we again need to multiply the whole sum by 2^i, the weight of digit b_i.

Multiplication of Signed Numbers

To multiply numbers which do not need to be non-negative, we can multiply their absolute values and then calculate the sign bit, which is equal to the exclusive or of the sign bits of the multiplied numbers.

Cost of Multiplication

Let us hold on for a moment and analyze the presented solution. We have made an assumption about the greatest possible value of a variable. At first, it may seem harmful because it means that we will not be able to operate on unbounded numbers anymore. But there is a catch—we need to remember that we will probably be using a computer to find the solution and the machine cannot perform efficient calculations on unbounded numbers—it needs to represent them as packs of bits to be fast. So our assumption about the maximum possible value is still valid—we were never able to perform calculations on arbitrarily large numbers. Second, we need to calculate how many binary variables we will need to multiply two numbers. On the 32-bit architecture typical integer consists of 32 bits which allows us to represent unsigned numbers in range $[0; 2^{32} - 1]$. So to multiply two variables we will need $2 \cdot 32$ (for unsigned magnitude representation) plus $32 \cdot 32$ (for conjunctions) temporary variables which is equal to 1088. This means more than one thousand variables just to multiply two non-negative variables. But the good news is: it works!

Faster multiplication

There is another way to multiply two numbers, which requires fewer temporary variables. Instead of multiplying every digit of the multiplicand by every digit of the multiplier, we can multiply the whole

multiplicand by every digit of the multiplier. This can be done with the following formula:

$$a \cdot \overline{b_{n-1}b_{n-2}\ldots b_2b_1b_0} = \sum_{i=0}^{n-1} 2^i \cdot \min\left(a, b_i \cdot K\right),$$

where K means some value greater than 2^n (greater than any variable can be). Implementation within ILP for finding the minimum of two numbers is described in the next section.
This method needs only n temporary variables for unsigned magnitude representation and $5n$ variables for maximum function.

3.2.2. Integer Division

In this subsection we want to find the value of $c = \left\lfloor \frac{a}{b} \right\rfloor$, where a and b are non-negative integers. We use multiplication described in the previous section.
We want to express our requirement, that we are looking for the greatest possible value c which multiplied by the denominator is not greater than the numerator. We can express this with the followings:

$$c \cdot b \leq a$$
$$a \leq (c+1) \cdot b - 1$$

The first part of this formula ($c \cdot b \leq a$) means that we are looking for any value c which multiplied by the denominator b is not greater than the numerator a. Second part of this formula ($a \leq (c+1+) \cdot b$) requires that $c+1$ multiplied by the denominator is not less than numerator minus one. Because a, b and c are all variables, we need to multiply them using the method described in the previous section.
The reader might wonder why we need to subtract one from the numerator and not just use $a \leq (c+1) \cdot b$ as the latter expression. The problem arises when we try to calculate $\left\lfloor \frac{1}{1} \right\rfloor$ because while the proper result is 1, zero also fulfills the constraints (because $1 \leq (0+1) \cdot 1$). Subtracting one solves this problem.
To illustrate the method, let us calculate the following simple example: $c = \left\lfloor \frac{7}{2} \right\rfloor$. We add the following constraints (or, rather, abbreviations for the constraints that define multiplication):

$$c \cdot 2 \leq 7$$
$$7 \leq (c+1) \cdot 2 - 1$$

The solution is $c = 3$ because $3 \cdot 2 \leq 7 \leq 4 \cdot 2 - 1$.

To divide signed integers, we first divide them as if they were non-negative and then we calculate the sign—just like in calculating the sign in multiplication of possibly negative integers described in the previous section.

3.3. Comparisons

In this section we cover ways of answering questions of the form "is value of variable a greater than the value of variable b". We stress that it is not adding constraints such as $a > b$ because adding such a constraint would cause a to always be greater than b—we only want to know if a is greater than b in our particular case or not.

In order to check whether the value of variable a is greater than the value of variable b, we need to once again make an assumption about the greatest possible value held in a variable. Let us assume that every variable v satisfies $|v| < 2^n$ where n is some positive integer. Let us denote $2^n - 1$ by k so every variable v satisfies $|v| \leq k$. We also need value $K = 2 \cdot k + 1$.
Let us now write the constraint. We want binary variable x to be equal to 1 if and only if a is greater than b. We need to add the following constraints:

$$0 \leq b - a + Kx \leq K - 1$$

When $a < b$ is true, the value of $b - a$ is positive so x has to be equal to 0 ; if it were 1, the whole sum would be greater than $K - 1$. The same is true when $a = b$. On the other hand, when $a > b$ then their difference $b - a$ is negative and x has to be equal to 1 to satisfy the constraints. We need to have value K greater than $2 \cdot k$ because if $a = -k$ and $b = k$, $b - a$ is equal to $2k$. Let us for a moment assume, that $K \leq 2k$. In this situation our constraints are:

$$0 \leq b - a + Kx \leq K - 1$$

Now we substitute a and b by k and $-k$:

$$0 \leq k - (-k) + Kx \leq K - 1$$

After some simplifications we end up with:

$$0 \leq 2k + Kx \leq K - 1$$

Because x is a binary variable and $K \leq 2k$, $2k + Kx \geq 2k + 2kx \geq 2k\,(1 + x) \geq 2k$. On the other hand, $K - 1 \leq 2k - 1$, so inequality $2k + Kx \leq K - 1$ has no solutions.

We now know how to perform the "greater than" comparison. The less than comparison can be done in the same way as greater than. All we need to do is swap the roles of a and b.

Other comparisons can be expressed using greater than comparison and the Boolean algebra.

We know that a is equal to b when a is not greater than b and a is not less than b. To express this in an ILP, we first calculate whether $a > b$ denoted by $x_{a>b}$, then we calculate $a < b$ denoted by $x_{a<b}$, and finally the expression $\neg x_{a>b} \wedge \neg x_{a<b}$ is true if and only if a is equal to b.
To calculate whether $a \neq b$ we can calculate $a = b$ and calculate the logical negation of the result. We can also use the identity that $a \neq b$ when either a is greater than b or b is greater than a.

3.4. Miscellaneous Functions

In this section we provide formulas for some useful functions: absolute value, minimum, and maximum.

3.4.1. Absolute Value

Absolute value is defined as follows: $|a|$ is equal to a when $a \geq 0$ and is equal to $-a$ when $a < 0$. In this section we give a formula for calculating value b satisfying condition $b = |a|$.

First, we need to be sure that b is non-negative. To achieve that we can use the following constraints:

$$b \geq a$$
$$b \geq -a$$

Now we must ensure that b is equal to $|a|$. Let us add another variable x representing the following boolean expression:

$$x = (b + a = 0) \vee (b - a = 0)$$

And now we need to add one constraint:

$$x = 1$$

Since b is non-negative, when a is non-negative also $b - a$ must be equal to 0. But when a is negative, $b + a$ must be equal to 0. This ensures that $b = |a|$.

3.4.2. Maximum and Minimum

Now we want to find the maximum and the minimum of two values. To be precise, we want to find $M = \max(a, b)$ and $m = \min(a, b)$ where a and b are two variables. In order to do that, we first add constraints to be sure that M will not be less than either of a and b, and m will not be greater than either of a and b:

$$M \geq a$$
$$M \geq b$$
$$m \leq a$$
$$m \leq b$$

Thus, we know that $M \geq m$ and $M - m$ is non-negative. Now we only need to ensure that M and m are the values that we are looking for. We add the following condition:

$$M - m = |a - b|$$

This constraint means that the difference of M and m is equal to the absolute value of the difference of a and b. Since $M - m$ is non-negative, we do not need to use the absolute value operator on the left-hand side of the formula.

This formula says that the distance on the number line from m to M is equal to the distance on the number line from a to b. Since M is not less than $\max(a, b)$ and m is not greater than $\min(a, b)$, we conclude that M is equal to $\max(a, b)$ and m is equal to $\min(a, b)$ which is exactly what we want.

To sum up, we need to add the following constraints:

$$M \geq a$$
$$M \geq b$$
$$m \leq a$$
$$m \leq b$$
$$d_{M,m} = M - m$$
$$d_{a,b} = |a - b|$$
$$d_{M,m} = d_{a,b}$$

We calculate the absolute value using the method from the previous section. However, there is a catch. In Section 3.4.1, when we calculate the absolute value of variable a, we compare $b + a$ and $b - a$ to zero. But in Section 3.3 we assumed that the absolute values of the compared values are less than 2^n. Imagine now that we want to calculate maximum of $a = k$ and $b = -k$, where k is defined as in Section 3.3. We need to calculate the absolute value of $a - b = k - (-k) = 2k$, which is greater than k, so we cannot do it using the method from Section 3.4.1.

To solve this problem, we can define variable K from section 3.3 as $4 \cdot k + 1$.

3.5. Conditional Operator

In this section we describe a conditional operator that allows us to define variables as having some value depending on a given condition. This can be interpreted as a kind of "if" operator available in nearly all programming languages. We need to remember that in an ILP there is no such thing as a flow of execution. All we can do is add constraints which ensure that variables have correct values when there is a feasible solution. In effect, the conditional operator is simply a multiplication.

Let us assume that our condition is represented as a binary variable c. We also have two non-negative variables: t and f, which mean the value when condition is true and the value when the condition is false, respectively. We also have variable a, which we want to be equal to t when c is true, and to be equal to f otherwise. We denote this as $a = c\ ?\ t\ :\ f$, which resembles the "ternary operator" from many programming languages.

To express our idea we can use the following constraint:

$$a = c \cdot t + (1 - c) \cdot f$$

As we can see, when c is true the whole sum i equal to $1 \cdot t + 0 \cdot f = t$. On the other hand, when c is false, the result is equal to $0 \cdot t + 1 \cdot f = f$. We can calculate multiplication using the method from Section 3.2.1 or use the direct multiplication from ILPs if t and f are constants.

3.6. Sorting

In this section we describe a method for sorting a sequence of numbers. First we solve a simple example to build intuition for this operation, and then we describe the general formula.

3.6.1. Sorting Three Variables

Let us assume that we want to sort vector $v = (a; b; c)$ where $a = 5$, $b = 3$ and $c = 5$. As we can see, our vector does not need to have all components distinct. We sort it using the algorithm resembling the "selection sort" algorithm. First, for each variable we count the "not less" variables, and then we select variables for proper places in the result vector.

We need to compare all variables to find which are bigger than the others. We calculate:

$$V_{a \geq b} = a \geq b$$
$$V_{a \geq c} = a \geq c$$
$$V_{b \geq a} = b \geq a$$
$$V_{b \geq c} = b \geq c$$
$$V_{c \geq a} = c \geq a$$
$$V_{c \geq b} = c \geq b$$

Now we sum the above results to count the number of "not less than" variables for each element of the vector:

$$a_s = V_{a \geq b} + V_{a \geq c}$$
$$b_s = V_{b \geq a} + V_{b \geq c}$$
$$c_s = V_{c \geq a} + V_{c \geq b}$$

In our example, the variables have the following values:

$$V_{a \geq b} = 1$$
$$V_{a \geq c} = 1$$
$$V_{b \geq a} = 0$$
$$V_{b \geq c} = 0$$
$$V_{c \geq a} = 1$$
$$V_{c \geq b} = 1$$
$$a_s = 2$$
$$b_s = 0$$
$$c_s = 2$$

When we already know that variable a is not less than the two other variables, variable b is less than all the other variables and variable c is not less than the two other variables, we can sort them. The smallest variable is the one that is not less than no other variables, the smallest but one variable is the one that is not less than one other variable and so on.
We can express the above reasoning using the following conditions:

$$m_0 = \min(a_s \geq 0 \; ? \; a \; : \; k \; ; \; b_s \geq 0 \; ? \; b \; : \; k \; ; \; c_s \geq 0 \; ? \; c \; : \; k)$$
$$m_1 = \min(a_s \geq 1 \; ? \; a \; : \; k \; ; \; b_s \geq 1 \; ? \; b \; : \; k \; ; c_s \geq 1 \; ? \; c \; : \; k)$$
$$m_2 = \min(a_s \geq 2 \; ? \; a \; : \; k \; ; \; b_s \geq 2 \; ? \; b \; : \; k \; ; c_s \geq 2 \; ? \; c \; : \; k)$$

where k is defined as in Section 3.3. Each formula has the same structure: we compare each count of "not less than" variables (a_s, b_s and c_s in our example) to the consecutive non-negative integer numbers, which is simply the "selection" part of "selection sort" algorithm.
Now we can define our result vector w as $w = (m_0; m_1; m_2)$. In our example, values are:

$$m_0 = \min(1\ ?\ a\ :\ K\ ;\ 1\ ?\ b\ :\ K\ ;\ 1\ ?\ c\ :\ K) = \min(a; b; c) = b = 3$$
$$m_1 = \min(1\ ?\ a\ :\ K\ ;\ 0\ ?\ b\ :\ K\ ;\ 1\ ?\ c\ :\ K) = \min(a; K; c) = a = 5$$
$$m_2 = \min(1\ ?\ a\ :\ K\ ;\ 0\ ?\ b\ :\ K\ ;\ 1\ ?\ c\ :\ K) = \min(a; K; c) = a = 5$$
$$w = (3; 5; 5)$$

Which shows that we have sorted the initial vector.

3.6.2. General Sorting Formula

Let us assume that we want to sort vector $v = (a_0; a_1; \ldots; a_n)$ and denote the result as $w = (m_0; m_1; \ldots; m_n)$
In order to do that, first we need to calculate:

$$\underset{\substack{i=0,\ldots,n\\ j=0,\ldots,n\\ j\neq i}}{\forall} V_{i\geq j} = a_i \geq a_j$$

$$\underset{i=0,\ldots,n}{\forall} a_{i,s} = \sum_{\substack{j=0\\ j\neq i}}^{n} V_{i\geq j}$$

As before, for each variable we count the number of "not less than" variables and then sum the results. We add the final constraints of the form:

$$\underset{i=0,\ldots,n}{\forall} m_i = \min_{j=0,\ldots,n} (a_{j,s} \geq i\ ?\ a_j\ :\ k)$$

and define the final result:

$$w = (m_0; m_1; \ldots; m_n)$$

3.7. Summary

In this section we have shown all of the operators discussed throughout the chapter. We assume that the absolute value of each integer variable is not greater than $k = 2^n - 1$, and that $K = 4 \cdot k + 1$.

Operation	Arguments	Result	Constraints
		Boolean Algebra	
Conjunction	Booleans: $a_0, \ldots, a_v, b$	$b = \bigwedge_{i=0}^{v} a_v$	$0 \leq a_0 + a_1 + \cdots + a_v - 2b \leq v - 1$
Disjunction	Booleans: $a_0, \ldots, a_v, b$	$b = \bigvee_{i=0}^{v} a_v$	$-v + 1 \leq a_0 + a_1 + \cdots + a_v - 2b \leq 0$
Negation	Booleans: a, b	$b = \neg a$	$b = 1 - a$
Implication	Booleans: a, b, c	$c = a \rightarrow b$	$-1 \leq (a - 1) + b - 2c \leq 0$
If and only if	Booleans: $a, b, c, c_{a,b}, c_{\neg a,\neg b}$	$c = a \leftrightarrow b$	$0 \leq a + b - 2 \cdot c_{a,b} \leq 1$ $0 \leq (1 - a) + (1 - b) - 2 \cdot c_{\neg a,\neg b} \leq 1$ $-1 \leq c_{a,b} + c_{\neg a,\neg b} - 2 \cdot c \leq 0$
Exclusive or	Booleans: $a, b, c, c_{a,\neg b}, c_{\neg a,b}$	$c = a \oplus b$	$0 \leq a + (1 - b) - 2 \cdot c_{a,\neg b} \leq 1$ $0 \leq (1 - a) + b - 2 \cdot c_{\neg a,b} \leq 1$ $-1 \leq c_{a,\neg b} + c_{\neg a,b} - 2 \cdot c \leq 0$
		Arithmetic	
Unsigned magnitude representation	Non-negative integer: a Booleans: $a_0, a_1, \ldots, a_{n-1}$	$a = \sum_{i=0}^{n-1} a_i 2^i$	$a = \sum_{i=0}^{n-1} a_i 2^i$
Multiplication	Non-negative integers: a, b, c Booleans: $a_0, \ldots, a_{n-1}$ $b_0, \ldots, b_{n-1}$	$c = a \cdot b$	$a = \sum_{i=0}^{n-1} a_i 2^i$ $b = \sum_{i=0}^{n-1} b_i 2^i$ $c = \sum_{i=0}^{n-1} 2^i \cdot \left(\sum_{j=0}^{n-1} 2^j \cdot (a_j \wedge b_i) \right)$
Division	Non-negative integers: a, b, c	$c = \left\lfloor \frac{a}{b} \right\rfloor$	$c \cdot b \leq a$ $a \leq (c + 1) \cdot b - 1$
		Comparisons	
Greater than	Integers: a, b Boolean: c	$c = a > b$	$0 \leq b - a + Kx \leq K - 1$
Less than	Integers: a, b Boolean: c	$c = a < b$	$0 \leq a - b + Kx \leq K - 1$
Greater or equal	Integers: a, b Boolean: c	$c = a \geq b$	$0 \leq a - b + K\,(1 - x) \leq K - 1$
Less or equal	Integers: a, b Boolean: c	$c = a \leq b$	$0 \leq b - a + K\,(1 - x) \leq K - 1$

Equal to	Integers: a, b Booleans: $c_{a\geq b}, c_{a\leq b}, c$	$c = a = b$	$c_{a\geq b} = a \geq b$ $c_{a\leq b} = a \leq b$ $c = c_{a\geq b} \wedge c_{a\leq b}$
Not equal to	Integers: a, b Booleans: $c_{a>b}, c_{a<b}, c$	$c = a \neq b$	$c_{a>b} = a > b$ $c_{a<b} = a < b$ $c = c_{a>b} \vee c_{a<b}$
Miscellaneous functions			
Absolute value	Integer: a Non-negative integer: b	$b = \lvert a \rvert$	$b \geq a$ $b \geq -a$ $(b + a = 0) \vee (b - a = 0) = 1$
Maximum and Minimum	Non-negative integers: a, b, M, m	$M = \max(a, b)$ $m = \min(a, b)$	$M \geq a$ $M \geq b$ $m \leq a$ $m \leq b$ $M - m = \lvert a - b \rvert$
Conditional	Boolean: c Non-negative integers: x, t, f	$x = c\ ?\ t\ :\ f$	$x = c \cdot t + (1 - c) \cdot f$
Sorting	Vectors of non-negative integers: $u = (a_0; \ldots; a_w)$ $v = (m_0; \ldots; m_w)$ Booleans: $\underset{0\leq i,j\leq w}{\forall} V_{i\geq j}, a_{i,s}$	v - sorted u	$\underset{\substack{i=0,\ldots,w \\ j=0,\ldots,w \\ j\neq i}}{\forall} V_{i\geq j} = a_i \geq a_j$ $a_{0,s}, a_{1,s}, \ldots, a_{w,s}$ $\underset{i=0,\ldots,w}{\forall} a_{i,s} = \sum_{\substack{j=0 \\ j\neq i}}^{w} V_{i\geq j}$ $\underset{i=0,\ldots,w}{\forall} m_i = \min_{j=0,\ldots,w} (a_{j,s} \geq i\ ?\ a_j\ :\ k)$

4. Students Placement Problem Represented Using ILP

In this chapter we describe in detail our Students Placement Problem. We define all the necessary constraints required to find a "good" solution and consider what does "good" mean in this context. We also look into some methods of calculating students' happiness and answer the question, whether we can make everyone happy.

4.1. Students Placement Problem

Imagine a university and students who want to enroll for some courses. Every person has different needs—some people are working full-time, some are learning how to play the guitar, some do not like to get up early. Students cannot decide about every detail of their classes—for instance they are not able to change the days on which they have lectures—but they can choose particular classes they want to attend. Our goal is to find an allocation of students to the courses which will make them as happy as possible.

At the beginning of each semester, the authorities of the university prepare a timetable of classes. The timetable describes starting and ending times of lectures/laboratories/exercises (e.g., calculus begins at 8:00 AM on Monday and ends at 9:30 AM on Monday). Because there are many students and they are physically not able to attend the same laboratories at the same time (because, for instance, the room is not large enough), the timetable consists of several laboratories/exercises from which the students can choose ones that suit them most. For example, the timetable could describe three calculus laboratories, the first one on Monday, the second one on Tuesday, and the third one on Friday. Each of them can fit up to 20 students, so at most 60 people can attend calculus laboratories every week.

By a "course" we understand for instance "Calculus", by a class we understand, for example, a single instance of "calculus exercises" starting at 8:00 AM on Monday.

Having such a timetable, we calculate the placement of students, taking their particular needs into account. We are bound by some formal requirements—for instance, the maximum number of students able to attend particular classes—but the authorities of the university do not impose restrictions on the form of final students' schedules (for instance, there is no requirement of attending university every day).

To sum up, we acquire a general timetable of classes and calculate the allocation of students to classes, which allows them to finish the semester and pass the exams. And we do our best to make the final schedules as suitable for the students as possible.

Definition of the Timetable

Let $\#courses$ denote the total number of courses and let $G = \{C_1, C_2, \ldots, C_{\#courses}\}$ be the general schedule consisting of $\#courses$ courses. Each course C_i is defined as a set of classes $\{K_{i,1}, K_{i,2}, \ldots, K_{i,\#classes(C_i)}\}$, where $\#classes\,(C_i)$ means the total number of classes in course C_i. Every class has assigned time, for example lecture can start at 8:00 AM on Monday and finish at 9:30 AM on Monday. Each class has its own start time, end time, and the maximum number of people that can attend it. Each course has a number of classes which every student needs to attend every week to finish the course.

We ask the students to provide their preferences regarding which classes they would like to attend. Let $\#students$ be the total number of students and let $S = \{S_1, S_2, \ldots, S_{\#students}\}$ be the set of students who want to enroll for courses. Every student S_i is defined as a set of wishes for classes:

$$S_i = \{preference\,(S_i, K_{1,1})\,, preference\,(S_i, K_{1,2})\,, \ldots, preference\left(S_i, K_{1,\#classes(C_1)}\right),$$
$$preference\,(S_i, K_{2,1})\,, preference\,(S_i, K_{2,2})\,, \ldots, preference\left(S_i, K_{2,\#classes(C_2)}\right),$$
$$\ldots, preference\left(S_i, K_{\#courses,\#classes\left(C_{\#courses}\right)}\right)\Bigg\}$$

For the sake of brevity, we assume that every student is registered for all the courses and has preferences regarding all of the classes. It might look like a simplification of the problem, but in fact it is not—this assumption only allows us to not bother with not existing values where it would make description of the constraints more difficult. Absence of values can be handled quite naturally—for instance when some student is not registered for a course, we simply do not consider his preferences for that course. The actual implementation described in Chapter 5 handles the fact that some values are missing.

Every preference is defined as a single integer $preference\,(S_i, K_{j,k})$ satisfying the condition $-1 \leq W^{i,j,k} \leq 10$. This means that each student can say how much he wants to attend a particular class : 0 means that he doesn't want to attend the class, 10 means that he wants to attend the class a lot. Of course, the lower and the upper bounds are not significant, they can be adjusted to particular needs. We also consider special value -1, which means that the student cannot attend the class—because he is, for instance, working during the class.

Let us define our goal. We want to assign classes for students in a way that every student is able to attend his or her classes (which means, for instance, that he or she would not need to be in two places at the same time). We will need to consider some necessary constraints in order to generate a feasible plan, i.e., a schedule that is not necessary "good" but allows the students to successfully finish their studies.

In Section 4.4 we consider some methods of calculating students' happiness, but for now we will focus on schedules that maximize the sum of the preference points of each student assigned to the classes to which this student was allocated. To understand it better, let us consider a simple example. Imagine a "calculus" course which has three classes per week—the first class starting at 8:00 AM on Monday, the second class starting at 10:00 AM on Tuesday, and the third class starting at 11:00 AM on Friday. Let

us say that student John has preference for the first class equal to 5, for the second class equal to 7, and for the third class equal to 3. If the schedule says that John should attend classes on Monday and Friday, he gets 8 preference points (5 for the class on Monday and 3 for the class on Friday) and we say that John gained 8 points. We compare schedules by simply comparing the sums of the preference points which students gain in them. In Section 4.4 we describe many more methods of calculating happiness, but we do not need them to define the necessary constraints—for now we can treat "happiness" as an intuitive measure of the quality of the solution, indicating that the final schedule for students goes with their wishes.

4.2. ILP for the Problem

In this section we define Students Placement Problem as an ILP problem. In order to do that, for each class and each student we define a single binary variable $x_{i,j,k}$ meaning if student S_i should attend class $K_{j,k}$ in course C_j. Formally we define variables:

$$\underset{\substack{i=1,\ldots,\#students\\ j=1,\ldots,\#courses\\ k=1,\ldots,\#classes(C_j)}}{\forall} x_{i,j,k}$$

In the solution, value one of $x_{i,j,k}$ means that student should attend class $K_{j,k}$, and value zero means that he should not.
Let us now define some necessary constraints allowing the solution to be acceptable to students.

4.2.1. Necessary Constraints

We now define some constraints which our schedule should satisfy. For instance we cannot assign a student to two classes which start at the same time because the student cannot be in two places at the same time. We also cannot assign too many people to a single class because we do not have infinite rooms. These constraints will make our solution "feasible". It may not be the best one, but it will give students an opportunity to successfully finish their studies.

Enough Classes for each Student

When a student registers for a course, he needs to attend a specified number of classes every week to complete the course. The number of classes to attend is denoted as $\#attend\,(C_i)$. We need to be sure that after solving the problem every student will be assigned to exactly $\#attend\,(C_i)$ classes in each course C_i that he is registered for.
In order to do that, we need the following constraints:

$$\underset{\substack{i=1,\ldots,\#students\\ j=1,\ldots,\#courses}}{\forall} \sum_{k=1}^{\#classes(C_j)} x_{i,j,k} = \#attend\,(C_j)$$

This formula means that exactly $\#attend\,(C_j)$ variables of student S_i for course C_j are equal to 1, which means that this student is assigned to the correct number of classes.

Collisions

We do not want students to be in two places at the same time so we need to make sure that they are assigned to at most one class from the set of all classes happening concurrently. In order to do that, for all pairs of classes happening at the same time we need to add the following constraints:

$$\underset{i=1,\ldots,\#students}{\forall} \quad \underset{\substack{(j,k)\\(l,m)\\K_{j,k}\text{ and }K_{l,m}\text{ happens concurrently}}}{\forall} \quad x_{i,j,k} + x_{i,l,m} \leq 1$$

Room Capacity

We are bounded by the maximum number of people who can attend a single class. Let $\#max\,(K_{i,j})$ be the maximum number of students that can attend class $K_{i,j}$. We need to add the following constraints:

$$\underset{\substack{j=1,\ldots,\#courses\\k=1,\ldots,\#classes(C_j)}}{\forall} \quad \sum_{i=1}^{\#students} x_{i,j,k} \leq \#max\,(K_{j,k})$$

Blocked Classes

Every student can mark some classes as blocked (the student assigns preference -1 to this class), which means that he is not able to attend them. We need to make sure that in our final schedule students do not need to attend blocked classes:

$$\underset{\substack{i=1,\ldots,\#students\\j=1,\ldots,\#courses\\k=1,\ldots,\#classes(C_j)\\preference\left(S_i,K_{j,k}\right)=-1}}{\forall} \quad x_{i,j,k} = 0$$

4.3. Additional Features

Apart from the constraints, which are necessary for finding a schedule allowing the students to complete their studies, we can imagine many constraints, whose goal is to improve the quality of the schedule.

4.3.1. Lower Bound on the Number of Students in a Class

Let us assume that we have a lower bound on the number of people that attend a given class. Let us denote it by $\#min\,(K_{i,j})$. In our final solution we want at least $\#min\,(K_{i,j})$ students to be assigned to that class. In order to achieve this, we need the following constraint:

$$\underset{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#classes(C_j)}}{\forall} \sum_{i=1}^{\#students} x_{i,j,k} \geq \#min\left(K_{i,j}\right)$$

4.3.2. Class Sizes Divisible by Team Sizes

Sometimes the lecturer wants to divide the students into teams of equal size. If we want to add this constraint to our requirements, we need to use the following formula:

$$\underset{(j,k)}{\forall} \sum_{i=1}^{\#students} x_{i,j,k} = a \cdot \#team\left(K_{j,k}\right)$$

for each class $K_{i,j}$ where students should create teams ($\#team\left(K_{i,j}\right)$ means the size of a team). In this formula variable a is any integer (it does not need to be explicitly marked as non-negative, but it will be so anyway).

4.3.3. Removing Classes

So far, we have assumed that we have a general schedule of classes for which students can register. However, we can imagine a different case—for instance we might want to remove some classes. Let us say that "Calculus" has three classes and each of them allows at most 20 students to attend. Let us consider what will happen if there will be only 30 people registered for calculus—we need only 30 places but we have room for 60 people. Since we will probably want to have groups with similar sizes, we want to remove one instance of the class from the final schedule, but we want to find the best way of doing so. In other words, we will try to add constraints which will force the solver to remove (the most unwanted) classes from the solution. But there is a catch—what if we have also added a lower bound for some classes? When a class is removed, we cannot satisfy the lower bound requirement for that class, so we need to find another way to represent our needs.

First, we need to add a boolean variable $r_{j,k}$ meaning that class $K_{j,k}$ is removed. Let us assume that $\#remove\left(C_j\right)$ denotes the total number of classes of course C_j which need to be removed (of course $\#remove\left(C_j\right) < \#classes\left(C_j\right)$. The constraint forcing the solver to remove classes is:

$$\underset{j=1,\ldots,\#courses}{\forall} \sum_{k=1}^{\#classes(C_j)} r_{j,k} = \#remove\left(C_j\right)$$

This formula says that the sum of variables representing classes' removal must be equal to the total number of classes which should be removed. Now we need to change our constraint for lower bound on the number of people assigned to a class to the following:

$$\underset{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#classes(C_j)}}{\forall} \sum_{i=1}^{\#students} x_{i,j,k} \geq \left(1 - r_{j,k}\right) \cdot \#min\left(K_{j,k}\right)$$

We can do this, because $\#min\,(K_{j,k})$ is a constant value, so we can multiply it easily by variable $r_{j,k}$. So how does it work? When $r_{j,k}$ is 1 (which means that class $K_{j,k}$ is removed), lower bound on the number of people count assigned to that class cannot be less than $(1 - r_{j,k}) \cdot \#min\,(K_{j,k}) = (1-1) \cdot \#min\,(K_{j,k}) = 0$, so it can by any non-negative integer. On the other hand, when $r_{j,k}$ is equal to 0, it works just like before.
So, now we can remove a class and not bother with its participation lower bound. But we still can assign students to removed classes. We need to change our formula for room capacity, but we can do it in the same way:

$$\forall_{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#classes(C_j)}} \sum_{i=1}^{\#students} x_{i,j,k} \leq (1 - r_{j,k}) \cdot \#max\,(K_{j,k})$$

As we can see, it works in the same way as the previous constraint.

4.4. Cost Function

So far, we have defined constraints which make our final solution "feasible". The students will likely not be very happy with their schedules, but they will at least be able to attend all the classes they want to. It is time to find a way to make them happy.
There is a question—what does it mean to be “happy”? As we will shortly see, there are at least few ways to calculate the level of happiness and each of them gives different results. It is not easy to compare two solutions because we always need to remember that even an optimal solution does not mean that students are completely happy, because it depends on the method of calculating happiness.
Before we describe the methods of calculating happiness, we need to consider a few details of preference points. Every student can make his decision by distributing preference points for classes in the courses. We assume, that the student cannot have more than 20 preferences points in a single course—which means, that $\forall_{\substack{i=1,\ldots,\#students \\ j=1,\ldots,\#courses}} \sum_{k=1}^{\#classes(C_j)} preference\,(S_i, K_{j,k}) \leq 20$. This also means that most of the students' wishes $preference\,(S_i, K_{j,k})$ will be equal to 0.

4.4.1. Sum of Preference Points

The simplest method of calculating happiness is by adding all the gained preference points. As we have defined in Section 4.1, when variable $x_{i,j,k}$ is equal to 1, we say that student S_i gained points for his wish for class $K_{j,k}$. We define our goal function as a sum of all these points:

$$c = \sum_{i=0}^{\#students} \sum_{j=0}^{\#courses} \sum_{k=0}^{\#classes(C_j)} x_{i,j,k} \cdot preference\,(S_i, K_{j,k})$$

By maximizing this function, we increase the total happiness of students, because the solver will try to assign them to classes they want to be assigned to. This is called the “Utilitarian Approach”.

Let us consider advantages and disadvantages of this approach. This formula is easy in implementation and can be calculated very quickly. Perhaps even more importantly, it is rather easy to answer the question "how should students distribute their preference points", because the total value of the cost function is simply the sum of students' preferences. So when two students decide to give preference points for a particular class, we can suspect that the student who gave more preference points for that class will be assigned to it. This is an advantage and a disadvantage at the same time.

Let us consider two students, who are registered for calculus exercises. Imagine that student A gave 10 points for class C, and student B gave 5 points for that class. In an isolated case, when there is the last place left in class C, the solver should assign student A, because he gave 10 points, so he will increase the cost function by 10, whereas student B would increase it only by 5. Let us assume that student B gave no more than 5 points for any of the classes. In this distribution of points, he is in worse position (from the solver's point of view), because he cannot increase the cost function by more than 5 points from one class, so in direct confrontation with student A he will always lose. However, placing student B in that class would mean giving him his most wanted class (because it is his highest preference points decision).

Whether this situation is advantageous or disadvantageous cannot be simply answered. It is a matter of choice or particular needs. Let us now consider different methods of calculating happiness, which do not suffer from this confrontation effect.

4.4.2. Sum of Preferences per Student

This method is very similar to the previous one. We once again sum all of the preference points, but before summing them, we divide them by the maximum amount of the preference points which students can gain.

To explain this, let us consider an example. Imagine that calculus course has two classes: class M (which happens on Monday) and class F (which happens on Friday). We have two students: student A gives 5 points for class M and 3 points for class F, student B gives 10 points for class M and 8 points for class F. We need to assign each of them to exactly one class of calculus per week, which means, that every student will be assigned to class M or to class F, and not to both of them. In this case, the total number of points which student A can gain is 5, because he can gain 5 points for class M or 3 points for class F, and he cannot gain more. Student B can gain at most 10 points for his wish for class M.

To calculate the cost function, we sum the students' gained preference points, but we divide them by the total amount of points they can gain for all their courses. In our example, we divide student A's points by 5, and student's B points by 10. Let us denote the total possible amount of points which student S_i can gain as $\#maxPossiblePoints\,(S_i)$, the general cost function formula is as follows:

$$c = \sum_{i=0}^{\#students} \sum_{j=0}^{\#courses} \sum_{k=0}^{\#classes(C_j)} \frac{x_{i,j,k} \cdot preference\,(S_i, K_{j,k})}{\#maxPossiblePoints\,(S_i)}$$

With this approach we treat students' preferences as their relative needs. 10 points no longer means the most wanted class, student can indicate that, for instance, all of the classes are equally desired (by giving the same amount of points to all of them).
We can consider this approach even further, by dividing student's points by the total possible amount of points which the student can gain in just one course. As with the previous method, this is a matter of particular needs.

4.4.3. Teams

Sometimes students need to work in teams during the classes. We would like to allow students to express their preferences as to what teams they would like to have—we can ask them to prepare a list of groups they would like to belong to, and then use it to create the schedule in which the students from the same teams are assigned to the same classes.
We need to add constraints which in some way tell the solver that it should try to keep students from the same team together. On the other hand, we cannot add this requirement as a simple constraint, because than we could end up with constraints that cannot be fulfilled (for instance imagine two classes with 7 places and 5 places respectively, and two teams, each consisting of 6 people). Our only way to express something which "we highly would like to occur" and at the same time "we do not require" is to add it to the cost function.
To sum up, we would like to be able to tell the solver that some groups of students should be assigned together, if it is possible. We would also prefer to keep as many students together as possible, when it is not possible to preserve the whole team together, so we prefer when solver assigns together 5 out of 6 students and there is one student assigned to a different class, rather than assigning them as if there was no team, when solver decides that it is not possible to preserve the team.

To achieve the described feature, for each class of each course and for each pair of students from the same team we create a boolean variable saying whether these two students are assigned to the same class or not. Next, we simply sum all of these variables and multiply them by some coefficient w_t, whose meaning we will consider shortly. Let us denote the team for course C_j consisting of $\#studentsCount\left(T_i\right)$ students by T_i, defined as $\left\{C_j, \left\{S_1, S_2, \ldots, S_{\#studentsCount(T_i)}\right\}\right\}$ (the course and the set of students belonging to the team). Let us also define the function $course\left(T_i\right)$ indicating the course, for which the team T_i is created. We define the following variables:

$$\forall_{\substack{t=1,\ldots,\#teams \\ j=1,\ldots,\#studentsCount(T_t) \\ k=1,\ldots,\#studentsCount(T_t) \\ j\neq k}} \; \forall_{l=1,\ldots,\#classes\left(C_{course(T_t)}\right)} \; t_{S_j,S_k,K_{course(T_t),l}} = x_{j,course(T_t),l} \wedge x_{k,course(T_t),l}$$

Every such variable is equal to one only when both of the students are assigned to the same class. We can now add these variables to the cost function with the following formula:

$$c = \sum_{\substack{t=1,\ldots,\#teams \\ j=1,\ldots,\#studentsCount(T_t) \\ k=1,\ldots,\#studentsCount(T_t) \\ j \neq k \\ l=1,\ldots,\#classes\left(C_{course(T_t)}\right)}} t_{S_j,S_k,K_{course(T_t),l}} \cdot w_t$$

We now consider the meaning of w_t. We cannot simply add the described variables to the cost function, because their total value is relatively small. Thus, we need to multiply them by some constant value, which we will call the "team coefficient". The value of w_t can be chosen arbitrarily, the greater it is, the more valuable the teams are, so we can control the solver decisions (whether to preserve the teams or not) by changing the value of the coefficient. Of course, we need to decide whether we prefer to have the most suitable placement of students (according to their wishes) or to keep as many teams as possible, probably with lower total happiness, as defined in the previous section.

4.4.4. Continuous Schedule

Some students do not care which classes they get and all they would like is to have a continuous schedule, which means that their classes occur one after another. It is so because students are working and they prefer to study in the evenings, or they live far away from the university and they need to travel by bus for a few hours every day. In this case having one class, then one hour break, and another class is inconvenient, because such a student cannot go home and rest or go to work. We can express our desire that the schedule should consist of classes which are "co-occurring".

Let us assume that we have a list of pairs of co-occurring classes. By "co-occurring" we mean situation, when one class starts right after another one ends. For each pair of such classes and for each student we create a boolean variable indicating whether the student is assigned to both of the classes. We define the following variables:

$$\forall_{i=1,\ldots,\#students} \; \forall_{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#courses}} \; \forall_{\substack{l=1,\ldots,\#classes(C_j) \\ m=1,\ldots,\#classes(C_k) \\ (j,l)\neq(k,m)}} c_{S_i,K_{j,l},K_{k,m}} = x_{i,j,l} \wedge x_{i,k,m}$$

When variable $c_{S_i,K_{j,l},K_{k,m}}$ is equal to one, the student is assigned to two classes occurring one after another. We can now sum all these variables, multiply them by coefficient w_c (whose meaning is analogous to the coefficient w_t described in the previous section), and add to the cost function:

$$c = \sum_{i=1,\ldots,\#students} \; \sum_{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#courses}} \; \sum_{\substack{l=1,\ldots,\#classes(C_j) \\ m=1,\ldots,\#classes(C_k) \\ (j,l)\neq(k,m)}} c_{S_i,K_{j,l},K_{k,m}} \cdot w_c$$

By changing the value of w_c, we can change the behaviour of the solver—whether we prefer the more continuous schedules or schedules more going on with students' preference points.

4.4.5. Day Off

We now consider students who prefer to have as many days off as possible. They prefer to have classes from morning to evening on one day of a week, provided that on all the other days of the week they will not need to attend the university. We can express this idea using the same reasoning as in the previous section—we create variables indicating whether each student is assigned to classes occurring on the particular day, and then we add these variables to the cost function with some coefficient.

We define the following variables:

$$\bigvee_{d=\{\text{monday,tuesday,wednesday,thursday,friday}\}} \bigvee_{i=1,\ldots,\#students} f_{S_i,d} = \bigvee_{\substack{j=1,\ldots,\#courses \\ k=1,\ldots,\#classes(C_j) \\ K_{j,k} \text{ happens on day } d}} x_{i,j,k}$$

When variable $f_{S_i,d}$ is equal to 1, student S_i is assigned to some classes on day d. To tell the solver that we want to have as many days off as possible, we need to negate this variable and add it to the cost function:

$$c = \bigvee_{d=\{\text{monday,tuesday,wednesday,thursday,friday}\}} \sum_{i=1,\ldots,\#students} f_{S_i,d} \cdot w_f$$

Once again, w_f is the coefficient indicating how much we prefer to end up with a schedule with many days off, or with the schedule with classes assigned as much according to the students' other preferences as possible.

4.4.6. Students' Coefficients

In previous sections we were multiplying particular variables (corresponding to the teams, continuous schedule or days off) by global coefficients, which were arbitrarily chosen. To give the students even greater possibility to direct the solver, we can ask them to list their "global preferences", indicating whether each particular student wants to have continuous schedule or many days off. When we have such global preferences, we can multiply described variables not only by our global coefficients, but also by global coefficients of each particular student, in a natural way.

4.4.7. Final Form of the Cost Function

To create the final form of the cost function, we can choose from the elements described in the previous sections. By manipulating the coefficients, we can affect the final solution created by the solver. In fact, we can calculate many different schedules for different sets of coefficients and then try to compare them—remembering that it cannot be done easily (if it would, we could just change the cost function and ask the solver to find the optimal solution). Characteristics of the particular schedule depend on the preference points distribution, global students' preferences, density of classes, and many more factors, so sometimes the human decision is necessary to choose "the best" schedule in the most intuitive meaning of this phrase.

5. Environment and Implementation

In this chapter we describe the actual implementation of the system generating the ILP program (by which we understand the set of all the constraints described in previous chapter) and solving it using one of several popular solvers. The system (called "Mrowki System" in the rest of this document) has been successfully used by the students of computer science at the Department of Computer Science of AGH University of Science and Technology in years 2013-2014, so it is not only based on strong theoretical basis, but also has been tested in real-life settings.

We first describe the environment and the parts of the Mrowki System, and then we present the results of tests and calculations performed with both real-life test cases and synthetic ones.

5.1. System Components

The Mrowki System consists of the following components:

- Input Parser and Validator - written mainly in Python
- ILP Program Generator - written in C#
- MILP Solver

The first component parses the input files describing the students' preferences and the general timetable prepared by the authorities of the university, and performs necessary validations. The second component generates the ILP program (creates the constraints described in the previous chapter and generates the MPS file describing the program). The last component is a specialized MILP Solver, such as Gurobi, SCIP or CPLEX.

5.1.1. Input Parser and Validator

This component is written in Python 2.7.3 with PLY library. It consists of several scripts parsing the input files, validating them, and calculating some statistics (e.g., the number of students, the number of courses etc.). There are three input files: the first one describes the timetable (courses, co-occurring and colliding classes), the second one describes the teams, and the last one describes the students' preferences. Each file has textual, space-separated format, which can be parsed easily.

After reading the files, the scripts validate them and try to find the most typical errors (too many people registered for a course, preferences for not existing classes etc.). In fact, finding all of such issues is almost impossible without solving the problem, because, for instance, the program might be not solvable due to too many blocked classes or due to relationships between the classes that cannot be detected easily.

5.1.2. ILP Program Generator

The main component of the Mrowki System is written in C# 5.0 with .NET 4.5.1 and Microsoft Solver Foundation (MSF) library. The MSF library is used to model different optimization problems—linear, integer, quadratic and many more—which can then be solved by the built-in solver or by external tools.
The component reads the parsed files, calculates all the necessary constraints, creates the ILP program and saves it to an MPS file. MPS stands for "Mathematical Programming System" and it is a file format for storing mixed integer linear programming problems, supported by most of today's popular solvers.
The fact, that the generated program is being solved by external solvers is a huge advantage—we do not need to develop and maintain algorithms for solving ILP programs, but we can use the best available software. When we need high performance, we can use highly scalable solvers and run them on super computers, but we can also use any freely available solver and solve the program using personal machines. This also allows us to perform tests and compare the solvers on how they manage to solve our problem.

5.1.3. MILP Solver

The last part of the Mrowki System is an ILP solver, which is used to solve the program generated by the previous component. Because the program is archived in the well-supported MPS format, we can choose among many different applications, according to our particular needs. We use Gurobi most of the time, because it is efficient, highly scalable, and free for academic use. We also performed the tests with other solvers—CPLEX from IBM and non-commercial SCIP. If we do not need (or want) to run the solver on our computer, we can also use the NEOS Server [9, 14], which is a free Internet-based service. While performing the tests, we have been using the following solvers:

- Gurobi 5.6 for Windows
- SCIP 3.0.1 x64 for Windows
- CPLEX 12.6 x64 for Windows

5.1.4. Parameters

The cost function can be generated with respect to the three global parameters, which are: the coefficient for teams, the coefficient for obtaining a continuous schedule, and the coefficient for having

days off. Each of these values can be configured manually, before generating the MPS file. Furthermore, every student can express his global preferences by providing three values: the coefficient for accounting his preferences points, the coefficient for preferring continuity of the schedule, and the coefficient for preferring days off. These three coefficients can have integer values not greater than 100, and all need to sum up to 100.

5.1.5. Solver's Precision

Every solver has a parameter describing its precision in finding the optimal solution. Such a parameter may have a great influence on the process of finding a solution.

Gurobi

During the tests, we were changing the MIPGap parameter which stands for relative MIP optimality gap. Its default value is $1e^{-4}$. By setting this parameter, the MIP solver will terminate (with an optimal result) when the relative gap between the lower and upper objective bound is less than MIPGap time upper bound.

SCIP

During the tests, we were changing the limits/gap parameter. Its default value is equal to 0. Assuming that $primal$ is the current solution of the primal problem and the $dual$ is the current solution of the dual problem (as described in [21]), solving stops if the relative gap equal to $\frac{|primal - dual|}{\min(|dual|, |primal|)}$ is below the given value.

CPLEX

During the tests, we were changing the mip tolerances mipgap parameter. Its default value is equal to $1e^{-4}$. This parameter sets a relative tolerance on the gap between the best integer objective and the objective of the best node remaining.

5.2. Data Sets

The Mrowki System was used to solve several data sets obtained from real-life situations. These data sets represent preferences created by the computer science students of AGH University of Science and Technology, throughout their registrations for courses in years 2013-2014. There are 10 data sets:

Id	Courses	Classes	Pairs of colliding classes	Students	Teams
1	8	55	79	103	99
2	11	60	120	109	76
3	10	77	122	135	61
4	6	52	60	105	100
5	7	56	108	127	62
6	8	67	193	129	32
7	6	15	10	179	0
8	14	56	111	95	17
9	11	98	350	142	61
10	9	81	139	112	75

Each of these data set consists of real data—timetable, students' preferences and teams. These data sets were also used in tests described in the next sections.

5.3. Test Cases

We performed a few types of tests to examine the behaviour of the developed system. Each test was performed on a machine with two virtual cores AMD Opteron Processor 4171 HE, 3.50 GB RAM and running Windows Server 2012 x64.

5.3.1. Tests with a Varying Number of Students

We have performed tests with a varying number of students. We accounted the following elements:

1. The number of students ranging from 1 to the maximum number of students in a given dataset, with step equal to 5. The students were selected in order of appearance in real-life data set, for instance, number of students equal to twenty means that we selected first twenty students from the data set.
2. Three types of students' preferences: the real ones, random ones, and all preference points equal to 10.
3. Three sets of coefficients:
   (a) 1 for points, 25 for teams, 0 for continuous schedule, 0 for days off.
   (b) 1 for points, 25 for teams, 10 for continuous schedule, 5 for days off.
   (c) 1 for points, 25 for teams, 10 for continuous schedule, 25 for days off.
4. Solver's gap set to $1e^{-1}$, $1e^{-2}$ and $1e^{-3}$.
5. Three data sets: set 1, set 2 and set 3 from the table 5.2.

6. Three solvers: Gurobi, SCIP and CPLEX.

Every test was performed ten times and the running times were averaged. Tests for which solving time was greater than one hour were discarded from the charts. Here we include only a part of the obtained results.

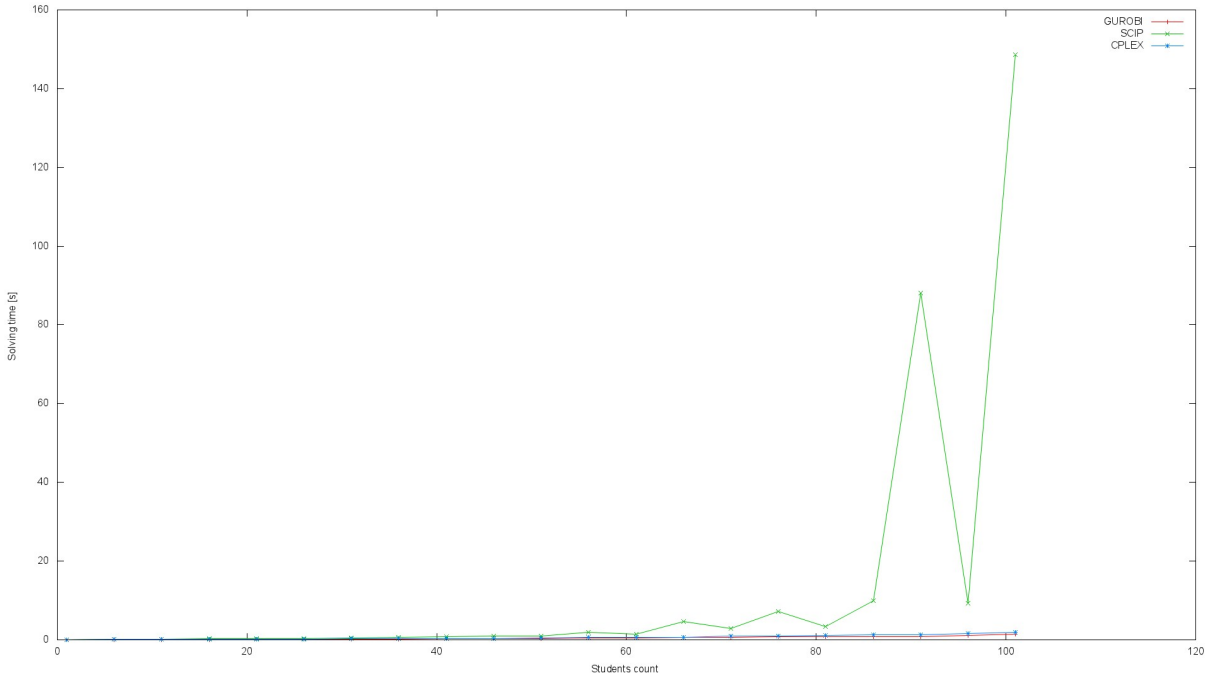

Figure 5.1: Solving time per solver, for different number of students (data set 1, coefficients set 1, gap $1e^{-3}$, all preferences equal)

We can see, that the set of coefficients has a great influence on the solving time. For instance, all solvers were able to find the optimal solution with the gap set to $1e^{-3}$ when the objective function included only preferences points and teams (see Figure 5.1), but calculating the solution with objective function including continuous schedule and days off using the SCIP solver was possible only for fewer than 30 students, as we can see on Figure 5.2. We can see that including more pieces of the objective function results in longer solving time. By comparing these results we can infer that the last set of coefficients is the hardest one (meaning the longest solving time) and the first set of coefficients is the easiest one.

Different types of students' preferences resulted in very different calculation times, but it is not clear whether there are some clear relationships. For instance, Gurobi performed best for all preferences equal (as shown on Figure 5.3), on the other hand, all preferences equal caused the longest calculation using SCIP (as we can see on Figure 5.4).

We can also see that different data sets result in very different calculation times. For instance, data set 3 was rather easy (all the solvers calculated the optimal solution for the smallest gap and the hardest

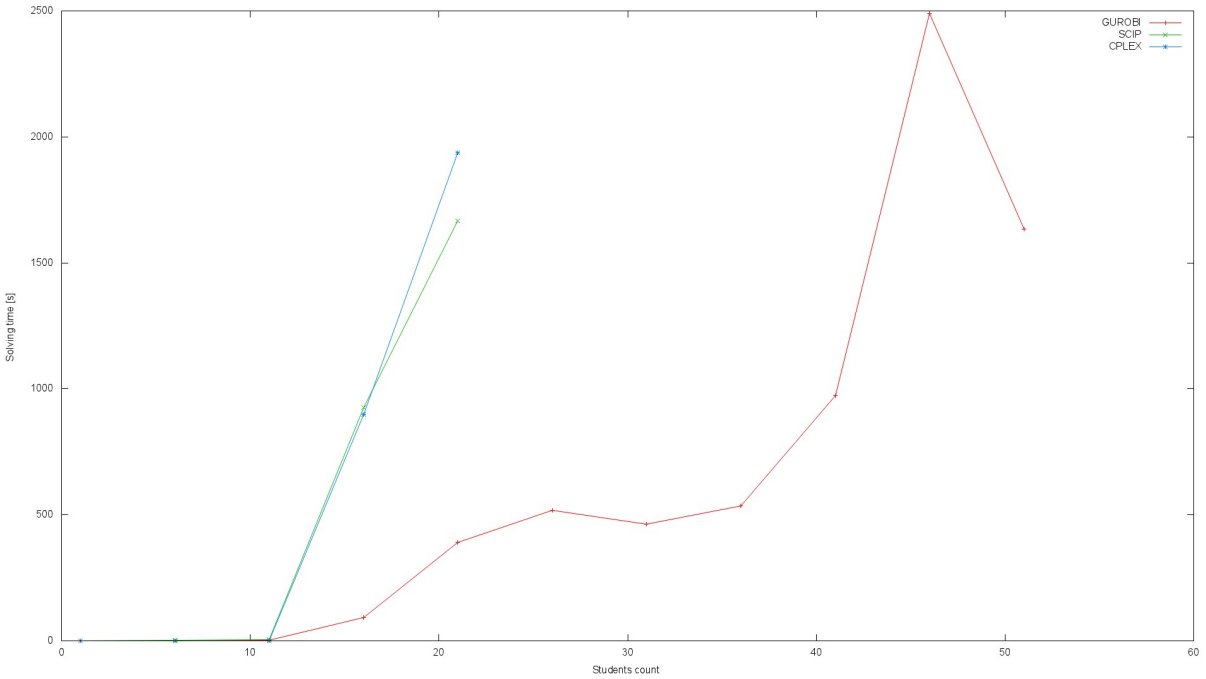

Figure 5.2: Solving time per solver, for different number of students (data set 1, coefficients set 3, gap $1e^{-3}$, all preferences equal)

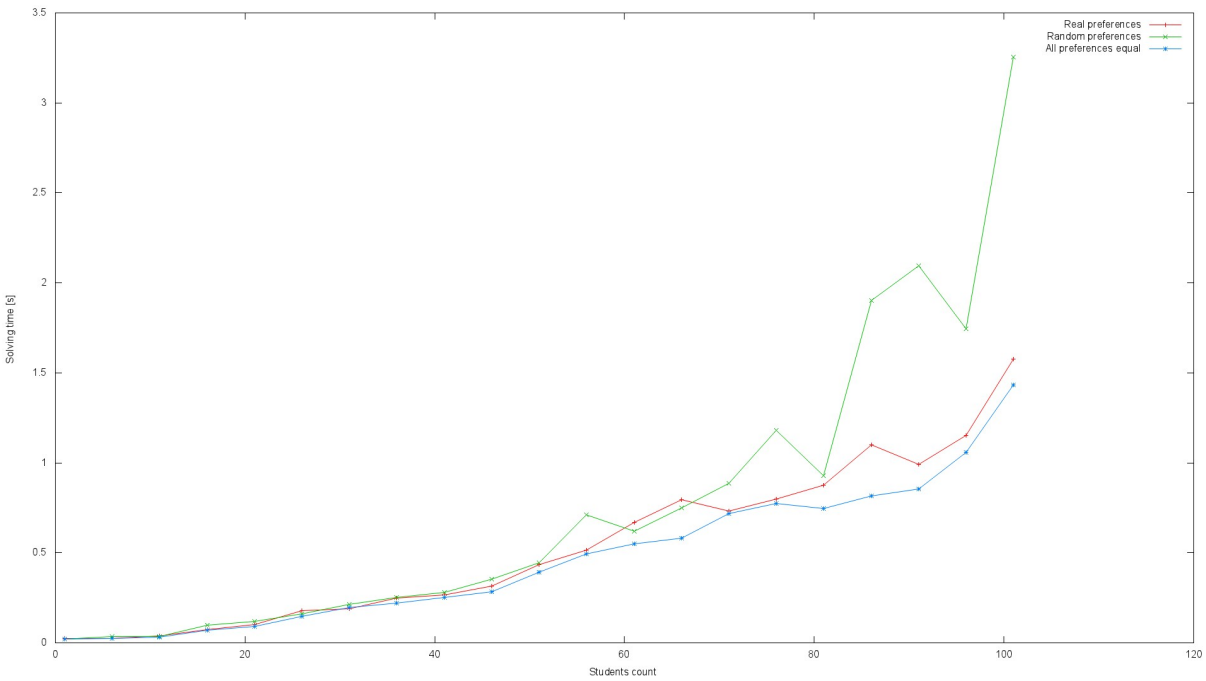

Figure 5.3: Solving time per different types of preferences, for different number of students (data set 1, coefficients set 1, gap $1e^{-3}$, Gurobi)

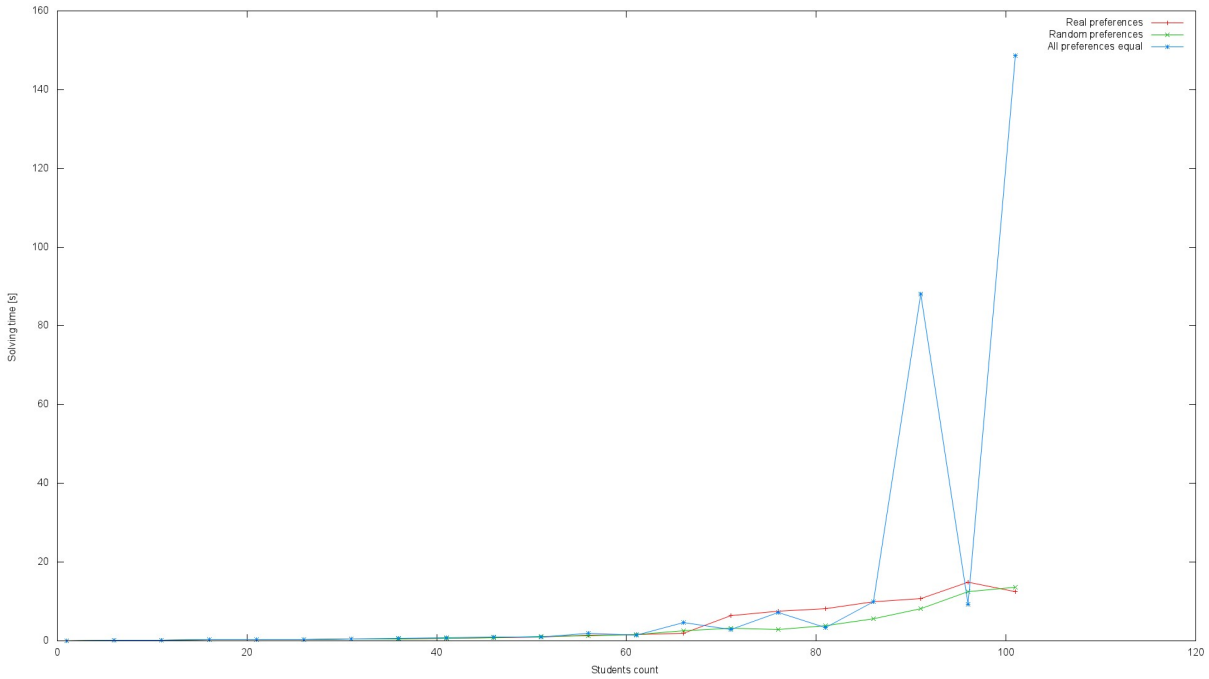

Figure 5.4: Solving time per different types of preferences, for different number of students (data set 1, coefficients set 1, gap $1e^{-3}$, SCIP)

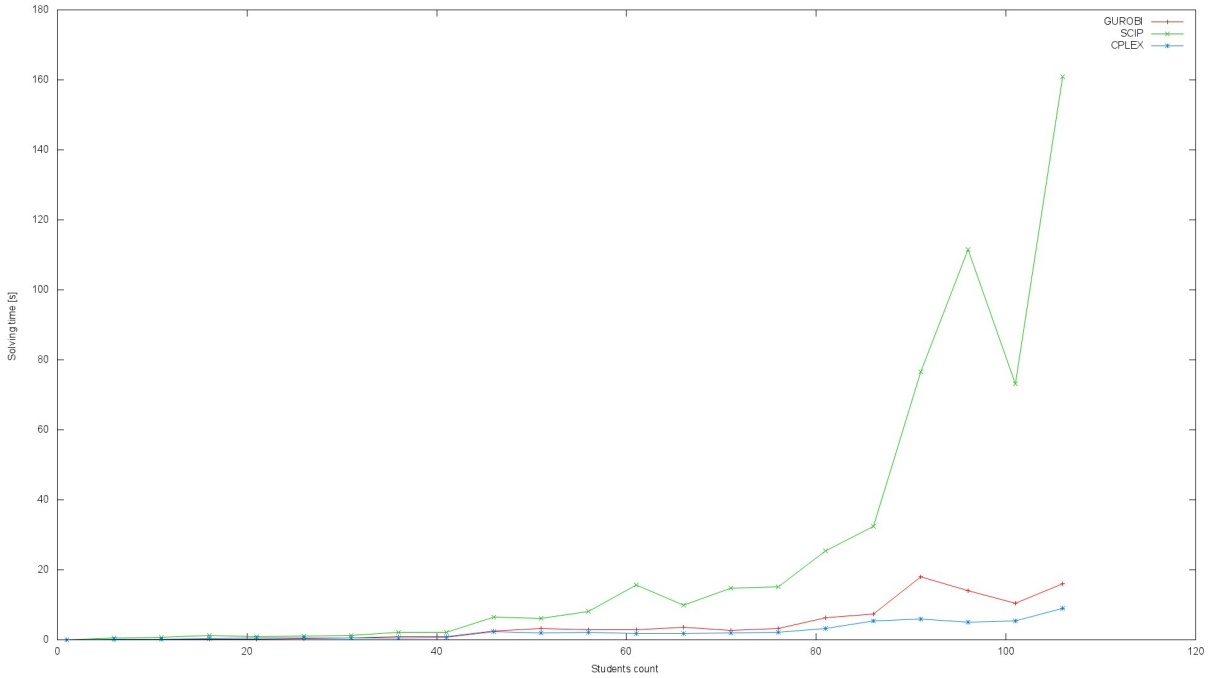

Figure 5.5: Solving time per solver, for different numbers of students (data set 3, coefficients set 3, gap $1e^{-3}$, real preferences)

coefficients, as shown on Figure 5.5), whereas data set 1 was much more difficult. Calculation times for data set 3 were lower than 5 minutes, whereas SCIP didn't manage to calculate results for 30 students for data set 1 (see Figure 5.2, remember that tests longer than one hour were discarded).

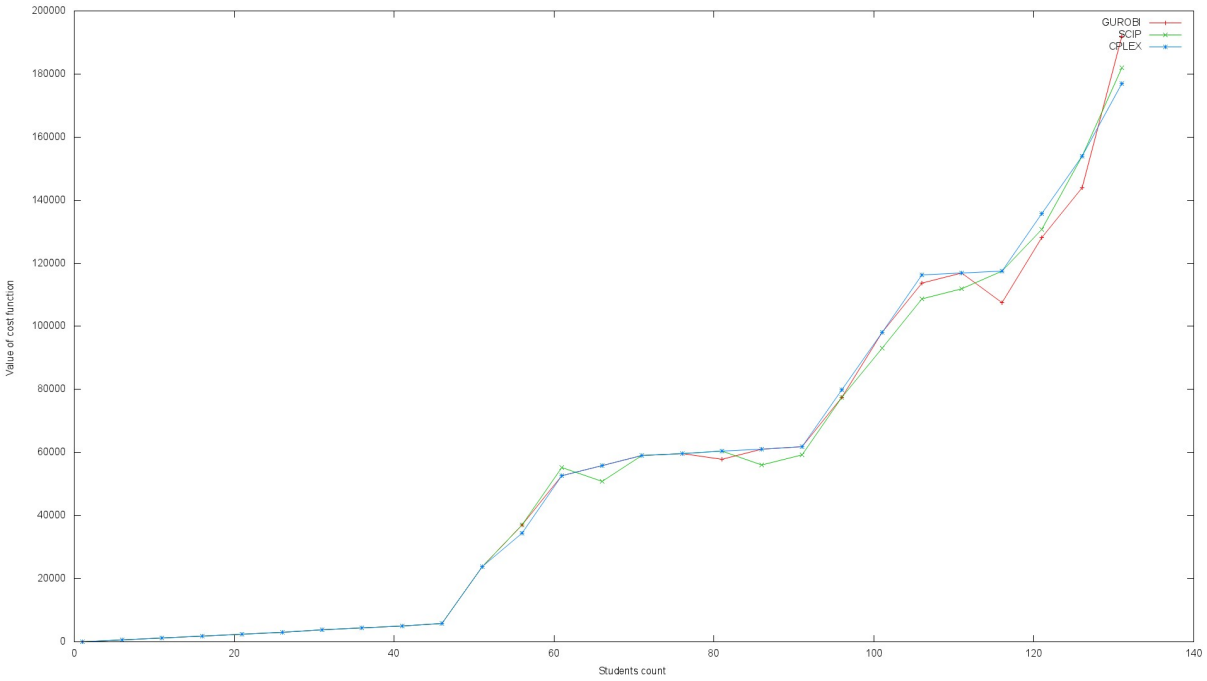

Figure 5.6: Objective function value per solver, for different numbers of students (data set 2, coefficients set 1, gap $1e^{-1}$, real preferences)

When it comes to the achieved value of the objective function, all solvers give similar results. The biggest difference are when the gap is equal to $1e^{-1}$ (see Figure 5.6), but setting the smallest gap causes all solvers to return the same result (see Figure 5.7).

From the achieved values, we can deduce that in general the SCIP solver is much slower than the other two solvers. The CPLEX is roughly as fast as Gurobi, however, sometimes CPLEX is as fast as SCIP which means a great slowdown.

5.3.2. Tests with Large Number of Students

We have performed tests for data set 2 for the number of students from 1 up to 5000 using the greatest gap $\left(1e^{-1}\right)$, the real preferences and the first set of coefficients (only points and teams). The students were selected in order of appearance in the real-life data set and were replicated when necessary. For instance, number of students equal to 300 means that there were three copies of each of the first eighty two students and two copies of each of the students between eighty three and one hundred nine in the data set. Every test was performed ten times and the running times were averaged. Tests for which

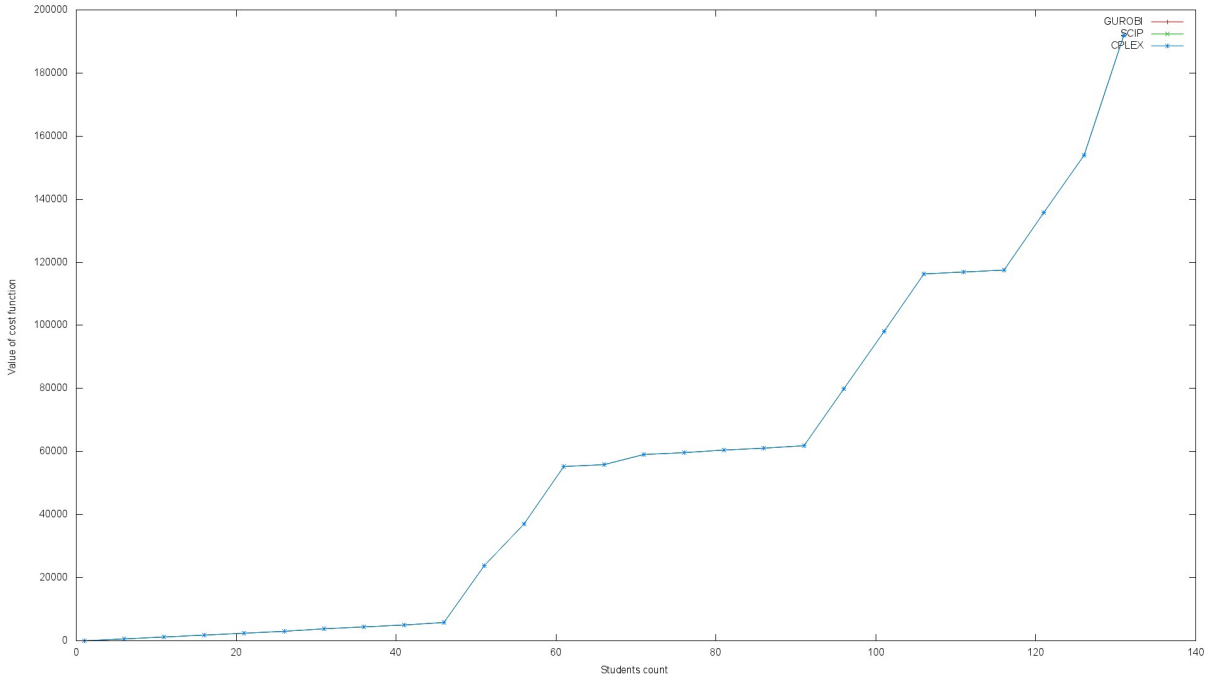

Figure 5.7: Objective function value per solver, for different numbers of students (data set 2, coefficients set 1, gap $1e^{-3}$, real preferences)

solving time was greater than one hour were discarded from the charts. Here we include only a part of the obtained results.

We can see that all the solvers give almost the same results (see Figure 5.8) and that the SCIP solver is much slower than the other two solvers (as shown on Figure 5.9).

5.3.3. Tests with Different Numbers of Courses

We have performed tests with different numbers of courses, using data set 2, using gaps $1e^{-1}$, $1e^{-2}$ and $1e^{-3}$, using three sets of coefficients (same as described in Section 5.3.1) and real preferences. The courses were selected in order of appearance in the real-life data set. Every test was performed ten times and the running times were averaged. Tests for which solving time was greater than one hour were discarded from the charts. Here we include only a part of the obtained results.

Figure 5.10 shows that with increasing number of courses the solving time also increases. We can also observe significant slowdown when calculating results for more than 6 courses.

5.4. Summary

Obtained results show that there are big differences between data sets and it is rather difficult to predict how long will the calculations take. It is clear that Gurobi and CPLEX are much faster than SCIP,

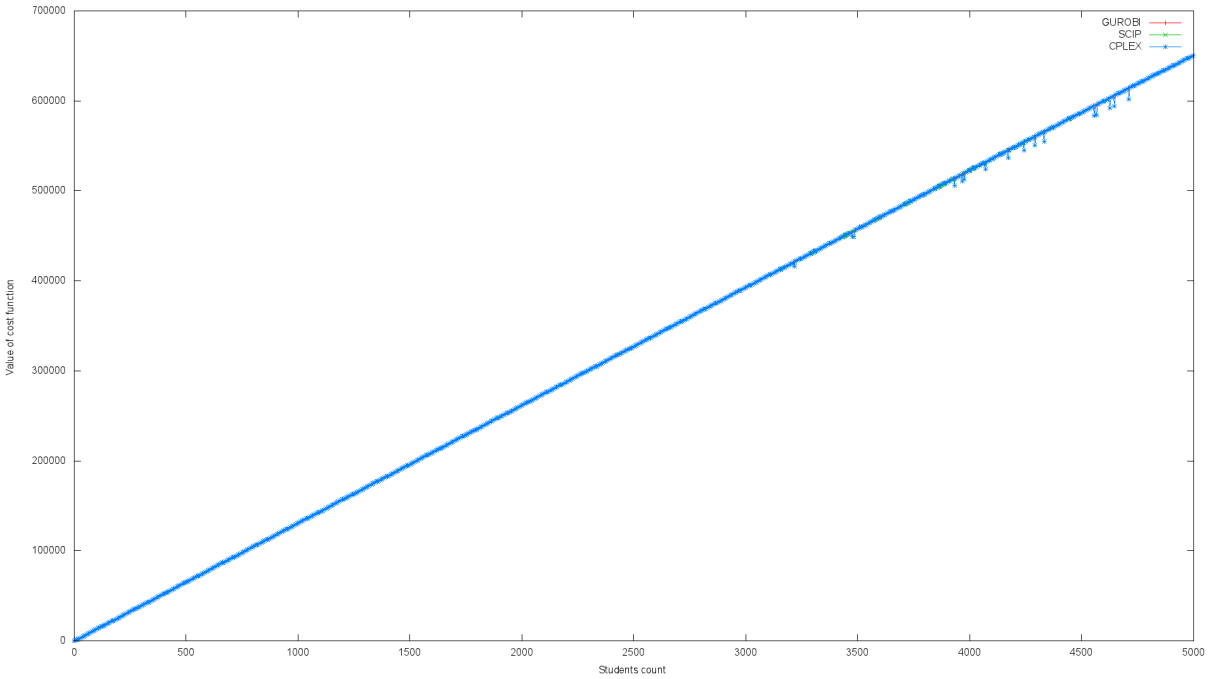

Figure 5.8: Objective function value per solver, for different numbers of students (data set 2, coefficients set 1, gap $1e^{-1}$, real preferences)

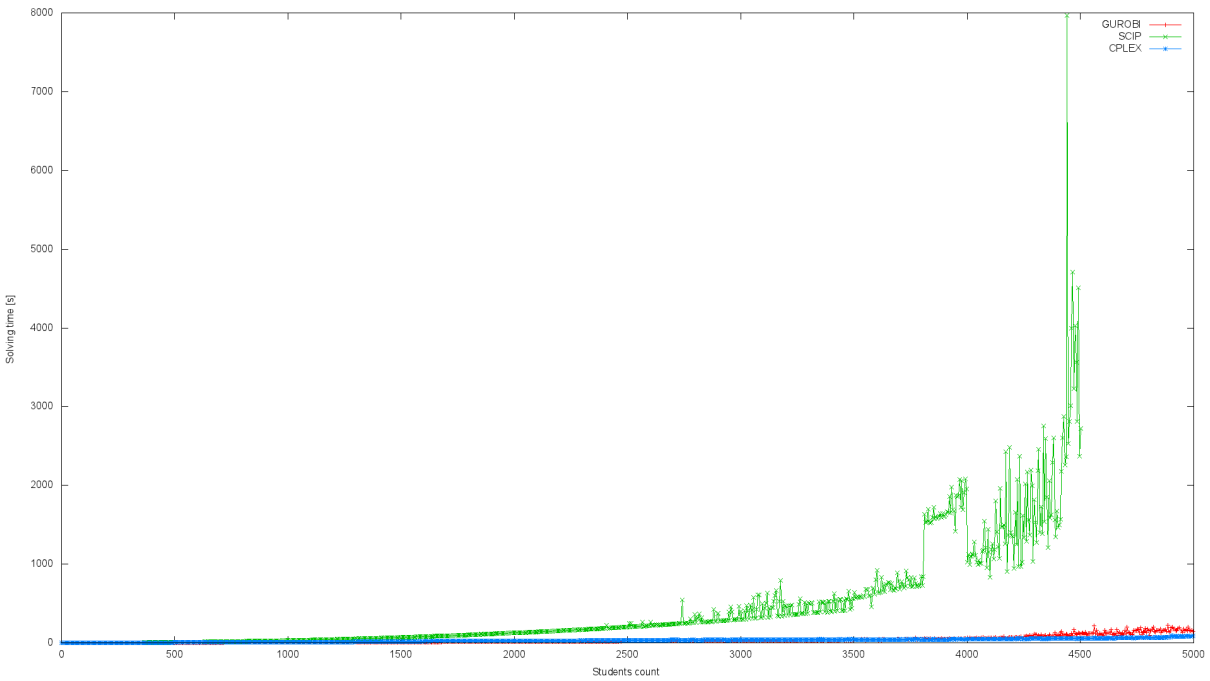

Figure 5.9: Solving time per solver, for different numbers of students (data set 2, coefficients set 1, gap $1e^{-1}$, real preferences)

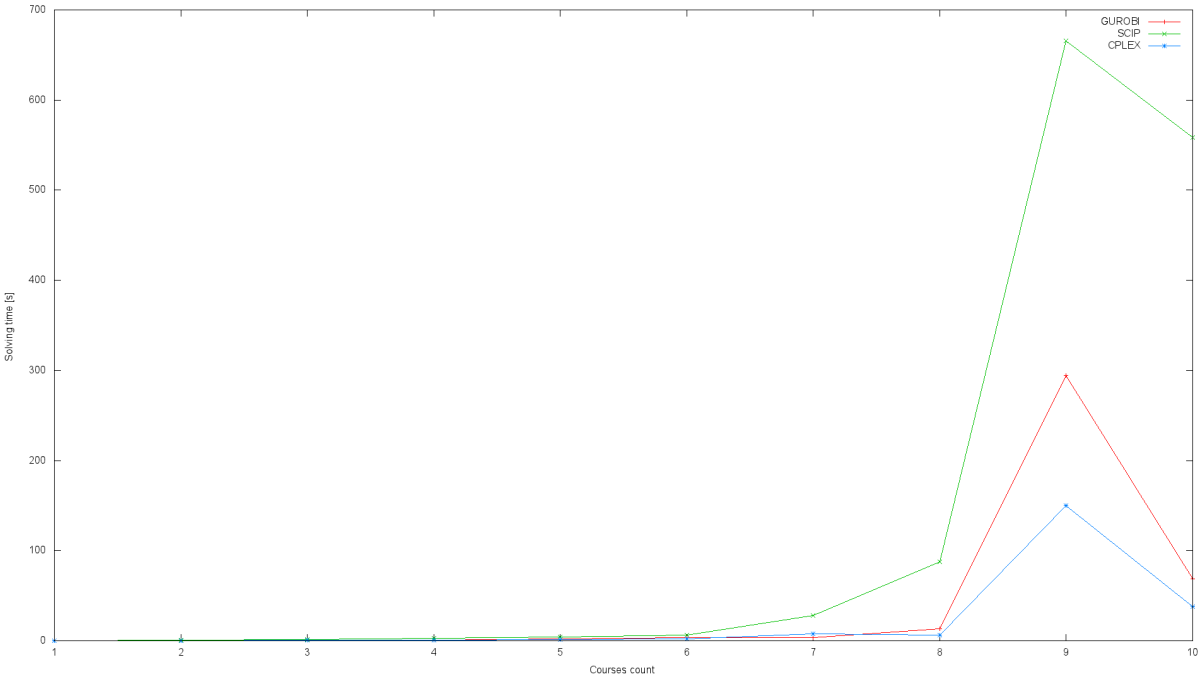

Figure 5.10: Solving time per solver, for different number of courses (data set 2, coefficients set 1, gap $1e^{-3}$, real preferences)

and that these two solvers should be used when possible.

It is not a surprise that accounting for more pieces of the cost function results in longer calculation time. We need to remember that different coefficient values can highly change the characteristic of the final solution and that we should always try to find the optimal solution with the smallest possible gap. Otherwise, accounting for more and more pieces of the objective function will not be reflected in the results.

What is most important, from the obtained results we can deduce that the proposed ILP program is efficient and allows to represent real students' demands very well. Most of the calculations can be performed in less than an hour, so calculating the schedule is fast and can be performed many times for different coefficients in only a few hours. Despite the fact that the ILP problem is NP-complete, the implemented approach gives very good results. Considering the high scalability of modern solvers and easiness of representing new ideas as ILP programs, there are many possible improvements for our system.

6. Possible Extensions

In this chapter we describe possible ways of extending the Mrowki System. We divide them in three categories:

– Building blocks—here we describe development of other mathematical operators

– Schedule constraints—ideas of changing schedule to better suit students' demands

– Happiness calculation—other ways of calculating happiness

6.1. Building Blocks

We can develop new operators but we need to remember that calculating more sophisticated functions might result in adding many temporary variables. We should always try to find a way to simplify the formula in order to keep the program small. Sometimes we should try to calculate something seemingly different but equivalent for our needs—for instance, in order to maximize the standard deviation we do not need to calculate the square root of the variance. Maximizing the variance will result in maximizing the standard deviation.

6.1.1. Truncation

Most of our building blocks work using integer numbers, but sometimes we need to use non-integer values. To perform operations described in Chapter 3, we need to find a way to truncate numbers. It is worth noting that if our variables might have non-integer values, we are no longer using ILP (which stands for Integer LP), but we use Mixed Integer Linear Programming (MILP). Most solvers for ILP in fact regard programs as MILP programs, so we are still able to solve our problems.

In order to truncate a floating-point value, we need to make a decision as to what precision do we need. In other words, how many digits after decimal point we want to preserve. Let us say that we want to have a precision up to 10^{-p}. To perform calculations, we can at the beginning multiply every number by 10^p, truncate it to the nearest integer, solve the problem, and while reading results divide every variable by 10^p.

Let us say that we want the variable a to be a truncated variable x, which means $a = \lfloor x \rfloor$. To achieve this, we use the following constraints:

$$a \leq x$$
$$a + 1 \geq x$$

This formula has one drawback: it is not deterministic when we want to truncate an integer value. For instance, when $x = 2$, a might be equal to either 1 or 2, both of the values satisfy the constraint.

6.1.2. Square Root and Other Functions

We can calculate the square root in a similar way to the division. Let us say that we want the variable a to be the square root of a variable b. We need to use the following constraints:

$$a \cdot a \leq b$$
$$(a + 1) \cdot (a + 1) \geq b$$

These constraints mean that the variable a is the highest possible value whose square is not greater than the value of the variable b.

By using two's complement representation of a variable, we can perform many operations in the same way as a CPU does them. For instance, we can calculate logarithms or exponents. Many bit operations are described in "Bit Twiddling Hacks" [2].

6.1.3. Ordered Weighted Averaging

Ordered Weighted Averaging (OWA) is a class of mean type aggregation operators. Many common mean operations (such as max, min, arithmetic mean) are members of this class. Using this approach we can calculate the happiness in many different ways.

Formally an OWA operator of dimension n is a mapping $F : \mathbb{R}_n \to \mathbb{R}$ with associated collection of weights $W = [w_1, \ldots, w_n]$ where

$$F(a_1, \ldots, a_n) = \sum_{j=1}^{n} w_j b_j$$

where b_j is the j-th largest of the a_is. By choosing different weights we can create different aggregation operators. For instance, $W = [1, 0, 0, \ldots, 0]$ represents maximum and $W = \left[\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right]$ represents arithmetic mean.

To calculate the OWA operator, we need to sort the numbers (using the method described in Section 3.6.2) and calculate the dot product of the happiness vector and the weights vector. Calculating the dot product can be done using the multiplication method described in Section 3.2.1.

We need to remember that sorting entails great expansion of the problem, because it needs lots of variables. Solving such a problem might take a very long time.

6.2. Schedule Constraints

In Chapter 4 we described the following elements:

- Assignment for exactly one class in each course for students, accounting for blocked classes and collisions
- Lower and upper bounds for room capacity, removing classes
- Dividing students into teams

These constraints allow us to easily represent basic requirements and create assignment which allow students to finish their courses. Sometimes we might want to add some constraints which are not required to generate a schedule, but might be useful in general.

6.2.1. Balanced Classes

We might add some constraints to keep classes balanced, which means that the difference between the highest and the lowest number of students assigned to some class is not greater than some value. This requirement is very useful when the lecturer wants to have classes with similar numbers of students.
It is worth noting that to add such a balance constraint we do not need to calculate the absolute value of some variable. For instance, if we want the number of assigned students in class A (denoted as a) to differ from the same number for class B (denoted as b) by no more than 2, we can use the following constraints:

$$-2 \leq a - b \leq 2$$

Using this approach we do not need to calculate the absolute value of $a - b$, which would be slower.

6.2.2. Soft-Blocked Classes

Blocked classes described in Section 4.2.1 might result in no solution. We might want to ignore some of these constraints in order to make the problem solvable.
To achieve this, we need to account for the blocked classes in the cost function. For every rejected blocked class, we decrease the value of the objective function. This can be interpreted as a penalty for breaking a constraint. This approach is described in the book of Williams [24]

6.3. Happiness Calculation

As we have described in Chapter 4, we are able to account for the following elements in the cost function:

– Sum of preferences

– Teams

– Continuous schedule

– Day off

By adding or removing particular pieces of the cost function, or by changing the coefficients, we can easily modify the outcome of the system. We could easily add many other elements for calculating happiness, some of which are described below.

6.3.1. Groups of Classes

In typical situations we increase the cost function when a student is assigned to the classes he gave some points. This is a rather straightforward approach, but it has some drawbacks:

– When a student gives points for three or more classes in a row, solver might assign him to the first and the last classes, which means that the student will have a gap between the classes, but the solver does not consider this as a worse situation than assigning the student to the first and the second class.

– A student might want to let solver know that he wants all the classes from a particular group of classes or else he will not be happy.

We would like to be able to treat some classes as one and get points only if the student is assigned to all of them. For instance, let us say, that Calculus starts on Monday at 8:00 AM and ends at 9:30 AM, and Operating Systems starts on Monday at 9:30 AM and ends at 11:00 AM. We would like to formulate the cost function in a way that it will increase only when the student is assigned to the Calculus and to the Operating Systems classes. To achieve this, we need to calculate the conjunction of variables representing student's assignment to the Calculus and the Operating Systems and then use it in the cost function.

6.3.2. Points for Lecturers

Sometimes a student does not care when he has classes, he is only interested in who is the teacher. With this approach, we increase the cost function only when the student is assigned to the classes with a specified professor.
To achieve this, we need to calculate the disjunction of all the variables representing classes with the particular lecturer and use this variable in the cost function.

6.3.3. Lower Bound of Happiness

We might want to find a schedule with lower sum of preference points, but with greater lowest sum of preference points for a student. This egalitarian approach might result in a lower average happiness, but at the same time the standard deviation might be lower and the solution can be viewed as more fair.

6.3.4. Best Time for Classes

Sometimes a student can attend classes only in particular hours, because he is working full-time. In that case, we want to increase the cost function only when he is assigned to the classes occurring in the time slots he specified.
To achieve this, we may calculate the sum of variables representing classes occurring in time slots specified by the student and use these variables in the cost function.

6.3.5. Non-Continuous Classes and No Days Off

Some students do not like having many classes in a row because they get tired and weary. In that case, we would like to assign them to classes every day and with many gaps between them. This can be done using a similar approach as the one used in Chapter 4 for the opposite goals; we only need to increase the cost function when student is not assigned for continuous terms or has no day off.

6.3.6. Generating Timetable From Scratch

In theory, using approaches described in Section 6.3.2 and Section 6.3.4, we would be able to generate the whole timetable, not only assignment to particular classes. This would need a major redesign of the system, because we would not receive the timetable from the authorities of the university and we would need to create it on our own. In fact, this would, in some way, change the problem from assigning students to assigning lecturers, and would need calculating many things (such as collisions for professors) and great cooperation between students and the authorities of the college. On the other hand, accounting students' and lecturers' preferences in timetable generation could be very beneficial.

7. Conclusion

In this thesis we have presented an analysis of Students Placement Problem. We have described the main requirements and key concepts allowing to represent the students' preferences in a concise and readable way. We have also presented the main parts of ILP and its possibilities. Finally, we have shown implementation of the system designed for solving Students Placement Problem using Integer Linear Programming.

We have seen that ILP allows to express constraints in a readable form and gives opportunity to create more sophisticated requirements using simple building blocks. This allows us to represent real-life relationships and solve problems representing day-to-day tasks. We are not limited by the narrow range of operators usable in ILP because (as we have seen) it is not difficult to define common mathematical functions and operators using simple methods.

We have presented the Students Placement Problem in details. We have shown its practical usage and the steps necessary for defining and solving instances of the problem. We have also shown that there are many possible extensions allowing to define even more cost functions to better represent particular university's demands.

We have also shown an actual implementation of the system created for constructing and solving the Students Placement Problem. By comparing different solvers we have shown that the problem can be effectively solved using contemporary personal computers in reasonably short time.

There is a place for further extensions to the problem and we hope that with the increase of the computers' power there will be an option to define even more complex cost functions which will be able to better represent the "happiness" of the ordinary student. By including more and more factors students will be able to define even better what a good assignment is.

Bibliography

[1] AKKOYUNLU, E. A linear algorithm for computing the optimum university timetable. *The Computer Journal*.

[2] ANDERSON, S. E. Bit twiddling hacks.

[3] ANITESCU, M., AND POTRA, F. A. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics 14*.

[4] BADRI, M., DAVIS, D., DAVIS, D., AND HOLLINGSWORTH, J. A multi-objective course scheduling model: Combining faculty preferences for courses and times. *Computers and Operations Research*.

[5] BOHANNON, J. M. A linear programming model for optimum development of multi-reservoir pipeline systems. *Journal of Petroleum Technology 22*.

[6] BRESLAW, J. A linear programming solution to the faculty assignment problem. *Socio-Economic Planning Science*.

[7] COOK, S. A. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*.

[8] COSTA, D. A tabu search algorithm for computing an operational timetable. *European Journal of Operational Research*.

[9] CZYZYK, J., MESNIER, M., AND MORÉ, J. The neos server. *IEEE Journal on Computational Science and Engineering*.

[10] DE WERRA, D. An introduction to timetabling. *European Journal of Operational Research*.

[11] DIXIT, A. K. Optimization in economic theory.

[12] DREXL, A., AND SALEWSKI, F. Distribution requirements and compactness constraints in school timetabling. *European Journal of Operational Research*.

[13] GOSSELIN, K., AND TRUCHON, M. Allocation of classrooms by linear programming. *Journal of Operational Research Society*.

[14] GROPP, J., AND MORÉ, J. Optimization environments and the neos server. *Approximation Theory and Optimization*.

[15] HULTBERG, T., AND CARDOSO, D. The teacher assignment problem: A special case of the fixed charge transportation problem. *European Journal of Operational Research*.

[16] KANG, L., AND WHITE, G. A logic approach to the resolution of constraints in timetabling. *European Journal of Operational Research*.

[17] LAWRIE, N. An integer linear programming model of a school timetabling problem. *The Computer Journal*.

[18] MCCLURE, R., AND WELLS, C. A mathematical programming model for faculty course assignment. *Decision Science*.

[19] PAECHTER, B. Optimising a presentation timetable using evolutionary algorithms. *AISB Workshop on Evolutionary Computation, Lecture Notes in Computer Science 865*.

[20] PUTERMAN, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

[21] SCHRIJVER, A. *Theory of Linear and Integer Programming*. Wiley, 1998.

[22] TRIPATHY, A. Computerized decision aid for timetabling—a case analysis. *Discrete Applied Mathematics*.

[23] TRIPATHY, A. School timetabling—a case in large binary integer linear programming. *Management Science*.

[24] WILLIAMS, H. P. *Model Building in Mathematical Programming*. Wiley, 2013.

[25] WOLSEY, L. A. *Integer Programming*. Wiley, 1998.

[26] WOLSEY, L. A., AND NEMHAUSER, G. L. *Integer and Combinatorial Optimization*. Wiley, 1999.