ITC 2019: University Course Timetabling with MaxSAT

Alexandre Lemos · Pedro T. Monteiro · Inês Lynce

Abstract This paper describes the *UniCorT* tool designed to solve **uni**versity **co**u**r**se **t**imetabling problems specifically tailored for the 2019 International Timetabling Competition (ITC 2019). The proposed approach comprehends pre-processing, the use of a Maximum Satisfiability (MaxSAT) solver, and a local search procedure.

UniCorT is assessed with the benchmark instances from ITC 2019. The impact of a handful of techniques in the quality of the solution and the execution time is evaluated. We take into account different pre-processing techniques and Conjunctive Normal Form (CNF) encoding, as well as the combination with a local search procedure. The success of our tool is attested by having been ranked among the five finalists of the ITC 2019 competition.

Keywords ITC 2019 · MaxSAT · University Course Timetabling

1 Introduction

The University Course Timetabling Problem (UCTTP) was introduced in the context of the fourth International Timetabling Competition (ITC) 2019 [1]. UCTTP can be informally defined as two complementary problems: (i) course timetabling; and (ii) student sectioning. The goal of *course timetabling* is to find a feasible assignment for all the classes of all courses to a time slot and a room, subject to a set of time constraints. The goal of *student sectioning* is to

The authors would like to thank the reviewers for their helpful comments and suggestions that contributed to an improved manuscript. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference SFRH/BD/143212/2019 (PhD grant), DSAIPA/AI/0033/2019 (project LAIfeBlood), DSAIPA/AI/0044/2018 (project Data2help) and UIDB/50021/2020 (INESC-ID multi-annual funding).

Instituto Superior Técnico, Universidade de Lisboa
INESC-ID, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
E-mail: {alexandre.lemos,pedro.tiago.monteiro,ines.lynce}@tecnico.ulisboa.pt

section students into all the classes required by the courses they are enrolled in, subject to capacity and time constraints.

Timetabling problems have been encoded in the past into propositional logic [2, 3]. This approach has the advantage of making use of propositional satisfiability (SAT) solvers, which are well-known for being quite effective [4].

Example 1 Let us consider four Boolean variables $r_{c_1}^{r_1}$, $r_{c_1}^{r_2}$, $r_{c_2}^{r_1}$ and $r_{c_2}^{r_2}$ representing the possible assignments of the classes c_1 and c_2 to all available rooms (r_1 and r_2). Furthermore, let us consider that the classes are taught at the same time, and thus they cannot be taught in the same room. The same room constraint is encoded as follows: $\neg r_{c_1}^{r_1} \vee \neg r_{c_2}^{r_1}$ and $\neg r_{c_1}^{r_2} \vee \neg r_{c_2}^{r_2}$. A possible solution to this SAT problem is the assignment of class c_1 to room r_1 and class c_2 to room r_2, *i.e.* $r_{c_1}^{r_1} = 1$, $r_{c_1}^{r_2} = 0$, $r_{c_2}^{r_1} = 0$ and $r_{c_2}^{r_2} = 1$. Naturally, for larger domains, one may need to encode cardinally constraints.

UCTTP usually requires to optimize a set of non-mandatory (soft) constraints. Therefore, in this paper we use a maximum satisfiability solver. The maximum satisfiability problem (MaxSAT) [4] is an optimization version of the SAT problem.

This paper provides a detailed description and evaluation of the MaxSAT-based approach that was ranked amongst the five finalists of ITC 2019. The resulting tool *UniCorT* combines pre-processing methods, a MaxSAT solver and a local search procedure to solve UCTTP. We use a MaxSAT solver to find a complete solution to a problem instance, followed by a local search procedure to further optimize the solution. We evaluate two different encodings within *UniCorT* [1]. This tool is evaluated with the large data sets from the ITC 2019 benchmark [1]. Furthermore, we discuss the advantages and disadvantages of the different components of the implementation submitted to ITC 2019.

This paper is organized as follows. Section 2 provides the required background on UCTTP and MaxSAT solving. Section 3 formally describes the ITC 2019 problem. Section 4 describes *UniCorT*. Section 4.1 details the pre-processing techniques. Section 4.2 describes the two different MaxSAT encodings for the course timetabling and student sectioning problems. Section 4.3 describes the local search. Section 5 analyses the evaluation of the proposed approach considering different encodings. The impact of different pre-processing techniques is also taken into account. Finally, Section 6 concludes the paper and discusses possible future directions.

2 Background

This section provides a background on MaxSAT, followed by an overview of existing approaches to solve the UCTTP.

[1] One of these encodings has already been successfully applied to solve the *minimal perturbation problem* in a university course timetabling setting [5]. The paper describing the encoding is available at `http://web.tecnico.ulisboa.pt/~alexandre.lemos/papers/CPAIOR20.pdf`.

2.1 MaxSAT

A propositional formula in conjunctive normal form (CNF) is defined as a conjunction of clauses, where a clause is a disjunction of literals and a literal is either a Boolean variable x or its complement $\neg x$. The propositional satisfiability (SAT) problem consists of deciding whether there is a truth assignment to the variables such that a given CNF formula is satisfied. A formula is satisfied iff there is at least one assignment where all the clauses are satisfied. A clause is satisfied iff there is at least one literal satisfied. Nowadays, most SAT solvers apply conflict-driven clause learning algorithms [6, 7], which are based on the well-known Davis-Putnam algorithm [8] (see [4] for more details).

The MaxSAT problem is an optimization version of SAT, where the *objective* is to find an assignment that maximizes the number of satisfied clauses. A partial MaxSAT formula ($\varphi = \varphi_h \cup \varphi_s$) consists of a set of hard clauses (φ_h) and a set of soft clauses (φ_s). The *objective* in partial MaxSAT is to find an assignment such that all hard clauses in φ_h are satisfied, while maximizing the number of satisfied soft clauses in φ_s.

In this paper, we consider the weighted variant of partial MaxSAT where there is a function $w^\varphi : \varphi_s \rightarrow \mathbb{N}$ associating an integer weight to each soft clause. In this case, the *objective* is to satisfy all the clauses in φ_h and maximize the total weight of the satisfied clauses in φ_s.

Example 2 Recall example 1, where $r_{c_1}^{r_1} = 1$, $r_{c_1}^{r_2} = 0$, $r_{c_2}^{r_1} = 0$ and $r_{c_2}^{r_2} = 1$ was a feasible solution. Now, let us consider that the assignment of class c_1 to room r_1 has a penalty of 1 associated. For this reason, we add $\neg r_{c_1}^{r_1}$ as a soft clause with weight 1. The previously found solution has now a cost 1 and it is not optimal. The optimal solution is $r_{c_1}^{r_1} = 0$, $r_{c_1}^{r_2} = 1$, $r_{c_2}^{r_1} = 1$ and $r_{c_2}^{r_2} = 0$.

Most MaxSAT solvers [9, 10] call a SAT solver iteratively to improve the quality of the solution. There are different algorithms to guide the search. In this work, we use the linear search with clusters algorithm [11]. The basic idea is the following. We start with a formula where all clauses, including the soft clauses, are considered hard. If a solution is found, then the process ends with the optimal solution. Otherwise, the SAT solver is restarted with a relaxed formula. The relaxed formula consists of adding one new variable to each soft clause. Additionally, we add a constraint imposing a limit on the number of relaxed clauses. This limit is incremented each time the formula is not satisfied. This process ends when a solution is found, or when it is impossible to satisfy all the hard clauses.

In general, we assume that all formulas are encoded into CNF. Nevertheless, we will write some constraints in pseudo-Boolean (PB) form for the sake of readability. PB constraints are nothing more than linear constraints over Boolean variables, and can be written as follows: $\sum q_i x_i \text{ OP } K$, where K and all q_i are integer constants, all x_i are Boolean variables, and $\text{OP} \in \{<, \leq, =, \geq, >\}$. PB constraints can be easily translated into CNF [12]. In this work, we tested different encodings for PB constraints.

Example 3 Consider the following PB constraint: $\sum_{i=0}^{2} r_{c_1}^{r_i} \leq 1$. The constraint ensures that the class c_1 can only be assigned to at most one room. One possible CNF encoding is as follows: $(\neg r_{c_1}^{r_0} \vee \neg r_{c_1}^{r_1}) \wedge (\neg r_{c_1}^{r_0} \vee \neg r_{c_1}^{r_2}) \wedge (\neg r_{c_1}^{r_1} \vee \neg r_{c_1}^{r_2})$.

2.2 University Course Timetabling

UCTTP is known to be NP-complete [13, 14]. The organization of timetabling competitions in the past has led to important advances in solving UCTTP [15, 16]. In the literature, there are several different approaches to solve UCTTP, namely: Constraint Programming (CP) [17, 18], Answer Set Programming (ASP) [19], Boolean Satisfiability (SAT) [2], Maximum Satisfiability (MaxSAT) [3, 5], Integer Linear Programming (ILP) [20–22] and local search [20, 23].

Lemos *et al.* [24] proposed two integer programming models to solve university timetabling problems. The *Boolean* model that used two decision variables to describe the assignment of a class to a time slot and the assignment of a class to a room. The authors also proposed a *mixed* model that had an integer variable representing the start time of class and a Boolean variable representing the assignment of a class to a room. The *direct* model presented in this paper can be seen as the extension of the *Boolean* model.

In the context of SAT, Asín Achá *et al.* [3] proposed a CNF encoding with four types of decision variables to solve curriculum-based course timetabling with data from ITC-2007. The authors proposed variables to described: the day of the class; the hour of the class; the room of the class, and finally the different times a curriculum is taught. Obviously, the problem is different from ours. For example, in ITC 2019 the classes can be scheduled in different weeks.

Later, Lemos *et al.* [5] proposed a CNF encoding with four types of decision variables to solve the *minimal perturbation problem* applied to UCTTP with data from ITC 2019. The authors proposed variables to describe: the week of the class; the day of the class; the hour of the class and finally the room of the class. This *linked* encoding can be seen as the extension to the work proposed in [3].

3 Problem Definition

In this section, we formally describe the ITC 2019 problem adapted from [1]. Let us consider a set of courses Co. A course ($co \in Co$) is composed by a set of classes C_{co}. These classes are characterized in configurations ($Config_{co}$) and organized in parts ($Parts_{config}$). A student must attend the classes from a single configuration. A student enrolled in the course co and attending the configuration $config \in Config_{co}$ must attend *exactly one* class from each part $Parts_{config}$. The set of classes belonging to $part \in Parts_{config}$ is represented by C_{part}.

All classes C (from different courses) must have a schedule assigned to them. Each class $c \in C$ has a set of possible periods (P_c) to be scheduled

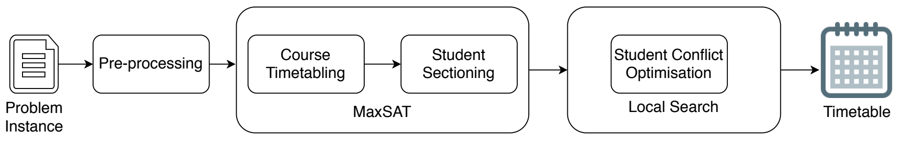

Fig. 1: Overall schema of *UniCorT*.

in. Class c has a hard limit on the number of students that can attend it (lim_c). A class c may have a set of possible rooms (R_c). Each room $r \in R_c$ has $capacity \geq lim_c$. Each class may also have a parent-child relation with another class, *i.e.*, a student enrolled in class c must also be enrolled in the parent $parent_c$.

A time period p corresponds to a 4-tuple (W_p, D_p, h_p, len_p) denoting a set of weeks (W_p), a set of days (D_p), an hour (h_p), and its duration ($len_p > 1$).

Let us consider a set of rooms R where the classes can be scheduled. The travel time, in slots, between two rooms $r_1, r_2 \in R$ is represented by $travel^{r_1}_{r_2}$. Each room $r \in R$ has a set of unavailable periods P_r.

Given a set of students S, each student $s \in S$ is enrolled in a set of courses Co_s. Furthermore, UCTTP is subject to a set of constraints ($constraint_c$ is the set of constraints relating to class c) that can be divided into hard or soft. For brevity, we defined the constraints as needed in the encoding section (for a full description see [1]).

4 Proposed Solution

In this section, we describe *UniCorT*. Figure 1 describes the overall schema of the tool, which has three separate components: pre-processing the UCTTP instance; using a MaxSAT solver to find a solution; and improving the quality of the solution with a local search procedure.

4.1 Pre-processing

The pre-processing component relies on two techniques: (i) identifying of independent sub-instances; and (ii) merging students with exactly the same course enrollment plan.

Technique (i) divides the problem instance into self-contained sub-instances. A set of sub-instances ($Inst$) are self-contained if and only if the following four constraints are upheld:

1. $\forall_{i_1,i_2 \in Inst}\ Co_{i_1} \cap Co_{i_2} = \emptyset$;
2. $\forall_{i_1,i_2 \in Inst}\ R_{i_1} \cap R_{i_2} = \emptyset$;
3. $\forall_{i_1,i_2 \in Inst}\ S_{i_1} \cap S_{i_2} = \emptyset$;
4. $\forall_{i_1,i_2 \in Inst}\ \forall_{c \in C_{i_1}} constraint_c \cap C_{i_2} = \emptyset$.

If these constraints are upheld then we can split the instances without removing any non redundant solution. Note that it is possible to consider a relaxation of this procedure.

Example 4 Let us consider an instance *Inst* with five courses $co_1, \dots, co_5 \in Co_{Inst}$ and five rooms $r_1, \dots, r_5 \in R_{Inst}$. The classes of the courses co_1 and co_2 can only be taught in two rooms r_1 and r_2. The classes of the courses co_3, co_4 and co_5 can only be taught in rooms r_3, r_4 and r_5, respectively. Therefore, we can create four sub-instances $i_1, \dots, i_4 \in Inst$ such that $co_1, co_2 \in Co_{i_1}$, $co_3 \in Co_{i_2}$, $co_4 \in Co_{i_3}$ and $co_5 \in Co_{i_4}$.

Consider three students $s_1, s_2, s_3 \in S_{Inst}$ with the following enrollments: s_1 is enrolled in courses co_1, co_2; s_2 is enrolled in courses co_3, co_4; and finally s_3 is enrolled in co_5. The student enrollments reduces the number of sub-instances since sub-instances i_2 and i_3 violate constraint 3. These two sub-instances must be solved together.

Consider a *no overlap* constraint between the classes of the courses co_4 and co_5. The sub-instances i_3 and i_4 violate constraint 4. For this reason, the instance *Inst* can only be split into two self-contained sub-instances such that $co_1, co_2 \in Co_{i_1}$ and $co_3, co_4, co_5 \in Co_{i_2}$.

Technique (ii) reduces the number of variables and constraints by creating groups of students that share the same curricular plan [25, 26]. The following example illustrates the identification of groups of students with the same curricular plan.

Example 5 Let us consider three courses $co_1, \dots, co_3 \in Co$ and six students $s_1, \dots, s_6 \in S$ that are enrolled in courses as follows: $s_1, \dots, s_4$ are enrolled in the courses co_1 and co_2; and s_5, s_6 are enrolled in the courses co_2 and co_3. In this example, it is possible to generate two *perfect* clusters: clu_1 for all the students enrolled in courses co_1 and co_2; and clu_2 for all the students enrolled in courses co_2 and co_3.

However, this process may remove all the feasible solutions since the classes of each course may have a limitation on the number of students enrolled. Let us denote the greatest common divisor (GCD) between the numbers n_1 and n_2 as $GCD(n_1, n_2)$. Consider now an expansion of the previous example.

Example 6 Let us revisit the Example 5 and consider that each course has two classes, and so $c_1, c_2 \in C_{co_1}, c_3, c_4 \in C_{co_2}$ and $c_5, c_6 \in C_{co_1}$. A student enrolled in the courses co_1, co_2, co_3 must attend exactly one class. The limit on the number of students that can attend is, for each class, as follows: $lim_{c_1} = lim_{c_2} = 4$; $lim_{c_3} = lim_{c_4} = 3$; and $lim_{c_5} = lim_{c_6} = 2$. Figure 2 shows the clusters defined in Example 5 and a possible student sectioning to classes. One can see that the solution is infeasible.

For this reason, we computed GCD between the total number of students enrolled in a course and the smallest capacity of the classes of that course. In this case, we obtain: $GCD(4, 4) = 4$ for course co_1; $GCD(6, 3) = 3$ for

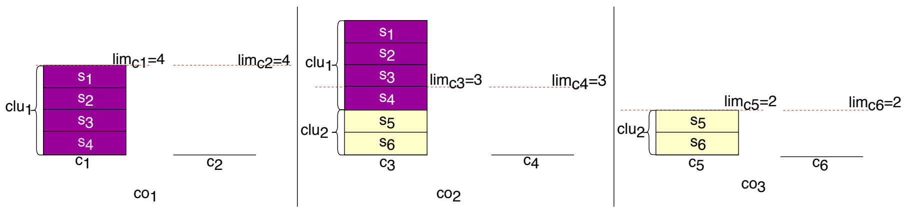

Fig. 2: An infeasible assignment of students to classes based on the clusters defined in Example 5.

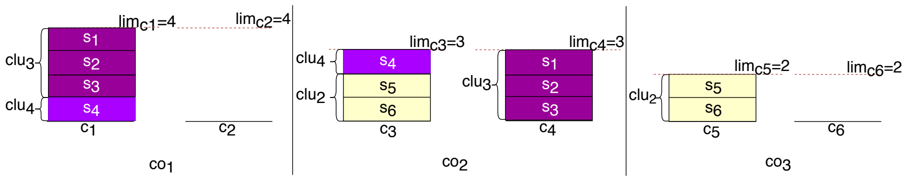

Fig. 3: A feasible assignment of students to classes based on the clusters defined in Example 6.

course co_2; and $GCD(2,2) = 2$ for course co_3. This indicates that the cluster of students enrolled in co_2 needs to be smaller or equal to 3. Therefore, we need to split clu_1. One possibility is to create two clusters. One feasible solution is shown in Figure 3.

This process ensures that it is possible to find a feasible solution to a problem instance, since it is possible to combine all groups of students into classes. However, we may remove the optimal solution by not allowing the assignment of a single student to a given class. The pros and cons of creating clusters are discussed in Section 4.2.3. The GCD can also be used to choose the number of sections of a course in order to reduce the number of conflicts *a priori* [26].

4.2 MaxSAT

In this section, we formally describe two MaxSAT encodings for course timetabling. The two MaxSAT encodings for course timetabling are denoted as: *direct* and *linked* [5]. The ITC 2019 optimization criteria are encoded as soft constraints in both encodings. Furthermore, we also describe a MaxSAT encoding for student sectioning.

4.2.1 Direct Course Timetabling

The most *direct* encoding has only one type of variable to describe the assignment of a class to an allocation slot and a room. This type of encoding is commonly used to describe scheduling problems [20, 21, 24, 27]. For this reason, we decided to start with this type of encoding in CNF. This encoding can be seen as an expansion of the Boolean encoding we proposed in [24]. Our *direct* encoding to solve course timetabling has only two Boolean decision variables:

– t_c^{slot} represents the assignment of class c to the allocation slot $slot$, with $c \in C$ and $slot \in [0, \ldots, |P_c|]$;
– r_c^{room} represents the assignment of class c to the room $room$, with $c \in C$ and $room \in R_c$.

The separation of the decision variables into two variables allows us to reduce the number of unnecessary variables. Using only one decision variable would increase the amount of memory allocation ($|R_c| \times |P_c|$), where most of these variables are always with value 0.

Most UCTTP constraints have a similar encoding when using the *direct* encoding. For this reason, here we only show a few examples.

In the *direct* encoding we need two types of *exactly one* constraints. We need to ensure that each class is assigned *exactly one* slot and in some cases that each class is assigned *exactly one* room.

Example 7 Let us consider two classes c_1 and c_2 with the following characteristics: $D_{c_1} = D_{c_2} = \{0101, 1010\}$; $W_{c_1} = W_{c_2} = \{11110, 01111\}$; $H_{c_1} = \{10, 11\}$; $H_{c_2} = \{11\}$; $P_{c_1} = \{1, \ldots, 12\}$; $P_{c_2} = \{1, \ldots, 8\}$; $R_{c_1} = \{1, 2\}$ and $R_{c_2} = \emptyset$. In this example, we generate the following *exactly one* constraints: $\sum_{i=1}^{12} t_{c_1}^i = 1$,$\sum_{i=1}^{8} t_{c_2}^i = 1$ and $\sum_{i=1}^{2} r_{c_1}^i = 1$.

All constraints related to time allocations, can be encoded into one binary clause for each pair of classes c_i, c_j where $p_i \in P_{c_i}$ and $p_j \in P_{c_j}$:

$$\neg t_{c_i}^{p_i} \vee \neg t_{c_j}^{p_j}. \tag{1}$$

All constraints that involve both time and room allocation can be encoded as follows:

$$\neg t_{c_i}^{p_i} \vee \neg t_{c_j}^{p_j} \vee \neg r_{c_i}^{room_i} \vee \neg r_{c_j}^{room_j}. \tag{2}$$

The following constraints are encoded the same way for both encodings. In order to ensure that a given class cannot be taught in more than V different days (*MaxDays*(V)) we use an auxiliary variable $dayofweek_d^{const}$, where *const* is the identifier of the constraint and $d \in \{1, \ldots, |Days|\}$. This variable corresponds to having at least one class, of this constraint, assigned to weekday d. Now, we only need to ensure that:

$$\sum_{c \in C} \sum_{p \in P_c} \sum_{d \in [1, \ldots, |Day_p|]} dayofweek_d^{const} \leq V. \tag{3}$$

In order to ensure that there are no more than V consecutive slots (breaks) throughout a day between a set of classes (*MaxBlock*/*MaxBreaks*) we need to generate all blocks. After computing all blocks, we add a clause for every class

c_1 to c_n assigned to a period ($p_1 \in P_{c_1}$ to $p_n \in P_{c_n}$) in such a way that it forms a block of classes that breaks one of these constraints:

$$\neg t_{c_1}^{p_1} \vee \ldots \vee \neg t_{c_n}^{p_n}. \tag{4}$$

Example 8 Let us consider two classes c_1 and c_2 that are taught in the same day and cannot overlap in time. Furthermore, all classes are involved in the *MaxBreaks* constraint which ensures that there are 0 breaks (of 1 time slot or more) between them. For simplicity, let us consider that there are only three time slots, t_1, t_2, t_3, per day and all the classes have the same duration of 1 time slot. To ensure that the constraint *MaxBreaks* holds, we add clauses $\neg t_{c_1}^{t_1} \vee \neg t_{c_2}^{t_3}$ and $\neg t_{c_2}^{t_1} \vee \neg t_{c_1}^{t_3}$.

4.2.2 Linked Course Timetabling

It is obvious that we do not need always to take into account the complete schedule information. For some constraints, we only need the information about week, day or hour, and not all the three. For this reason, our *linked* encoding to solve course timetabling has only *four* Boolean decision variables:

- $w_c^{Week_p}$ represents the assignment of class c to the set of weeks $Week_p$, with $c \in C$, and $p \in P_c$;
- $d_c^{Day_p}$ represents the assignment of class c to the set of days Day_p, with $c \in C$, and $p \in P_c$;
- $h_c^{hour_p}$ represents the assignment of class c to the hour $hour_p$, with $c \in C$ and $p \in P_c$;
- r_c^{room} represents the assignment of class c to the room $room$, with $c \in C$ and $room \in R_c$.

The usage of these variables can be seen as an expansion of the encoding proposed in [3]. The scheduling possibilities of a class are usually just a small part of the complete set. For this reason, we only define these variables for acceptable values of the class domain. Furthermore, the *linked* encoding reduces the number of constraints required. For example, *SameStart* constraints (*i.e.* forcing a set of classes to start at the same time) do not require information about the day or week of the class.

In contrast to the *direct* encoding we can reduce the size of each *exactly one* constraint since we have separated the variables for the time allocation. Therefore, we have four *exactly one* constraint for each class (room, hour, day and week). The reduction in the size of the *exactly one* constraints is important since it allows us to avoid a known bottleneck of timetabling encodings using CNF [2].

Example 9 Recall Example 7. The linked encoding for the same instance generates a much smaller number of *exactly one* constraints. In this example, we generate the following *exactly one* constraints: $\sum_{i=1}^{2} w_{c_1}^{i} = 1$, $\sum_{i=1}^{2} w_{c_2}^{i} = 1$, $\sum_{i=1}^{2} d_{c_1}^{i} = 1$, $\sum_{i=1}^{2} d_{c_2}^{i} = 1$, $\sum_{i=1}^{2} h_{c_1}^{i} = 1$, $h_{c_2}^{11} = 1$ and $\sum_{i=1}^{2} r_{c_1}^{i} = 1$.

Once again, most UCTTP constraints have a similar encoding. Therefore, here we only show a few examples. For constraints involving only the variable *hour* (*e.g. SameStart*) we add the clause:

$$\neg h_{c_i}^{hour_{p_i}} \vee \neg h_{c_j}^{hour_{p_j}}. \tag{5}$$

Similarly, the same type of clauses have to be added for constraints involving only the variables *day* (*e.g. SameDay*), *week* (*e.g. SameWeek*) and *room* (*e.g. SameRoom*).

However, not all constraints are so simple to write. For constraints that involve all the time assignments (week, day and hour) we add an auxiliary variable t. This variable has exactly the same meaning that the decision variable of the *direct* encoding. This allows us to generate binary clauses to encode more complex constraints (*e.g.* ensuring that two classes do not *overlap* in time).

To further reduce the size of the clauses, we define the auxiliary variables $sd_{c_j}^{c_i}$ to represent two classes taught in the same day (*i.e.* with at least one day overlap). For each two classes c_i, c_j with $i \neq j$, consider that overlap in days Day_0 to Day_n belong to the domain of class c_i, Day_{n+1} to Day_m belong to the domain of class c_j, with $0 < n < m$. Hence, we add the following equivalence:

$$sd_{c_j}^{c_i} \iff (d_{c_i}^{Day_0} \vee \ldots \vee d_{c_i}^{Day_n}) \wedge (d_{c_j}^{Day_{n+1}} \vee \ldots \vee d_{c_j}^{Day_m}). \tag{6}$$

Similarly, one can define an auxiliary variable $sw_{c_j}^{c_i}$ to represent two classes overlapping in at least one week.

To guarantee that no two classes (c_i, c_j) are taught in the same room (ro) in overlapping times, we add the clause:

$$\neg sd_{c_j}^{c_i} \vee \neg sw_{c_j}^{c_i} \vee \neg h_{c_i}^{hour_{p_i}} \vee \neg h_{c_j}^{hour_{p_j}} \vee \neg r_{c_i}^{ro} \vee \neg r_{c_j}^{ro}. \tag{7}$$

The remaining constraint types are encoded in the same way for both encodings (see previous section).

4.2.3 Student Sectioning

The usage of clusters requires us to define a set $\mathcal{C}luster$ of clusters of students. The number of students merged in the $clu \in \mathcal{C}luster$ is represented by $|clu|$.

In order to solve student sectioning our encoding is extended with *one* decision variable s_{clu}^{c}, where $c \in C$ and $clu \in [1, \ldots, |\mathcal{C}luster|]$. The advantage of the pre-processing step of creating clusters is to reduce the number of variables and constraints required to model students. Note that ITC 2019 instances do not require student sectioning to be *balanced* as in [26].

Example 10 Let us consider again Example 5. Recall that we have three courses $co_1, \ldots, co_3 \in Co$ and six students $s_1, \ldots, s_6 \in S$ that are enrolled in the following courses: students $s_1, \ldots, s_4$ are enrolled in the courses co_1 and co_2; and students s_5, s_6 are enrolled in the courses co_2 and co_3. Therefore, it is possible to generate two *perfect* clusters: clu_1 for all the students enrolled in courses co_1 and co_2; and clu_2 for all the students enrolled in courses co_2 and co_3. Additionally, consider that $|C_{co_1}| = |C_{co_2}| = 1$ and $|C_{co_3}| = 6$.

The two clusters allow to reduce the number of variables from *22* (the number of students times the number of classes available for each student) to *9* (the number of clusters times the number of classes). Note that the impact of the clusters depends not only on the number of students merged but also on the course composition. In this case, the smaller cluster (clu_2) has a greater impact since the courses have a larger number of classes, more precisely 7 variables.

In order to ensure that a student can only be assigned to a single course configuration, we define an auxiliary variable for each pair configuration-cluster of students. The variable is denoted as $conf_{clu}^{config}$, where $clu \in [1, \ldots, |Cluster|]$, $config \in Config_{co}$ and $co \in Co$.

We need to add an *exactly one* constraint to ensure that each cluster of students *id* is enrolled in *exactly one* configuration of each course. To ensure that the class capacity is not exceeded, we add the following constraint for each class c:

$$\sum_{clu \in [1,\ldots,|Cluster|]} |clu| \times s_{clu}^{c} \leq lim_c. \tag{8}$$

In addition, to ensure that a cluster of students clu enrolled in a class c is also enrolled in this parent class $parent_c$, we add the following clause:

$$\neg s_{clu}^{c} \vee s_{clu}^{parent_c}. \tag{9}$$

Finally, we need to ensure that a cluster of students clu is enrolled in *exactly one* class of each part of a single configuration of the course *co*. The conflicting schedule of classes attended by the same cluster of students is represented by a set of weighted soft clauses. For each cluster of students *id* enrolled in two classes c_i, c_j overlapping in time, we add:

$$\neg s_{clu}^{c_i} \vee \neg s_{clu}^{c_j} \vee \neg sw_{c_j}^{c_i} \vee \neg sd_{c_j}^{c_i} \vee \neg h_{c_i}^{hour_{c_i}} \vee \neg h_{c_j}^{hour_{c_j}}. \tag{10}$$

4.3 Local Search: Student Conflict Optimisation

The *goal* of this procedure is to improve the quality of the solution found without changing the schedule and room assignments of the courses. Neighborhood structures are the basis of this local search (LS) procedure. In this work, the neighborhood consists of small changes in the student sectioning. To create a new neighborhood two operations can be performed: (i) allocating a cluster of students to a different class with empty seats and (ii) *swaping* two clusters of students between classes. Considering these moves, the procedure does not require the knowledge of course timetabling constraints. The LS procedure stops when the neighbors of the best solution cannot reduce the number of conflicts (*i.e.* the solution found has the best cost of its neighborhood).

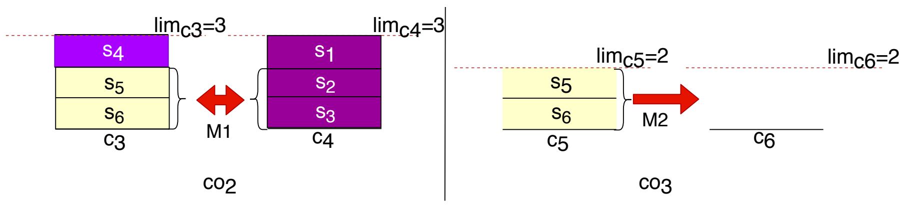

Fig. 4: The two neighbors of the solution in Example 6.

Example 11 Let us consider again Example 6. Additionally, consider that the classes c_3 and c_5 are taught at the same time, on the same day and in the same week. For this reason, the solution shown in Figure 3 has two conflicts (students s_5 and s_6 are sectioned into two classes that overlap in time). This solution would have two neighbors for which the solution would improve. These two neighbors are shown in Figure 4. The neighbor **M1** *swaps* students s_5 and s_6 with students s_2 and s_3 (from class c_3 to c_4). However, this move is not possible since the clusters do not allow to separate the students s_2 and s_3 from s_1. The neighbor **M2** results from is just sectioning students s_5 and s_6 to the class c_6 instead of c_5. This change does not require to break any clusters and reduces the number of conflicts to zero.

5 Experimental Evaluation

In this section, we discuss the main computational results obtained. First, we describe the experimental setup used to validate *UniCorT* (Section 5.1). Next, we discuss our results for UCTTP (Section 5.2). Finally, we present a summary of the results (Section 5.3).

5.1 Experimental Setup

The experimental evaluation was performed on a computer with Fedora 14, with 32 CPUs at 2.6 GHz and 126 Gb of RAM. All results were obtained when running the solver with a time out of 6,000 seconds.

We used the benchmark obtained from ITC 2019 [1] to validate our tool. The benchmark is divided into three groups of instances (early, middle, late). All results were verified by an online validation tool provided by the organizers[2].

UniCorT was implemented in C++, using the MaxSAT solver *TT-Open-WBO-Inc* [10, 28][3] as a black box. *TT-Open-WBO-Inc* is a linked MaxSAT

[2] `https://www.itc2019.org/validator`, accessed in January of 2020

[3] *TT-Open-WBO-Inc* won both the Weighted Incomplete tracks at *MaxSAT Evaluation 2019*. The results are available at `https://maxsat-evaluations.github.io/2019/`.

solver [9] that has different algorithms and encodings to solve a given problem. The results shown in this paper correspond to the best configuration of the parameters of the solver. The solver was executed with the following parameters:

⋆ *-algorithm=6* corresponding to the use of linear search with the clusters algorithm [11];
⋆ *-pb=2* corresponding to the use of the adder encoding [29] to convert the PB constraints to CNF;
⋆ *-amo=0* corresponding to the use of the ladder encoding [30] to convert *exactly one* constraints to CNF.

The linear search with the clusters algorithm uses a lexicographic optimization criterion [31]. Recall that ITC 2019 considers four optimization criteria: the cost of assigning a class to a room; the cost of assigning a class to a time slot; the number of students conflicts; and the weighted sum of violated soft constraints. Each instance has its own weight for each criterion. We have computed the worst possible penalization of each criterion and used the value to order the lexicographic optimization.

The XML parser used to parse the ITC 2019 input file was *RAPIDXML*[4]. Also, we make our implementation available on github (`https://github.com/ADDALemos/MPPTimetables/tree/ITC-2019`).

5.2 Computational Evaluation

In this section, we discuss the results of *UniCorT* and all possible configurations tested.

5.2.1 Pre-processing Techniques

Recall that we discussed two pre-processing techniques: (i) identification of independent sub-instances and (ii) merging students into clusters.

Identification of independent sub-instances The identification of independent sub-instances allows us to split the problem without loosing solutions. In the end, it is just a question of combining all the solutions. On average, we can split an instance into 3 sub-instances. In most cases, the instances have one large instance and two smaller instances. A detailed description of the sub-instances is shown in Table 1.

[4] *RAPIDXML* is available at `http://rapidxml.sourceforge.net/manual.html`, accessed in February 2019.

Table 1: Number of sub-instances found and the respective average size.

	Instance	**# Inst**	**# Classes**	
			Avg.	**Median**
Early	agh-fis-spr17	2	599.5	599.6
	agh-ggis-spr17	3	617	12
	bet-fal17	4	292.25	127
	iku-fal17	4	649.5	140
	mary-spr17	3	281.3	10
	muni-fi-spr16	1	575	575
	muni-fsps-spr17	3	543.3	10
	muni-pdf-spr16c	4	635.25	9.5
	pu-llr-spr17	3	388.6	165
	tg-fal17	1	711	711
Middle	agh-ggos-spr17	2	620.5	620.5
	agh-h-spr17	1	460	460
	lums-spr18	1	487	487
	muni-fi-spr17	2	515	515
	muni-fsps-spr17c	6	130.1	6.5
	muni-pdf-spr16	5	909	2
	nbi-spr18	3	260	34
	pu-d5-spr17	6	389.7	12
	pu-proj-fal19	4	2207	51
	yach-fal17	3	156.3	165
Late	agh-fal17	4	1876.66	10
	bet-spr18	4	340.75	140
	iku-spr18	5	556.4	56
	lums-fal17	1	502	502
	mary-fal18	3	319.6	5
	muni-fi-fal17	1	535	535
	muni-fspsx-fal17	4	326.4	32
	muni-pdfx-fal17	6	1854.83	2.5
	pu-d9-fal19	7	816.42	8
	tg-spr18	1	676	676

Merging students Merging students with the same curricular plans allows to reduce the number of variables and constraints on the student sectioning part of the problem. Figure 5 shows the percentage of the total number of variables required to model students using different clusters. The *clusters* represent the percentage of the total number of variables required to model students with different curricular plans per instance. However, this type of clusters cannot be applied in practice since they would remove feasible solutions (see Example 3). Alternately, the *GCD clusters* represent the cluster divided using the GCD method discussed above. Recall that the number of variables needed to model students is influenced by the number of classes per enrolled course (see Example 10).

Most instances have a significant bottleneck in the creation of clusters caused by the hard limit on the number of students enrolled in a class. On average the GCD clusters are 40 points worse than a normal cluster. On average, one can reduce the number of variables relating to students up to 23%. Instances *nbi-spr18* and *yach-fal17* have a larger reduction on the number of

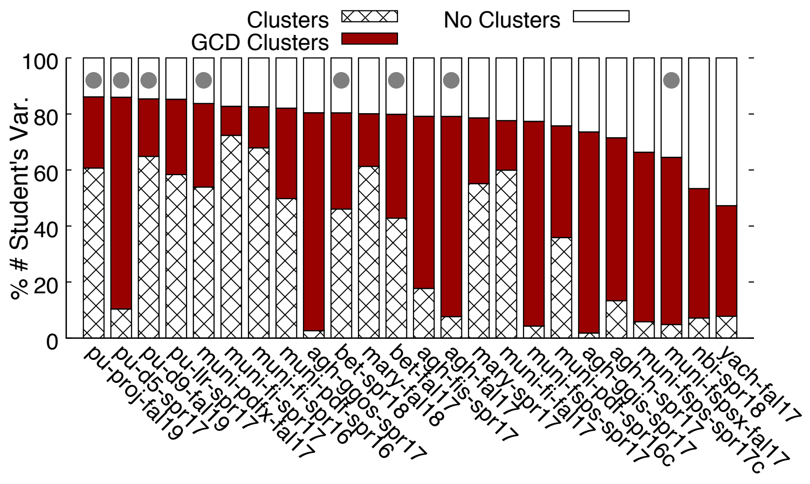

Fig. 5: Percent decrease in the number of variables required to model students increasing clustering strategy. The grey circles represent timed out instances when using GCD clusters.

variables (around 50%). On the other hand, instances *pu** have the smallest reduction (14%).

5.2.2 MaxSAT Solving

In this section, we compare the different CNF encodings and the advantages of solving the course timetabling problem separated from the student sectioning problem.

Figure 6 compares the total number of hard clauses with the number of soft clauses generated by our encodings. It is clear that the number of soft clauses is considerably smaller for all instances. On average, the number of soft clauses is 2% of the global number of clauses. Most instances that timed out have a higher percentage of soft clauses but these instances also have a larger overall number of soft clauses. With this difference in mind, we focused more on the hard constraints as they are dominant.

We can find a solution within the time limit for 20 out of 30 instances using our best approach (see Table 2). However, the solver was not able to prove optimality within the time limit on any of the instances.

Figure 7 compares the number of hard clauses generated by the CNF encoding and the CPU time needed to find the best solution for each instance considering two approaches to encode the problem (*direct* and *linked*). In general, one can see that the instances with a larger number of hard constraints take a larger amount of time. Using the *linked* encoding reduces the number of constraints needed per instance. Therefore, one can solve 9 more instances within the time limit. Most of the unsolved instances actually have two orders

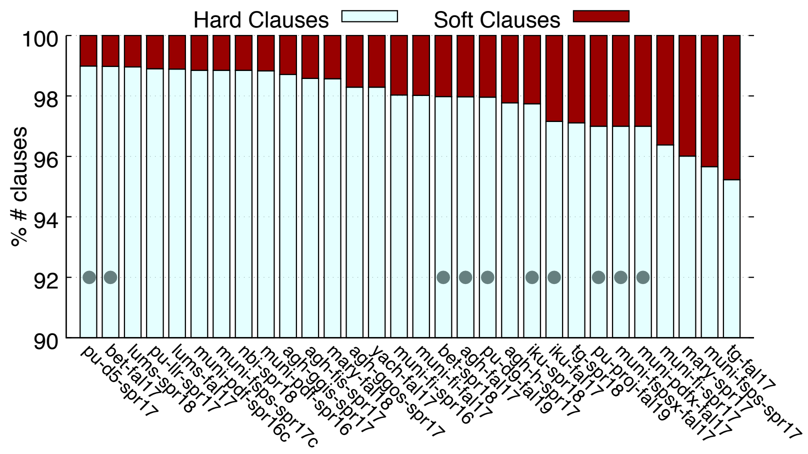

Fig. 6: Percentage of soft clauses for each instance. The grey circles represent timed out instances.

of magnitude more constraints than the other instances (top right corner of Figure 7). Most of these constraints result from the *MaxBlock* and *MaxBreak* constraints (to be discussed further on).

Figure 8 compares the number of hard clauses generated by the CNF encoding for each approach tested. One can see that the *direct* encoding requires much more constraints to encode the same instance. The *direct* encoding requires, on average, 7×10^{10} more constraints than the *linked* encoding. This can be explained by the fact that most constraints are only related either to an hour, day or week. There are few constraints that involve all weeks, days and hours simultaneously. Furthermore, one can reduce the need to combine all these with the usage of auxiliary variables (*e.g. sd*). For this reason, the usage of only one variable for the time allocation problems creates unnecessary constraints.

Figure 9 compares the time spent to find the best solution for each approach tested. One can see that the *direct* encoding requires only a few more seconds to find a solution of the same cost for each instance that both approaches solve within the time limit. The *direct* encoding requires, on average, 200 seconds more than the *linked* encoding. Furthermore, we can clearly see that most instances that timed out with the *direct* encoding are solved in only a few seconds by the *linked* encoding (bottom right of Figure 9). On average, the *linked* encoding requires only 2,000 seconds to solve the timed out instances.

In case of the *linked* encoding, for most instances, the solver requires only a short amount of time to produce the best solution. Figure 10 shows a comparison between the normalized cost of the best found solution and the CPU time in seconds. The figure shows the normalized cost since each instance has its own weights on the optimization criterion and therefore would be impossible

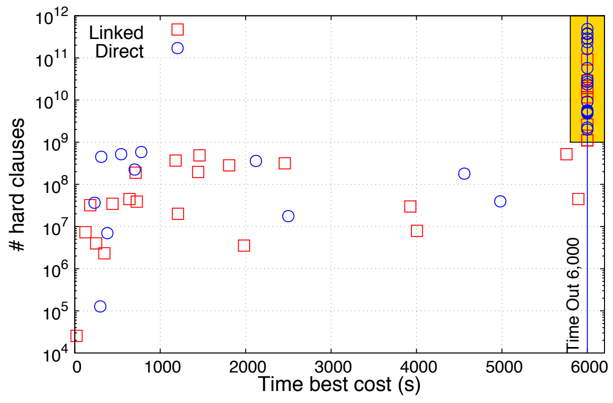

Fig. 7: A comparison between the *linked* and the *direct* encodings in terms of the number of hard constraints (log scale) versus the time spend to find the best solution. The orange square contains the instances that timed out. 33.3% and 63.3% of the instances are in the square for *linked* and the *direct* encodings respectively.

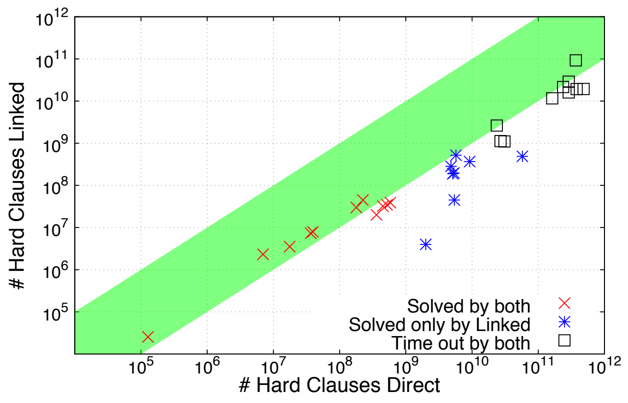

Fig. 8: A comparison between the *linked* and the *direct* encodings in terms of the number of hard constraints (log scale). The blue line represents the time limit.

to compare them in the same graph. One can see that the best solution, for most instances, is found early on (within 2,000 seconds). In fact, only 5 out of 20 instances improve their quality after 2,000 seconds. The quality of the solution does not improve until the time out is reached.

MaxBlocks and MaxBreaks. Figure 11 shows the percentage of clauses generated from *MaxBlocks* and *MaxBreaks* constraints for each instance. One can see that these constraints generate a significant number of additional clauses.

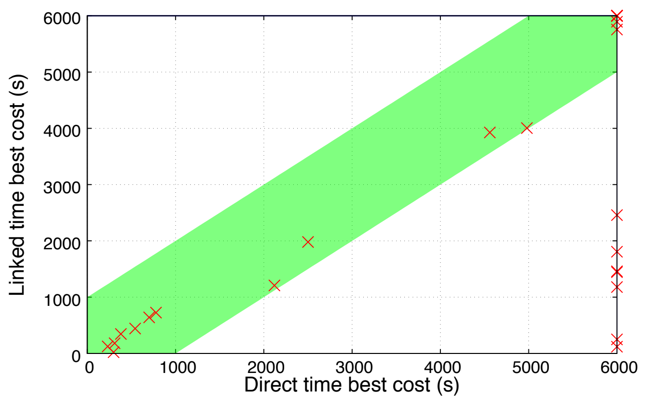

Fig. 9: A comparison between the *linked* and the *direct* encodings in terms of CPU time for each instance.

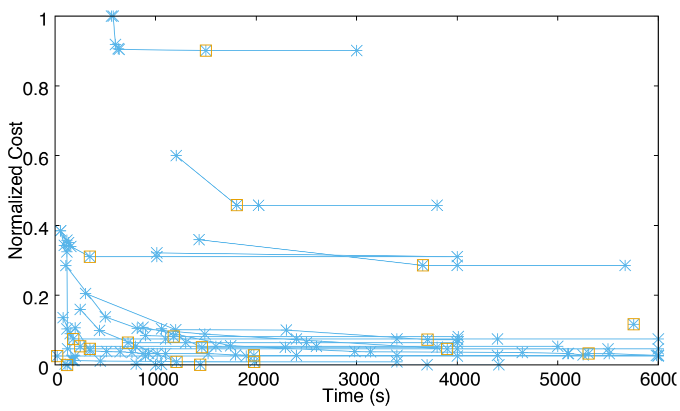

Fig. 10: Normalized cost versus CPU time for each instance with *linked* encoding. The grey square represents the best cost found.

In the worst case, we need to generate over 35% more clauses to deal with these constraints. Note that we cannot solve the 10 instances with a higher number of hard clauses (see Figure 8). In most cases, the high number of constraints is caused by *MaxBlocks* and *MaxBreaks* constraints. The exceptions are the instances from *iku**, which have the largest number of classes. In fact, the size of our *exactly one* constraints is much larger than 26 which is the limit found by Bittner *et al.* [2] for solvable instances.

Decomposing UCTTP. Our best approach decomposes the UCTTP into two sub-problems: (i) course timetabling and (ii) student sectioning. This decomposition may remove the optimal solution. However, it does not remove any feasible solution. The goal of decomposition is to reduce the size of the prob-

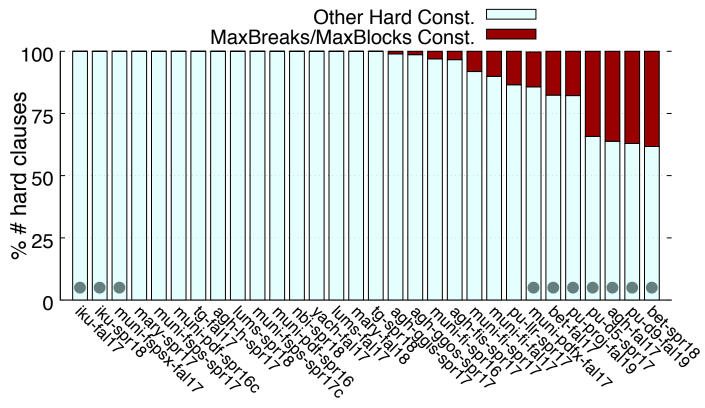

Fig. 11: Percentage of clauses generated by *MaxBlocks* and *MaxBreaks* constraints. The grey circles represent timed out instances.

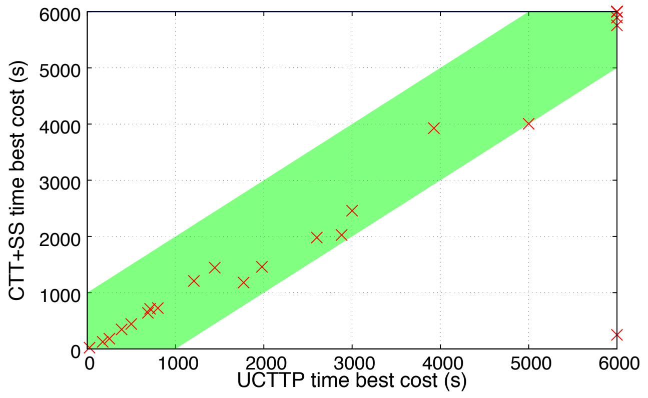

Fig. 12: A comparison of the CPU time, in seconds, when solving the *CTT+SS* problems separated or the *UCTTP* as a whole.

lem, especially for instances with a large number of clusters of students. The decomposition allows us to solve 3 more instances. Figure 12 compares the performance of the solver before and after decomposing the problem, in terms of CPU time.

5.2.3 Local Search

Our straightforward implementation of this method allows to improve the quality of the solution without adding significant overhead. On average, the method requires only 6% of the overall execution time of the approach. Fig-

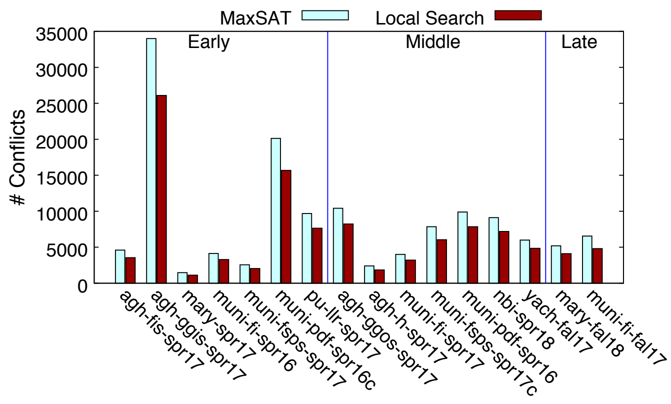

Fig. 13: A comparison of the cost, in terms of students conflicts, before and after applying the LS procedure.

ure 13 compares the number of conflicts, before and after this procedure. On average the procedure reduces the number of conflicts by 22%.

5.3 Final Results

Table 2 shows the best cost found by our best approach per optimization criteria and instance. Note that the penalties associated with the three optimization criteria (student conflicts, allocation penalty, and additional soft constraints) vary from instance to instance. Therefore, it is difficult to compare them. Nevertheless, one can see that the student conflict criteria, overall, is the most costly even with the LS method. The *muni** instances are on average the worst in terms of room allocation penalty. This can be explained by the normal structure of these instances since they have few room options (R_c) and a large penalty associated.

6 Conclusion and Future Work

This paper discusses the results obtained by our approach in ITC 2019. The resulting tool *UniCorT* is able to solve two thirds of the benchmark instances from ITC 2019 within the time limit of 6,000 seconds. This tool placed among the five finalists. *UniCorT* takes advantage of two pre-processing techniques that search for: (i) self-contained sub-instances and (ii) clusters of students. The first method is able to divide, on average, an instance into 3 sub-instances. The clustering of students is able to reduce the number of variables used, on

Table 2: The cost per optimization criteria and instance.

Instance	**Cost**	**Students**	**Time**	**Room**	**Distribution**
agh-fis-spr17	35139	3555	2248	2312	404
agh-ggis-spr17	194138	26097	2737	22270	2029
bet-fal17	UKN				
iku-fal17	UKN				
mary-spr17	51147	1114	1376	805	7290
muni-fi-spr16	19314	3286	352	628	120
muni-fsps-spr17	211142	2040	58	292	360
muni-pdf-spr16c	567900	15678	58316	27600	4361
pu-llr-spr17	68003	7642	1169		30
tg-fal17	6774	0	1792	30	158
agh-ggos-spr17	79745	8230	6045	9045	358
agh-h-spr17	55887	1848	1442	1039	2656
lums-spr18	594	0	0	509	17
muni-fi-spr17	18080	3212	284	958	21
muni-fsps-spr17c	618217	6048	411	1027	141
muni-pdf-spr16	310994	7853	38680	27094	900
nbi-spr18	49924	7196	5946	9208	5
pu-d5-spr17	UKN				
pu-proj-fal19	UKN				
yach-fal17	32198	4856	8	1008	687
agh-fal17	UKN				
bet-spr18	UKN				
iku-fal18	UKN				
lums-fal17	1151	0	105	626	63
mary-fal18	44097	4107	596	665	234
muni-fi-fal17	19683	3810	86	289	9
muni-fspsx-fal17	UKN				
muni-pdfx-fal17	UKN				
pu-d9-fal19	UKN				
tg-spr18	31900	0	1942	3996	1201

average, by 23%. The LS method, in the end, is able to reduce the number of conflicts by 22% without adding a significant overhead.

UniCorT solves the course timetabling and student sectioning problems separately in order to reduce the size of the problem and thus the execution time. This decomposition does not remove any feasible solutions. However, it may remove the optimal solution but allows us to solve more instances within the time limit.

The MaxSAT encodings applied in *UniCorT* encode *MaxBlock* and *MaxBreaks* constraints by blocking all invalid assignments. In order to block the invalid assignments, we generate all block combinations possible. However, this method proves inefficient for large instances. For this reason, we plan to work on new ways of encoding these constraints in such a way we avoid enumerating all possible blocks. More precisely, we can take advantage of symmetries in the blocks structure to reduce the clauses generated.

References

1. Müller, T., Rudová, H., Müllerová, Z.: University course timetabling and International Timetabling Competition 2019. In: Burke, E.K., Di Gaspero, L., McCollum, B., Musliu, N., Özcan, E. (eds.) Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT-2018). pp. 5–31 (2018)
2. Bittner, P.M., Thum, T., Schaefer, I.: SAT encodings of the at-most-k constraint - A case study on configuring university courses. In: Proceedings of the Software Engineering and Formal Methods (SEFM). pp. 127–144 (2019)
3. Asín Achá, R.J., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. Annals of Operations Research **218**(1), 71–91 (2014)
4. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
5. Lemos, A., Monteiro, P.T., Lynce, I.: Minimal perturbation in university timetabling with maximum satisfiability. In: Proceedings of 17th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR) (2020), preprint at `http://web.tecnico.ulisboa.pt/alexandre.lemos/papers/CPAIOR20.pdf`
6. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Proceedings of the International Conference on Computer-Aided Design (ICCAD). pp. 220–227. IEEE Computer Society / ACM (1996)
7. Jr., R.J.B., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Kuipers, B., Webber, B.L. (eds.) Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI) and Ninth Innovative Applications of Artificial Intelligence Conference (IAAI). pp. 203–208. AAAI Press / The MIT Press (1997)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
9. Martins, R., Manquinho, V.M., Lynce, I.: Open-WBO: A modular MaxSAT solver,. In: Theory and Applications of Satisfiability Testing (SAT) - 17th. pp. 438–445 (2014)
10. Nadel, A.: TT-Open-WBO-Inc: Tuning polarity and variable selection for anytime SAT-based optimization. In: Proceedings of the MaxSAT Evaluations (2019)
11. Joshi, S., Kumar, P., Martins, R., Rao, S.: Approximation strategies for incomplete MaxSAT. In: Principles and Practice of Constraint Programming (CP). pp. 219–228 (2018)
12. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4), 1–26 (2006)
13. Lovelace, A.L.: On the complexity of scheduling university courses. Master's thesis, California Polytechnic State University, San Luis Obispo

(2010)
14. Herres, B., Schmitz, H.: Decomposition of university course timetabling. Annals of Operations Research (2019)
15. McCollum, B.: University timetabling: Bridging the gap between research and practice. In: 5th International Conference on the Practice and Theory of Automated Timetabling (PATAT). pp. 15–35. Springer (2006)
16. Vrielink, R.A.O., Jansen, E.A., Hans, E.W., van Hillegersberg, J.: Practices in timetabling in higher education institutions: a systematic review. Annals of Operations Research **275**(1), 145–160 (2019)
17. Müller, T.: ITC-2007 solver description: a hybrid approach. Annals of Operations Research **172**(1), 429 (2009)
18. Atsuta, M., Nonobe, K., Ibaraki, T.: ITC-2007 track 2: an approach using a general CSP solver. In: 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT). pp. 19–22 (2008)
19. Banbara, M., Inoue, K., Kaufmann, B., Okimoto, T., Schaub, T., Soh, T., Tamura, N., Wanko, P.: *teaspoon* : Solving the curriculum-based course timetabling problems with Answer Set Programming. Annals of Operations Research **275**(1), 3–37 (2019)
20. Lemos, A., Melo, F.S., Monteiro, P.T., Lynce, I.: Room usage optimization in timetabling: A case study at Universidade de Lisboa. Operations Research Perspectives **6**, 100092 (2019)
21. Lindahl, M., Stidsen, T., Sørensen, M.: Quality recovering of university timetables. European Journal of Operational Research **276**(2), 422 – 435 (2019)
22. Phillips, A.E., Walker, C.G., Ehrgott, M., Ryan, D.M.: Integer programming for minimal perturbation problems in university course timetabling. Annals of Operations Research **252**(2), 283–304 (2017)
23. Gülcü, A., Akkan, C.: Robust university course timetabling problem subject to single and multiple disruptions. European Journal of Operational Research **283**(2), 630 – 646 (2020)
24. Lemos, A., Melo, F.S., Monteiro, P.T., Lynce, I.: Disruptions in Timetables: A Case Study at Universidade de Lisboa. Journal of Scheduling (2020)
25. Carter, M.W.: A comprehensive course timetabling and student scheduling system at the University of Waterloo. In: 3rd International Conference on the Practice and Theory of Automated Timetabling (PATAT). pp. 64–84 (2000)
26. Schindl, D.: Optimal student sectioning on mandatory courses with various sections numbers. Annals of Operations Research **275**(1), 209–221 (2019)
27. Burke, E.K., Mareček, J., Parkes, A.J., Rudová, H.: Penalising patterns in timetables: Novel integer programming formulations. In: Operations Research Proceedings, pp. 409–414. Springer (2008)
28. Nadel, A.: Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In: Proceedings of the 19th Conference on Formal Methods in Computer Aided Design (FMCAD) (2019)

29. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Information Processing Letters **68**(2), 63–69 (1998)
30. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with boolean variables. In: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT). vol. 3542, p. 1–15 (2004)
31. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Annals of Mathematics and Artificial Intelligence **62**(3-4), 317–343 (2011)