Modeling High School Timetabling as Partial Weighted maxSAT

Emir Demirović, Nysret Musliu

Vienna University of Technology
Database and Artificial Intelliegence Group
{demirovic,musliu}@dbai.tuwien.ac.at

Abstract. High School Timetabling (HSTT) is a well known and wide spread problem. The problem consists of coordinating resources (e.g. teachers, rooms), times, and events (e.g. lectures) with respect to various constraints. Unfortunately, HSTT is hard to solve and even simple variants of HSTT are NP-complete problems. We propose a new detail modeling of the general HSTT as SAT, in which all constraints are treated as hard constraints. In order to take into account soft constraints, we extend the SAT model to Partial Weighted maxSAT. In addition, we have developed a SMT approach, in which constraints are incrementally added rather then processed all at once. We perform detailed experiments on instances taken from the Third International Timetabling Competition 2011 (ITC 2011) benchmark repository in order to determine the most appropriate maxSAT solvers and cardinality constraint encodings, evaluate the developed SMT approach, and compare our maxSAT approach with the state-of-the-art exact Integer Programming (IP) approach and the ITC 2011 results. The results show that maxSAT is competitive for HSTT and outperforms the state-of-the-art exact IP approach in the used instances. Moreover, we show that combining several maxSAT solvers completely outperforms IP.

Keywords: High school timetabling, maxSAT, SAT encodings, cardinality constraints, XHSTT, SMT, ITC2011

1 Introduction

The problem of high school timetabling (HSTT) is to coordinate resources (e.g. rooms, teachers, students) with times in order to fulfill certain goals (e.g. scheduling lectures). Every high school requires some form of timetabling which is a well known and wide spread problem. The difference between a good and a bad timetable can be significant, as timetables directly contribute to the quality of the educational system, satisfaction of students and staff, etc. Every timetable affects hundreds of students and teachers for prolonged amounts of time, since each timetable is typically used for at least a semester, making HSTT an extremely important and responsible task. However, constructing timetables by hand can be time consuming, very difficult, and error prone. Thus, developing high quality algorithms which would generate automatically timetables is of great importance.

Unfortunately, High School Timetabling is hard to solve and just finding a feasible solution of simple variants of High School Timetabling has been proven to be NP-complete [?]. Apart from the fact that practical problems can be very large and have many different constraints, high school timetabling requirements vary from country to country. Due to this, many variations of the timetabling problem exist. A lot of research has been done and HSTT is still an active field of research, even having its own specific HSTT competition ITC 2011.

In order to standardize the formulation for HSTT, researches have recently proposed a general high school timetabling problem formulation [?] called XHSTT. This formulation has been endorsed by the Third International Timetabling Competition 2011 (ITC 2011) [?,?] which attracted 17 competitors from across the globe. In this work, we consider the general HSTT problem formulation (XHSTT).

Algorithm for XHSTT are mostly based on heuristic (incomplete) algorithms, as discussion in Section 3. These algorithms aim to provide upper bounds to the problem by searching through only a limited part of the search space. In this work, we consider an exact approach, which contrary to the heuristic algorithms, exhaustively explores the complete search space. In general, the first step in devising a complete algorithm is to precisely capture the problem definition using some mathematical formalism, which is difficult to do when there are many constraints and complex interaction between entities and constraints, as in XHSTT.

We model the complex formalism of XHSTT using only Boolean variables and basic logical connectives. Hard constraints are translated into propositional Boolean formulas (SAT). To account for the soft constraints, the model is extended with the use of Partial Weighted maxSAT. By using our modeling any solution of cost c for the Partial Weighted maxSAT formula can be directly translated into a XHSTT solution with cost c.

However, there are several additional difficulties. Apart from precisely modeling the problem using the restrictive language of propositional logic, one needs to take care of important special cases in order to significantly simplify the encodings in practice, as well as consider different modeling options in the form of cardinality constraints. In addition, there are many different maxSAT solvers, each with their own solving techniques.

To experimentally evaluate our approach, we took all relevant XHSTT instances (see Section 7.1) and out of the pool of 39 instances we were able to model 27 of them. The remaining 12 instances were not modeled because the currently used maxSAT formulation for XHSTT does not support resource assignments in general. We have a specific modeling for resource assignments (Assign Resource Constraints and related constraints) for two instances, but our current model is not practical for other instances with resource assignments.

We discuss these special cases, and empirically evaluate different cardinality constraint modelings and solvers, in order to determine the best modeling for XHSTT. We show that competitive results can be obtained by modeling XHSTT as Partial Weighted maxSAT. Our method is compared to the state-of-the-art Integer Programming approach, and computational results demonstrate that we outperform IP on the used instances. If we allowed to combine several maxSAT solvers, we are able to completely outperform IP. Furthermore, we have experimented with a SMT approach, in which soft constraints are gradually added throughout the search.

It is important to note that the competing IP approach relies on using Gurobi, a highly engineered piece of software, while we use publicly available (and in some cases open source) maxSAT solvers that are not so heavily engineered and still provide good results. In addition, the maxSAT model presented here plays a crucial role in the recently developed heuristic algorithm, which uses maxSAT in a large neighborhood search algorithm [**?**].

It is worth mentioning that in [**?**] a SAT encoding is studied for a related, albeit different problem, namely the Curriculum-based course timetabling (CTT) problem. Unfortunately, many important constraints of the general HSTT problem cannot be formulated as CTT. For example, Limit Idle Times constraint, which typically limits the number of idle times between lessons a teacher has, is an extremely important constraint in XHSTT and cannot be modeled in CTT. Many of CTT constraints are special cases of XHSTT ones or can be adapted for XHSTT. All of the constraints in CTT but one can be modeled as XHSTT. The translation of CTT into XHSTT has been studied in [**?**]. Because of this, new and more generalized encodings must be explored in order to model XHSTT.

To summarize, we state our main contributions explicitly:

– We show that XHSTT can be modeled as Weighted Partial maxSAT despite the fact that XHSTT is very general and has many different constraints, including both hard and soft constraints.

All constraints are included in their general formulations, with the exception of resource assignment constraint. Important alternative encodings for special cases are discussed, which includes a special case of resource assignments as well.
- We investigate empirically the performance of our model using 27 out of 39 instances from the Third International Timetabling Competition 2011 benchmark repository. We compare different maxSAT solvers and cardinality constraint encodings to determine the most appropriate combinations, evaluate a SMT approach, compare with the state-of-the-art Integer Programming approach, and show how well our maxSAT approach would perform if it was submitted to the International Timetabling Competition 2011. Based on the results we conclude that our developed exact approach outperforms the state-of-the-art IP approach. Moreover, we show that combining several maxSAT solvers completely outperforms IP.
- We created maxSAT instances based on XHSTT. These instances were submitted to the maxSAT Competition 2014 and have been used since. They have proven to be challenging benchmarks for maxSAT solvers.

Some of the results of this paper [?] have been presented in the LaSh 2014 workshop with no formal proceedings.

The rest of the paper is organized as follows: in Section 2, we give an informal overview of XHSTT, followed by Section 3 where we discuss related work. In Section 4, we give a detailed mathematical formulation of the XHSTT problem which we previously informally described. Our maxSAT modeling of XHSTT is given in Section 5, after which we describe our SMT approach in Section 6. Detailed experimental results are given in Section 7, where several important questions are answered. Finally, in Section 8 we give concluding remarks and ideas for future work.

2 Problem Description

High School Timetabling has been extensively studied in the past. However, a lot of work has been done in isolation, because each country has its own educational system which resulted in many different timetabling formulations. Thus, it was difficult to compare algorithms and the state-of-the-art was unclear. To solve this issue and encourage timetabling research, researchers have agreed on a standardized general timetabling formulation called XHSTT [?]. This formulation was general enough to be able to model education systems from different countries and was endorsed by the International Timetabling Competition 2011. This is the formulation used in this work.

The general High School Timetabling formulation specifies three main entities: times, resources, and events. Times refer to the available discrete time units, such as Monday 9:00-10:00 and Monday 10:00-11:00. Resources correspond to available rooms, teachers, students, etc. The main entities are the events, which in order to take place require certain times and resources. An event could be a mathematics lecture, which requires a math teacher (who needs to be determined) and a specific student group (both the teacher and the student group are considered resources) and two units of time (two *times*). Events are to be scheduled into one or more *solution events* or *subevents*. For example, a mathematics lecture with a total duration of four hours can be split into two subevents with a duration of two hours each, but can also be scheduled as a single subevent with a duration of four hours (constraints may restrict the durations of subevents).

Constraints impose limits on what kind of assignments are desirable. They may state that a teacher can teach no more than five lessons per day, that younger students should attend more demanding subjects (e.g. mathematics) in the morning, etc. It is important to differentiate between hard and soft constraints. The former are very important and are given precedence over the latter, in the sense that any single violation of a hard constraint is more important than all soft constraints violations combined. Thus, one aims to satisfy as many hard constraint violations as possible, and then optimize for the soft constraints. In the general formulation, any constraint may be declared hard or soft and no constraint is predefined as such, but rather left as a modeling option based on the specific timetabling needs. Additionally, each constraint has several parameters, such as the events or resources it applies to and to what extent (e.g. how many idle times are acceptable during the week), its weight, and other properties, allowing great flexibility.

We now give an informal overview of all the constraints in XHSTT (as reported in [**?**]). There is a total of 16 constraints (plus preassignments of times or resources to events, which are not listed).

Constraints related to events:

1. Assign Time Constraints - assign the specified number of times to specified events.
2. Split Events Constraints - limits the minimum and maximum duration of subevents and the amount of subevents that may be derived from specified events. Distribute Split Events Constraints (below) gives further control on subevents.
3. Distribute Split Events Constraints - limits the number and duration of subevents for specified events.
4. Prefer Times Constraints - specified times are preferred over others for specified events.
5. Avoid Split Assignments Constraints - assign the same resource for all subevents derived from the same event.
6. Spread Events Constraints - specified events must be spread out during the week.
7. Link Events Constraints - specified events must take place simultaneously.
8. Order Events Constraints - specified events must be scheduled one after the other with a specified number of times in between.

Constraints related to resources:

1. Assign Resource Constraints - assign specified resources to specified events.
2. Prefer Resources Constraints - specified resources are preferred over others for specified events.
3. Avoid Clashes Constraints - specified resources cannot be used by two or more subevents at the same time.
4. Avoid Unavailable Times Constraints - specified resources cannot be used at specified times.
5. Limit Idle Times Constraints - specified resources within specified days must have their number of idle times lie between given values.
6. Cluster Busy Times Constraints - specified resources' activities must all take place within a minimum and maximum number of days.
7. Limit Busy Times Constraints - specified resources within specified days must have their number of busy times lie between given values.
8. Limit Workload Constraints - specified resources must have their workload lie between given values.

We give a detailed description of the problem in Section 4.

3 Related work

Heuristic methods were historically the dominant approaches for XHSTT, as they were able to provide good solutions in reasonable amounts of time even when dealing with large instances, albeit not being able to always obtain or prove optimality. Recently exact methods have been proposed and had success in obtaining good results and proving bounds, but require significantly more time (days or weeks). In this section, we discuss the other exact approaches for XHSTT and then give an overview on the heuristic algorithm.

XHSTT has been modeled with Integer Programming (IP) in [**?**]. This exact approach is able to compute good (and in some cases optimal) solutions as well as lower bounds over longer periods of time using Gurobi (a commercial optimization solver).

Satisfiability Modulo Theory (SMT) using bitvectors has been investigated for XHSTT in [?] [?]. The main advantage of the bitvector-SMT approach is to provide compact ways of representing XHSTT which is useful when developing local search algorithms, rather than serving as a complete algorithm.

Furthermore, several IP-based techniques have been introduced for similar HSTT problems which provide bounds and good solutions after long running times [?] [?] [?].

Overall, the majority of the research work has been focused on developing heuristic algorithms. Therefore, we believe our work on using maxSAT for XHSTT is a significant contribution to the exact approach side for XHSTT. Below we continue to list important work for XHSTT that is based on heuristics.

A Large Neighborhood Search algorithm with IP has been developed in [?] which is more efficient than pure IP when given limited time. In a similar style, a maxSAT has been using within a Large Neighborhood Search framework [?], which is based on the maxSAT formulation given in this work. Related to these approaches, a fix-and-optimize IP-based hybrid approach is reported in [?].

All of the best algorithms in the International Timetabling Competition 2011 (ITC 2011) were algorithms based on heuristics. The winner was the group GOAL, followed by Lectio and HySST. In GOAL, an initial solution is generated, which is further improved with Simulated Annealing and Iterated Local Search, using seven different neighborhoods [?]. Lectio uses an Adaptive Large Neighborhood Search [?] with nine insertion methods based on the greedy regret heuristics [?] and fourteen removal methods. HySST uses a Hyper-Heuristic Search [?].

Afterwards, the winning team of ITC 2011 has developed several new Variable Neighborhood Search (VNS) approaches [?]. All of the VNS approaches have a common search pattern: from one of the available neighborhoods, a random solution is chosen, after which a descent method is applied and the resulting solution is accepted if it is better than the previous best. Each iteration starts from the best solution. The most successful VSN algorithm was the Skewed Variable Neighborhood in which a relaxed rule is used to accept the new solution, taking into consideration the cost of the new solution as well as its distance from the best solution. A related approach is Late Acceptance Hill Climbing for XHSTT [?], in which a solution is accepted based on its comparison with the previous k solutions, where k is a parameter.

Kingston [?] introduced an efficient heuristic algorithm called KHE14 which directly focuses on repairing $defects$ (violations of constraints). Constraint violations are examined individually and specialized procedures are developed for most constraints to repair them. The algorithm is designed to provide high quality solutions in a small amount of time, but does not necessarily outperform other methods with respect to solution quality.

A new modeling approach to XHSTT based on bitvectors in given in [?]. This modeling is suitable for local search algorithms and has been shown to provide significant speed ups when compared to the KHE, a leading engine for developing XHSTT algorithms. Additionally, it can also be used to represent XHSTT as a SMT bitvector problem, but the SMT approach is currently not competitive.

Even though significant work has been done for XHSTT, many problems are still not solved efficiently or optimally. Therefore, calculating high quality solutions and providing new modeling approaches are important issues in this domain.

4 Formal description of XHSTT

We give a description of XHSTT, similar as in [?], but with a mathematical formulation. The main entities of XHSTT are events (e.g. lessons that need to be scheduled). Each event has a predetermined duration (e.g. a Math lesson has duration of two hours) and requirements for resources (e.g a lesson requires one teacher and one room). These resource can be predetermined or left open to the solver to assign them. Events can be split into subevents as previously described in Section 2.

Resources are partitioned depending on their resource type (e.g. teachers, students, rooms, etc). Apart from this, resources have no other special properties, but are used extensively in constraints which imposes limits on their usage.

Both events, resources, and times can be grouped in groups. A particular event, resource, or time can be included in any number of groups. Groups are used when defining constraints.

We now discuss the auxiliary functions and variables used in order to describe XHSTT. Note that within the constraint descriptions we will include additional helper functions in order to ease the notation.

4.1 Variables

1. $X_{e,i,t} = 1$ if event e subevent i takes place at time t and $X_{e,i,t} = 0$ otherwise.
2. $S_{e,i,t} = 1$ if event e subevent i starts at time t and $S_{e,i,t} = 0$ otherwise. The starting time of a subevent is the first time in which it takes place.
3. $X_{r,t} = 1$ if resource r is being used at time t by an event and $X_{r,t} = 0$ otherwise.

4.2 Functions

1. $duration(e)$ - gives the duration for an event e. This value is fixed within an instance.
2. $nmbr\_subevents(e)$ - computes the number of subevents event e has been divided into. Note that this number is determined by the end solution and is not fixed up front, although constraints may impose restrictions.
3. $duration\_subevent(e, i)$ - gives the duration of the $i - th$ subevent of event e. Formally, it is the number of consecutive ones in $X_{e,i,t}$.
4. $events(R)$ - computes the set of all events to which resource R has been assigned to. Formally, it returns the set of all events for which the variables $X_{e,i,t,r}$ have nonzero values. Similar to $nmbr\_subevents$, this is determined by the end solution and is not fixed up front, although constraints may impose restrictions.
5. $step(x) = 1$ if $x > 0$, otherwise $step(x) = 0$.
6. $bound\_violation(x, a, b) = |a - x| * step(a - x) + |x - b| * step(x - b)$. If the value x lies in the interval $[a, b]$, it will evaluate to zero. Otherwise, it will evaluate to how far x is from the interval. This function is important as it is frequently used in XHSTT, as a common constraint is that a certain property value should be within a given interval (hard constraint), or as close to it as possible (soft constraint).
7. $equal(a, b) = 1 - step(a - b) - step(b - a)$. Evaluates to one if the arguments a and b are equal and zero otherwise.

4.3 Constraints

Each constraint applies to a subset of events, resources, and times. These will be denoted by the index *spec*, e.g. E_{spec}, T_{spec}, R_{spec}. These subsets are in general different from constraint to constraint. Note that it is possible to have several constraints of the same type, but with different subsets defined for them. For example, the constraint Prefer Times Constraint which states that certain events ($e \in E_{spec}$) are preferred to be scheduled in certain times ($time \in T_{spec}$). For example, we may have two constraints of type Prefer Times Constraint, but each of them can have different sets E_{spec} and T_{spec}, indicating that different events have different preferred times.

The computation of the total violation for a constraint is some in several steps. First, for each *point of application*, the integer value called *deviation* is computed. A cost function is applied to the sum of all deviations and the value produced is multiplied by a weight in order to obtain the total cost for the constraint. Each constraint is labeled as hard or soft. The goal is to minimize the sum of the violations of hard constraints and then minimize the sum of the violations of soft constraints. Note that no constraint is predefined as hard or soft, as this is left for the instance modeler to determine based on specific timetabling needs.

A *point of application* is the object to which the constraint applies. For example, for Prefer Times Constraint a point of application is a single event. The way a deviation is computed is unique and will be described for each constraint individually. Finally, there are three cost functions that can be applied to the deviations: linear, quadratic, and step. The linear makes no changes to the deviation, the quadratic function squares the deviation, while the step function evaluates to 1 if the deviation is non zero and evaluates to 0 otherwise.

We now go through every constraint in detail.

Assign Time Constraints Every event must be assigned a certain amount of times. For example, if a lecture lasts for two hours, two times must be assigned to it. Formally, it imposes that specified events should be assigned times equal to their duration. The point of application is an event and the deviation for a single event e is calculated as follows:

$$deviation_{atc}(e) = (\sum_{i=[1,nmbr\_subevents(e)]} \sum_{t \in T} X_{e,i,t}) - duration(e) \tag{1}$$

Split Events Constraints This constraint has two parts. The first part limits the number of starting times an event may have within certain time frames. For example, an event may have at most one starting time during each day, preventing it from being fragmented within days. The second part limits the duration of the event for a single subevent. For example, if four times must be assigned to a Mathematics lecture, we may limit that the minimum and maximum duration of a subevent is equal to 2, thus ensuring that the lecture will take place as two blocks of two hours, forbidding having the lecture performed as one block of four hours.

Formally, it limits the minimum d_{min} and maximum d_{max} duration of subevents and the minimum A_{min} and maximum A_{max} amount of subevents that may be derived from specified events. The point of application is an event and the deviation for a single event e is calculated as follows:

$$a_{split} = \sum_{i=[0,nmbr\_subevents(e)]} within\_bounds(duration\_subevent(e,i), A_{min}, A_{max}) \tag{2}$$

$$deviation_{splitec}(e) = within\_bounds(nmbr\_subevents(e), d_{min}, d_{max}) + a_{split} \tag{3}$$

Distribute Split Events Constraints This constraint specifies the minimum and maximum number of starting times of a specified duration. For example, if $duration(e) = 10$, we may impose that the lecture should be split so that at least two subevents must have duration three.

Formally, it limits the minimum A_{min} and maximum A_{max} amount of subevents of specified duration d for specified events. The point of application is an event and the deviation for a single event e is calculated as follows:

$$k = \sum_{i=[0,nmbr\_subevents(e)]} equal(duration\_subevent(e,i), d) \tag{4}$$

$$deviation_{dsec}(e) = within\_bounds(k, A_{min}, A_{max}) \tag{5}$$

Prefer Times Constraints This constraint specifies for certain events which times are allowed (hard constraint) or preferred (soft constraint). If an optional parameter d is given, then this constraint only applies to subevents of duration d. For example, a lesson of *duration=2* must be scheduled on Monday, excluding the last time on Monday.

Formally, let $notPrefTimes$ denote the set of times which are not preferred. The point of application is an event and the deviation for a single event e is calculated as follows:

$$deviation_{ptc}(e) = \sum_{i=[0,nmbr\_subevents(e)]} \sum_{t \in notPrefTimes} S_{e,i,t} * duration\_subevent(e,i) \tag{6}$$

If the constraint specified the optional parameter d, then the inner expression of the above equation should be multiplied by $(equal(duration\_subevent(e,i), d))$.

Spread Events Constraints Certain events must be spread across the timetable, e.g. in order to avoid situations in which an event would completely be scheduled only in one day.

Formally, it imposes that minimum A_{min} and maximum A_{max} amount of starting times in specified time groups (sets of times) for events from specified event groups (sets of events). The point of application is an event group and the deviation for a single event group eg is calculated as follows:

$$B_{eg,tg} = \sum_{e \in eg} \sum_{i=[0,nmbr\_subevents(e)]} \sum_{t \in tg} S_{e,i,t} \tag{7}$$

$$deviation_{spreadec}(e) = \sum_{tg \in TG_{spec}} within\_bounds(B_{eg,tg}, A_{min}, A_{max}) \tag{8}$$

Avoid Clashes Constraints Specified resources can only be used by at most one event at a time. For example, a student may attend at most one lecture at any given time.

Formally, it imposes that specified resources cannot be used by two or more subevents at the same time. The point of application is a resource and the deviation for a single resource r is calculated as follows:

$$k(t) = ((\sum_{e \in events(r)} \sum_{i=[0,nmbr\_subevents(e)]} X_{e,i,t}) - 1) \tag{9}$$

$$deviation_{acc}(r) = \sum_{t \in T} step(k(t)) * k(t) \tag{10}$$

Limit Idle Times Constraints This constraint specifies the minimum and maximum number of times in which a resource can be idle during the times in the specified time groups. For example, a typical constraint is to impose that teachers should not have any idle times. An idle time for a resource within a time group is a time in which it is not used, but is being used at some time before or after within the time group. Note that the first and last time within a time group cannot be idle times.

Formally, it imposes the minimum $idle_{min}$ and maximum $idle_{max}$ amount of idle times for specified resources within specified time groups. The point of application is a resource and the deviation for a single resource r is calculated as follows:

$$before(tg, j, r) = step(\sum_{t \in tg \wedge t < j} X_{r,t}) \tag{11}$$

$$after(tg, j, r) = step(\sum_{t \in tg \land t > j} X_{r,t}) \tag{12}$$

$$I(tg, j, r) = before(tg, j, r) * (1 - X_{r,t}) * after(tg, j, r) \tag{13}$$

$$k = \sum_{tg \in TG_{spec}} \sum_{t \in tg} I(tg, t, r) \tag{14}$$

$$deviation_{litc}(r) = within\_bounds(k, idle_{min}, idle_{max}) \tag{15}$$

Avoid Unavailable Times Constraints Specified resources are unavailable at certain times. For example, a teacher might be unable to work on Friday.

Formally, it imposes that specified resources cannot be used at specified times. Let UT denote the set of unavailable times. The point of application is a resource and the deviation for a single resource r is a calculated as follows:

$$deviation_{autc}(r) = \sum_{t \in UT} X_{r,t} \tag{16}$$

Cluster Busy Times Constraints This constraint specifies the minimum and maximum number of specified time groups in which a specified resource can be busy. For example, we may specify that a teacher must fulfill all of his or her duties in at most three days of the week.

Formally, it imposes that specified resources' activities must all take place within a minimum A_{min} and maximum A_{max} amount of specified time groups. The point of application is a resource and the deviation for a single resource r is calculated as follows:

$$a(tg, r) = step(\sum_{t \in tg} X_{r,t}) \tag{17}$$

$$k = \sum_{tg \in TG_{spec}} a(tg, r) \tag{18}$$

$$deviation_{cbtc}(r) = within\_bounds(k, A_{min}, A_{max}) \tag{19}$$

Limit Busy Times Constraints This constraints imposes limits on the number of times a resource can become busy within a certain time group, if the resource is busy at all during that time group. For example, if a teacher teaches on Monday, he or she must teach at least for three hours. This is useful for preventing situations in which teachers or students would need to come to school to attend only a lesson or two.

Formally, it imposes that specified resources the minimum A_{min} and maximum A_{max} amount of busy time within specified time groups. As a special case, if a resource is not busy at all within a time group, then the deviation cost is ignored for that time group. The point of application is a resource and the deviation for a single resource r is calculated as follows:

$$a(tg, r) = \sum_{t \in tg} \tag{20}$$

$$deviation_{lbtc}(r) = \sum_{tg \in TG_{spec}} (within\_bounds(a(tg, r), A_{min}, A_{max}) * step(a(tg, r)) \tag{21}$$

Link Events Constraints Certain events must be held at the same time. For example, physical education lessons for all classes of the same year must be held simultaneously.

Formally, it imposes that specified events from specified event groups must take place at the same time. The point of application is an event group and the deviation for a single event group *eg* is calculated as follows:

$$a(eg,t) = \sum_{e \in eg} \sum_{i=[0,nmbr\_subevents(e)]} X_{e,i,t} \tag{22}$$

$$deviation_{lec}(eg) = \sum_{t \in T} step(a(eg,t)) * (1 - equal(a(eg,t), |eg|)) \tag{23}$$

Order Events Constraints This constraint specifies that one event can start only after another one has finished. In addition to this, optional parameters B_{min} and B_{max} which define the minimum and maximum separations between the two events. The constraint specifies a set of pairs of events to which it applies.

Formally, it imposes that two specified events must take place one after the other with a minimum B_{min} and maximum B_{max} times between them. The point of application is a pair of events and the deviation for a single pair of events *ep* is calculated as follows:

$$a = max\left\{t + duration\_subevent(ep(e1), i) | X_{ep(e1),i,t} = 1\right\} \tag{24}$$

$$b = min\left\{t | X_{ep(e2),i,t} = 1\right\} \tag{25}$$

$$deviation_{oec}(ep) = within\_bounds(b - a, B_{min}, B_{max}) \tag{26}$$

5 Modeling XHSTT as maxSAT

We model XHSTT with Partial Weighted maxSAT. Once a XHSTT instance has been modeled as maxSAT, any satisfiable assignment of cost c for the maxSAT formulation directly corresponds to a XHSTT solution of cost c.

We use two approaches for solving XHSTT. The first one is to model all constraints as maxSAT and give the resulting formula to a maxSAT solver to calculate a solution. The second approach is an iterative one in which we start with a relaxed version of the original problem by omitting all soft constraints and iteratively add them during the solution process. This second approach corresponds to a SMT approach as described in Section 4. As is shown in Section 7, the first approach is more successful than the second one.

We now give a description of the maxSAT problem, the modeling of XHSTT as maxSAT, and give more details on the SMT approach.

5.1 SAT and maxSAT

The Satisfiability problem (SAT) is a decision problem where it is asked if there exists an assignment of truth values to variables such that a propositional logic formula is true (that is, the formula is $satisfied$). A propositional logic formula is built from Boolean variables using logic operators (such as $\wedge$ AND, $\vee$ OR, and $\neg$ NOT) and parentheses. The formula is usually given as a conjunction of clauses (in Conjunctive Normal Form). A clause is a disjunction of literals, where a literal is a variable or its negation. For example, the formula $(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_3)$ has three variables $(X_1, X_2$, and $X_3)$, two clauses, and is said to be satisfiable because there exists an assignment, namely $(X_1, X_2, X_3) = (true, false, false)$, which satisfies the formula. However, had we inserted the clause $(\neg X_1 \vee X_2 \vee X_3)$ the same assignment would no longer satisfy the formula. Instead of writing $\neg X_1$ it is common to write $\overline{X}_1$ and this is the notation used in this work.

The extension of SAT considered in this work is Partial Weighted maxSAT, in which clauses are partitioned into two types: hard and soft clauses. Each soft clause is given a weight. The goal is to find an assignment which satisfies the hard clauses and minimizes the sum of the weights of the unsatisfied soft clauses. For more information about SAT and maxSAT, the interested reader is referred to [**?**].

5.2 Cardinality Constraints

Cardinality constraints impose limits on the truth values assign to a set of literals. These are $atLeast\_k[x_i : x_i \in X]$, $atMost\_k[x_i : x_i \in X]$ and $exactly\_k[x_i : x_i \in X]$, which constraint that at least, at most or exactly k literals out of the specified ones must or may be assigned to true. For example, $atMost\_2\{x_1, x_2, x_3, x_4\}$ would enforce that at most two of the given literals may be assigned true, which would make the assignment $(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$ infeasible and $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$ feasible. Cardinality constraints are used frequently when modeling XHSTT as maxSAT.

We differentiate hard and soft cardinality constraints. Hard cardinality constraints are the traditional ones which strictly forbid certain assignments of truth values to literals. Soft cardinality constraints are similar to hard ones, except that instead of forbidding certain assignments they penalize them and are added to the cost function. In our case, the penalty is greater depending on the severity of the violation. For example, for the soft cardinality constraint $atMost\_2\{x_1, x_2, x_3, x_4\}$, the assignment $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$ would incur no penalty, while assignments $(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$ and $(x_1, x_2, x_3, x_4) = (1, 1, 1, 1)$ would incur a penalty of 1 and 2.

Many different encodings for cardinality constraints exist (e.g. see [**?**], [**?**]), each typically requiring a different amount of auxiliary variables and clauses. In the following we describe the ones used in our implementation in more detail.

Combinatorial Encoding One way to encode the cardinality constraints is to forbid all undesired assignments. We refer to this as the *combinatorial encoding.* In our instances in most cases the exponential growth of clauses is acceptable, but to avoid cases where it would blow up we use an alternative encoding (details given later on in the experimental phases). For example, for $atMost\_2\{x_1, x_2, x_3, x_4\}$ we forbid every possible combination of three literals simultaneously being set to true, giving the following clauses: $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$, $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_4})$, $(\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$ and $(\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$.

Bit Adders The idea is to regard each literal as a 1-bit number, take the sum of all the chosen literals by using a series of adders which sum a binary number and a 1-bit number. The end result is a binary representation of the number of literals set. Appropriate clauses would then be created to forbid specified outputs, which influences which inputs are feasible. For example, for $atMost\_1\{x_1, x_2, x_3\}$ we encode two adders. The first adder computes the sum of x_1 and x_2 and outputs two bits (auxiliary variables) which represent the result of the addition as a binary number (e.g. for $(x_2, x_1) = (1, 1)$, the output will be $(a_{12}, a_{11}) = (1, 0)$). The second adders takes this sum and adds it with x_3 and the results is stored in auxiliary variables a_{22} and a_{21}. Now, in order to encode $atMost\_1$, we add the following clauses which forbid the final result to obtain values 2 and 3 in binary form: $(\overline{a_{22}} \vee \overline{a_{21}})$ and $(\overline{a_{22}} \vee a_{21})$. Therefore, whenever two or more literals are assigned values true one of the two clauses will be unsatisfied, which enforces the constraint $atMost\_1$ as desired. The number of clauses and auxiliary variables is $O(nlog(n))$. We note this encoding use unit propagation to set unassigned literals to false after k_{max} literals are set to *true* as some other encodings and is not optimal with respect to its size, but we wanted to evaluate how well it would perform in practice.

Sequential Encoding This encoding was given in [**?**] for the *atMost_k* case. For completeness, we describe the encoding here with slightly lower number of auxiliary variables as well as include *atLeast_k* case in the encoding. The Sequential encoding [**?**] is closely related to unary numbers. The unary number representation for an integer n as given in [**?**] is:

$$\bigwedge_{\forall i \in [1, n-1]} (u_i \Rightarrow u_{i-1}). \tag{27}$$

The interpretation is that the value assigned with this representation is equal to i, where i is the largest number such that $u_i = \top$. For example, if we wish to encode a variable that can receive values from the interval $[0, 5]$, we need to create five auxiliary variables a_i. If the variable is assigned value 3, then the first three auxiliary variables will be set to true, while the rest will be false: $(a_1, a_2, a_3, a_4, a_5) = (1, 1, 1, 0, 0)$.

We now continue with the Sequential encoding as given in [**?**]. Given a set of literals $\{x_i : i \in [1..n]\}$ for which we wish to encode a cardinality constraint, the main idea of the encoding is to calculate the sum of all literals, similar as in the Bit Adder encoding, but this time using the unary number representation instead of a binary number representation, as addition with unary numbers is simple. This is done by encoding n unary numbers where the i-th unary number represents the i-th partial sum of the literals. We then forbid specified assignments of values to the unary numbers to enforce the desired encoding.

For example, for $atMost\_1\{x_1, x_2, x_3\}$, we require three unary numbers to store three partial sums. For the assignment $(x_1, x_2, x_3) = (1, 0, 1)$, the unary numbers representing partial sums will take values 1, 1 and 2: $(u_{11}, u_{12}, u_{13}) = (1, 0, 0)$, $(u_{21}, u_{22}, u_{23}) = (1, 0, 0)$ and $(u_{31}, u_{32}, u_{33}) = (1, 1, 0)$. Because we are encoding *atMost_1*, we add clauses which forbid the last partial sum to obtain the value 2 and greater, making the previous assignment infeasible. However, if the assignment was $(x_1, x_2, x_3) = (0, 0, 1)$, then the partial sums would be $(u_{11}, u_{12}, u_{13}) = (0, 0, 0)$, $(u_{21}, u_{22}, u_{23}) = (0, 0, 0)$ and $(u_{31}, u_{32}, u_{33}) = (1, 0, 0)$, which is a feasible assignment.

Taking into consideration the presented idea and after doing some optimization to remove redundant clauses, we arrive at the following:

We denote k_{max} and k_{min} to be the maximum and minimum number of literals which may be set to true and with $S_{i,j}$ we denote the j-th variable of the i-th unary number. Note that the i-th unary number representing the i-th partial sum we need $min(i + 1, k_{max})$ auxiliary variables (e.g. for the 1st unary number, which represents the partial sum of the first two literals, there is no need to use more than two auxiliary variables, as the partial sum can be at most two). This fact will also be used in the encoding indicies. Since we are constraining that at most k_{max} can be set, therefore each partial sum only needs to represent a number in the interval $[0, k_{max}]$ (meaning k_{max} auxiliary variables are needed), rather than the complete partial sum which ranges from $[0, n]$.

If the i-th partial sum was greater or equal than m, then $(i + 1)$-th partial sum must also be greater or equal than m:

$$\bigwedge_{\substack{\forall i \in [0..n) \\ \forall j \in [0..,kmax) \\ (i+1 \geq k_{max} \vee j \leq i)}} (S_{i,j} \Rightarrow S_{(i+1),j}) \tag{28}$$

If the i-th literal is set to true, then the i-th partial sum should be at least greater than the $(i - 1)$-th partial sum:

$$\bigwedge_{\substack{\forall i\in[1..n)\\ \forall j\in[1..,kmax)\\ (i+1\geq k_{max}\vee j\leq i)}} (x_i \wedge S_{(i-1),(j-1)} \Rightarrow S_{1,j}) \tag{29}$$

If the i-th literal is set to true, the corresponding i-th partial sum must be equal to at least one (without this constraint, having all partial sums equal to zero would be considered a valid solution):

$$\bigwedge_{\forall i\in[0..n)} (x_i \Rightarrow S_{i,0}) \tag{30}$$

The difference between the $(i+1)$-th and the i-th partial sum cannot be greater than 1:

$$\bigwedge_{\substack{\forall i\in[0..n)\\ \forall j\in[0..,kmax-1)\\ (i+1\geq k_{max}\vee j\leq i)}} (\overline{S_{i,j}} \Rightarrow \overline{S_{(i+1),(j+1)}}) \tag{31}$$

If the difference between the $(i+1)$-th and the i-th partial sum is at least equal to one, this must be because the $(i+1)$-th literal is true:

Corner cases:

$$\bigwedge_{\forall j\in[0..,kmax)} (\overline{S_{i,j}} \Rightarrow x_i) \tag{32}$$

General cases:

$$\bigwedge_{\substack{\forall i\in[0..n-1)\\ \forall j\in[0..,kmax)\\ (i+1\geq k_{max}\vee j\leq i)}} (\overline{S_{i,j}} \wedge S_{(i+1),j} \Rightarrow x_{i+1}) \tag{33}$$

The previous constraints were to ensure that the partial sums are calculated correctly. Note that it is not always necessary that the partial sums are calculated correctly, it is enough to make sure that their values do not exceed the desired value. Because of this, if we only wish to encode *atMost_k*, we can remove 32, since e.g. if $k_{max} = 3$ and we only have one literal set to true, having the partial sums being set to the value three will still be a valid solution, even though they are not correct partial sum. A similar situation holds if only *atLeast_k* is required, where we can ignore 31. Note that if we wish to encode both *atMost_k*$_{max}$ and *atLeast_k*$_{min}$ using the same partial sums (auxiliary variables), then all of the encodings must be included. In our implementation, both equations are always used, as explained in the soft version of this encoding.

Now, in order to encode *atLeast_k*$_{min}$ (with $k_{min} \neq 0$) or *atMost_k*$_{max}$ (with $k_{max} \neq n$), we encode the following:

The last partial sum must be at least equal to k_{min} and the following unit clause enforces this:

$$(S_{n-1,k_{min}-1}) \tag{34}$$

If a partial sum has reached the maximum value k_{max}, then its appropriate variable must be set to false:

$$\bigwedge_{\forall i\in[k_{max}-1..n)} (\overline{x_i} \vee \overline{S}_{(i-1),(k_{max}-1)}) \tag{35}$$

Note that as soon as k_{max} literals are set to true, the remaining unassigned literals will all be forced by unit propagation to be set to false by 35.

This encoding requires exactly $O(nk - k^2)$ auxiliary variables and $O(kn)$ clauses.

Cardinality Networks Cardinality Networks are described in [**?**]. The main idea is to create two types of encodings: one that *sorts* a set of literals and one that *merges* two *sorted* arrays of literals. For *sorting* and *merging*, new auxiliary variables are created which capture the results. To increase clarity we give some examples.

For sorting, if an assignment for literals is $(x_1, x_2, x_3, x_4) = (0, 0, 1, 0)$, an encoding is create which forces the new auxiliary variables to be $(a_1, a_2, a_3, a_4) = (1, 0, 0, 0)$. If the initial assignment was $(x_1, x_2, x_3, x_4) = (1, 0, 0, 1)$, then the auxiliary variables are set to $(a_1, a_2, a_3, a_4) = (1, 1, 0, 0)$.

For merging, if two sets of sorted literals are assigned the following truth values: $(x_1, x_2, x_3) = (1, 1, 0)$ and $(y_1, y_2, y_3) = (1, 0, 0)$, the output auxiliary variables will be forced to the assignments $(a_1, a_2, a_3, a_4, a_5, a_6) = (1, 1, 1, 0, 0, 0)$. One could also view sorted literals as a unary number and their merge as an addition between two unary numbers.

The idea of Cardinality Networks is to sort the given set of literals and then force or forbid certain outputs. For example, if wish to enforce *atMost_k*, we first sort the literals and then forbid the $(k + 1)$-th output, meaning that there cannot be more than $(k + 1)$ literals set to true. For *atLeast_k*, the k-th output is forced to be true, meaning that at least k literals must be set to true. This sorting is performed in a recursive fashion, in similar way to which *mergesort* sorts integers: the set of literals are split into two equal sets, each set is sorted recursively, and then are merged together.

There are a number of intricate details which we do not describe here, but rather direct the interested reader to the original paper [**?**]. The number of auxiliary variables and clauses required for this encoding is $O(nlog^2k)$.

5.3 Soft Cardinality Constraints

Soft cardinality constraints are similar to the previous ones, except that penalize violations of the constraint rather than forbidding it.

Combinatorial Encoding We present the encoding for the soft cardinality constraint $atLeast\_k[x_i : x_i \in X]$, while $atMost\_k[x_i : x_i \in X]$ is done in a similar fashion:

$$\bigwedge_{j \in P} (A_j \rightarrow atLeast\_j[x_i : x_i \in X]) \wedge \bigwedge_{j \in P} (w(j)(A_j)) \tag{36}$$

Where A_i are new auxiliary variables, P is a set of integers in the interval $[1, k]$ and $w(j)$ is a weight function which depends on j, while the *atLeast* is encoded by a basic encoding. The second equation is a series of soft unit clauses containing A_j and its weights are $w(j)$. The auxiliary variables serve as selector variables, which effectively allow or forbid certain assignments, depending on their truth value.

For example, for the encoding of $atLeast\_2\{x_1, x_2, x_3\}$, we obtain the following clauses: $(\overline{a_1} \vee x_1 \vee x_2 \vee x_3)$, $(\overline{a_2} \vee \overline{x_1} \vee x_2 \vee x_3)$, $(\overline{a_2} \vee x_1 \vee \overline{x_2} \vee x_3)$, $(\overline{a_2} \vee x_1 \vee x_2 \vee \overline{x_3})$, $(w\ a_1)$, $(w\ a_2)$. The last two clauses are soft clauses with weights w.

Similar to the case before, an alternative encoding is used to avoid blow ups (details given later on in the experimental phases).

Bit Adders We use bit adder encoding described previously, but instead of forbidding certain outputs, we penalize their assignments. Note that the weights may be assigned to each undesired output completely independently, unlike in the combinatorial encoding.

Sequential Encoding The original version of the Sequential encoding [?] was designed for standard cardinality constraints, not soft ones. We build upon the main idea and extend the encoding for soft cardinality constraints as well.

The main idea is similar as before: calculate the sum of all literals, representing all partial sums as unary numbers. However, in the case of soft cardinality constraints, the values of the partial sums can exceed k_{max}, rather than being capped at k_{max}. This leads to an increase in auxiliary variables and clauses used from $O(nk-k^2)$ to $O(n^2)$. This also reflects in the equations for calculating partial sums, which are the same ones used in the standard cardinality constraint except that we use n instead of k_{max}.

The difference comes in the equations which encode $atLeast\_k_{min}$ and $atMost\_k_{max}$ (34 and 35). Instead, we penalize certain assignments for the last partial sum (which contains the complete sum).

In our instances, we use two different cost functions: linear and quadratic. Each of them penalize assignments to variables based on how distant they are from the interval $[k_{min}, k_{max}]$. For example, for $k_{min} = 2$ and $k_{max} = 4$, if no literals are set to true, then the penalties for linear and quadratic cost functions are 2 and 2^2, respectively. If 7 literals are set true, then the penalties are 3 and 3^2. However, if the number of set literals are which in the interval $[2, 4]$, then no penalty incurs. To model these cost functions for the $atLeast\_k_{min}$ case, we use the following encodings (a similar encoding is used for $atMost\_k_{max}$):

$$\bigwedge_{\forall i \in [1..k_{min})} (w_i(S_{(n-1),(i-1)})) \tag{37}$$

Where $w(i)$ is the associated cost function with the unit clause. For the linear cost function, it is simply a constant $w(i) = c$, while for the quadratic case it is $w(i) = (k_{min} - (i-1))^2 - (k_{min} - i)^2$. In principle, any nonlinear cost function can be modeled by the following way:

$$(w_0(S_{(n-1),0})) \tag{38}$$

$$\bigwedge_{\forall i \in [1..k_{min}]} (w_i(\overline{S}_{(n-1),(i-1)} \vee S_{(n-1),i})) \tag{39}$$

Cardinality Networks Cardinality Networks [?] can be used to model soft constraints and this has been done in [?]. However, when doing so, since every output must be penalized[1], they require more auxiliary variables and degenerate into Sorting Networks [?] from which they offer improvements, meaning the number of auxiliary variables and clauses goes up to $O(nlog^2n)$.

5.4 Special Cases

There are a number of special cases for the encodings which may occur.

A very important special case for $atLeast\_k[x_i : x_i \in X]$ is when $k = |X|$ (a similar case for $atMost\_k[x_i : x_i \in X]$ occurs when $k = 0$) and the weight function $w(j)$ is of the form $w(j) = c * j$, where c is some constant. In this case, instead of using any of the previously described encodings, we encode the following soft unit clauses:

$$\bigwedge_{x_i \in X} ((c)(x_i)) \tag{40}$$

[1] Note that a similar situation happened in the soft version of the Sequential encoding. In the hard version, the partial sums could not exceed value k, while in the soft version they could, which led to an increase in variables and clauses required.

A simple case is when $k_{min} = 1$, in which a single clause which consists of the disjunction of literals in question is required.

Note that $atLeast\_k_{min}$ is equivalent to $atMost\_(n - k_{min})$ of the negated literals. For example, $atLeast\_2\{x_1, x_2, x_3\}$ is equivalent to $atMost\_1\{\overline{x_1}, \overline{x_2}, \overline{x_3}\}$. In our implementation for the combinatorial encoding, we choose to do this conversion if $k > n/2$. This kind of conversion only makes sense for hard cardinality constraints. To clarify this, note that for $atLeast\_k_{min}$ and $atMost\_k_{max}$ we create encodings $atLeast_i$ and $atMost\_k_j$ where $i \in [0, k_{min}]$ and $j \in [k_{max} + 1, n]$. Switching from *atLeast* to *atMost* does not reduce the number of encodings required in the soft case as we need to appropriately penalize all undesired assignments (we assign different penalties to assignments depending on the number of literals assigned to true), while in the hard case we could simply forbid undesired assignments without distinguishing any costs between undesired assignments.

For cases where we use intervals of allowed values (Sequential and Cardinality Networks) it is frequently required that $atLeast\_k_{min}$ and $atMost\_k_{max}$ are encoded on the same literals, and we can perform both the encoding using the same auxiliary variables as described previously. Note that for the combinatorial encoding two independent encodings must be made as there is no sharing of variables or clauses. The number of auxiliary variables and clauses depends on k_{max}, if $n - k_{min} < k_{max}$ we perform the cardinality encoding on the negated literals with $k_{min'new} = n - k_{kmax}$ and $k_{max'new} = n - k_{min}$. Once again, this kind of conversion only applies for hard cardinality constraints for similar reasons as before.

5.5 XHSTT constraints as maxSAT

In practice, some constraints are never used as soft constraints (e.g. a student cannot attend two lessons at the same time). Because of this, we only give the encodings for soft constraint where it is appropriate in order to avoid unnecessary technicalities.

We simplify the objective function by not tracking the infeasibility value, rather regarding it was zero or nonzero. That is, we encode hard constraints of XHSTT as hard clauses and we do not distinguish between two different infeasible solution in terms of quality. By doing so we simplify the computation, possibly offering a faster algorithm.

As noted in 4.3, each constraint applies to a subset of events, resources, times, etc. These will be denoted by the index *spec*, e.g. E_{spec}, T_{spec}, R_{spec}. We now give the modeling of constraints described in 4 as Partial Weighted maxSAT.

Assign Time Constraints We define decision variables $Y_{e,t}$ and other constraints rely on them heavily. For each $e \in E$ and $t \in T$, variable $Y_{e,t}$ indicates whether event e is taking place at time t. Each event must take place for a number of times equal to its duration d:

$$\bigwedge_{\forall e \in E} (exactly\_d[Y_{e,t} : t \in T]) \tag{41}$$

Avoid Clashes Constraint We introduce variables $Y_{e,t,r}$ which indicate whether event e at time t is using resource r. If an event is using a resource at a time, that means that the event must also be taking place at the same time:

$$\bigwedge_{\substack{\forall e \in E \\ t \in T \\ r \in R}} (Y_{e,t,r} \Rightarrow Y_{e,t}) \tag{42}$$

Let $E(r)$ be the set of events which require resource r. The constraint is encoded as follows:

$$\bigwedge_{\substack{\forall r \in R_{spec} \\ t \in T}} (at\_Most\_1[Y_{e,t,r} : e \in E(r)]) \tag{43}$$

Avoid Unavailable Times Constraints In order to keep track whether a resource r is busy at time t, we introduce auxiliary variables $X_{t,r}$ for each resource. They are defined as:

$$\bigwedge_{\substack{\forall r \in R \\ t \in T}} (X_{t,r} \Leftrightarrow \bigvee_{e \in E(r)} Y_{e,t,r}) \tag{44}$$

We now encode the previously described constraint by forbidding assignments at specified times:

$$\bigwedge_{\forall r \in R_{spec}} (atMost\_0[X_{t,r} : t \in T_{spec}]) \tag{45}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead.

Split Events Constraints In the formal specification of XHSTT, any time can be defined as a starting time as events can be split into multiple subevents. One could regard a starting point as a time t where a lecture takes place, but has not took place at $t-1$. However, while this is true, this cannot be the only case when a time would be regarded as a starting time, since e.g. time $t = 5$ and $t = 6$ might be interpreted as *last time of Monday* and *first time of Tuesday* and an event could be scheduled on both of these times, but we may regard both times as starting times. It is also worthy to note that we can also regard that as a double (block) lecture, even though it spans over two days (this is case in the Brazilian instances). Having such a double lecture is not the desired double lecture, but is still better than splitting the lecture into two lectures and assigning them in another fashion. Therefore, any time can in general be regarded as a starting time. Other constraints give more control over these kind of assignments.

For each event e, variable $S_{e,t}$ indicates whether event e has started taking place at time t. For example, if event e had a duration of two and its corresponding $Y_{e,t}$ were assigned at times t and $t+1$, then $S_{e,t} = true$, $S_{e,(t+1)} = false$. Formalities that are tied to starting times with regard to the specification are expressed as follows:

Event e starts at time t if e is taking place at time t and it is not taking place at time $(t-1)$:

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ t \in T}} (Y_{e,t} \wedge \overline{Y}_{e,(t-1)} \Rightarrow S_{e,t}) \tag{46}$$

Note that the other side of the implication does not hold (see first paragraph on this section). If a starting time for event e has been assigned at time t, then the corresponding event must also take place at that time:

$$\bigwedge_{\substack{\forall e \in E \\ t \in T}} (S_{e,t} \Rightarrow Y_{e,t}) \tag{47}$$

This constraint specifies the minimum A_{min} and maximum A_{max} amount of starting times for the specified events:

$$\bigwedge_{\forall e \in E_{spec}} (atLeast\_A_{min}[S_{e,t} : t \in T] \wedge atMost\_A_{max}[S_{e,t} : t \in T]) \tag{48}$$

In addition, this constraint also imposes the minimum d_{min} and maximum d_{max} duration for each subevent. For each specified event $e \in E_{spec}$, and duration d, variable $K_{e,t,d}$ indicates that event e has a starting time at time t of duration d. Formally:

If time t has been set as a starting time, associate a duration with it[2]:

[2] Remark: We could had encoded that exactly one of the right hand sides literals must be chosen, but this is handled in the later parts of this encoding.

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T}} (S_{e,t} \Rightarrow \bigvee_{d_{min} \leq d \leq d_{max}} K_{e,t,d}) \tag{49}$$

When $K_{e,t,d}$ is set, the event in question must take place during this specified time (where set D is the set of integers from the interval $[d_{min}, d_{max}]$):

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T \\ d \in D}} K_{e,t,d} \Rightarrow \bigwedge_{i \in [0,d-1]} Y_{e,(t+i)} \tag{50}$$

If a duration has been specified for time t, make sure that other appropriate $K_{e,t,d}$ variables must be false:

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T \\ d \in D}} (K_{e,t,d} \Rightarrow \bigwedge_{d_{min} \leq g \leq d_{max}} \bigwedge_{i \in [0,d-1] \wedge (i \neq 0 \vee g \neq d)} \overline{K}_{e,t+i,g}) \tag{51}$$

If a subevent of duration d has been assigned and immediately after the event is still taking place, then assign that time as a starting time:

$$K_{e,t,d} \wedge Y_{e,t+d} \Rightarrow S_{e,t+d} \tag{52}$$

Prefer Times Constraints The constraint is encoded as:

$$\bigwedge_{\forall e \in E_{spec}} (atMost\_0[\star : t \in T \setminus T_{spec}]) \tag{53}$$

where $\star$ is either $Y_{e,t}$ or $K_{e,t,d}$, depending on whether the optional parameter d is given. Note that this constraint is not the same in general when the optional parameter is not given and when $d = 1$.

Distribute Split Events Constraint There must be at least A_{min} starting times with given duration d:

$$\bigwedge_{\forall e \in E_{spec}} (atLeast\_A_{min}[K_{e,t,d} : \forall t \in T]) \tag{54}$$

There must be at most A_{max} starting times with given duration d:

$$\bigwedge_{\forall e \in E_{spec}} (atMost\_A_{max}[K_{e,t,d} : \forall t \in T]) \tag{55}$$

Similar as with $S_{e,t}$, for the last $d-1$ times, $K_{e,t,d}$ are set to $false$ and can be removed from the equations.

Spread Events Constraints First, we introduce auxiliary variables $Z_{eg,t}$.

An event group eg is a set of events. Variable $Z_{eg,t}$ indicates that an event from event group eg is being held at time t. Formally,

$$\bigwedge_{\substack{eg \in EG_{spec} \\ t \in T}} (Z_{eg,t} \Leftrightarrow \bigvee_{e \in eg} S_{e,t}) \tag{56}$$

This constraint specifies event groups to which it applies, as well as a number of time groups (sets of times) and for each such time group the minimum and maximum number of starting times an event must have within times of that time group. Note that an event group may consist of a single event and that it is not the same to have two event groups with one event and one event group with two events. Continuing with the constraint encoding, let TG_{spec} denote this set of sets of times:

There must be at least d_i^{min} starting times within the given time groups (min is a subscript, not exponentiation):

$$\bigwedge_{\substack{\forall tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atLeast\_d_i^{min}[Z_{eg,t} : t \in tg_i]) \tag{57}$$

There must be at most d_i^{max} starting times within the given time groups:

$$\bigwedge_{\substack{\forall tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atMost\_d_i^{max}[Z_{eg,t} : t \in tg_i]) \tag{58}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead. We note that if two events from the same event group are being held at the same time, the above encoding will not be properly capture the constraint. However, given the nature of the constraint, all events within the same event group are likely to share a resource and Avoid Clashes Constraint will make sure that they do not take place at the same time. This is the case with every instance in XHSTT.

Limit Busy Times Constraints Once more we first encode auxiliary variables:

A resource is busy at a time group tg iff it is busy in at least one of the times of the tg. Let TG_{spec} denote this set of sets of times:

$$\bigwedge_{\substack{\forall r \in R \\ \forall tg \in TG_{spec}}} (B_{tg,r} \Leftrightarrow \bigvee_{t \in T} X_{t,r}) \tag{59}$$

The constraint is now encoded as:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R_{spec}}} (B_{tg,r} \Rightarrow atLeast\_b_{min}[X_{t,r} : t \in tg] \wedge atMost\_b_{max}[X_{t,r} : t \in tg]) \tag{60}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead.

Cluster Busy Times Constraints Recall the auxiliary variables $B_{tg,r}$ defined in Section 5.5. There must be at least b_{tg}^{min} busy time groups:

$$\bigwedge_{\forall r \in R_{spec}} (atLeast\_b_{tg}^{min}[B_{tg,r} : tg \in TG_{spec}]) \tag{61}$$

There must be at most b_{tg}^{max} busy time groups:

$$\bigwedge_{\forall r \in R_{spec}} (atMost\_b_{max}^{tg}[B_{tg,r} : t \in TG_{spec}]) \tag{62}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead.

Limit Idle Times Constraints To encode the constraint, three different types of auxiliary variables are used.

Variables $G^{tg}_{t,r}$ and $H^{tg}_{t,r}$ indicate that a resource is being used strictly before or strictly after the $t-th$ time in time group tg, where a time group tg is viewed as an ordered list. Formally:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (G^{tg}_{t,r} \Leftrightarrow \bigvee_{i\ before\ t\ \wedge\ i \in tg} X_{i,r}) \tag{63}$$

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (H^{tg}_{t,r} \Leftrightarrow \bigvee_{i\ after\ t\ \wedge\ i \in tg} X_{i,r}) \tag{64}$$

The first and last time within a group can never have their appropriate $G^{tg}_{t,r}$ and $H^{tg}_{t,r}$ be set to $true$, respectively, and can be excluded from the above equation.

Variables $I^{tg}_{t,r}$ indicates that a resource is idle at time t with respect to time group tg (an ordered list of times) iff it is not busy at time t, but is busy at an early time and at a later time of the time group tg. For example, if a teacher teaches classes Wednesdays at $Wed2$ and $Wed5$, he or she is idle at $Wed3$ and $Wed4$, but is not idle at $Wed1$ and $Wed6$. Formally,

$$\bigwedge_{\substack{tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (I^{tg}_{t,r} \Leftrightarrow \overline{X}_{t,r} \wedge G^{tg}_{t,r} \wedge H^{tg}_{t,r}) \tag{65}$$

We now encode the constraint:
There must be at least $idle_{min}$ idle times during a time group:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R}} (atLeast\_idle_{min}[I^{tg}_{t,r} : t \in tg]) \tag{66}$$

There must be at most $idle_{max}$ idle times during a time group:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R}} (atMost\_idle_{max}[I^{tg}_{t,r} : t \in tg]) \tag{67}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead.

Order Events Constraints If the first event in a pair is taking place at time t, then the second event cannot take place at time t nor at any previous times (E^2_{spec} is the set of pairs of events given in the constraint):

$$\bigwedge_{\substack{\forall (e_1,e_2) \in E^2_{spec} \\ \forall t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{i \in [0,t]} \overline{Y}_{e_2,i}) \tag{68}$$

If the first event in a pair is taking place at time t, then the second event cannot take place in the next B_{min} times:

$$\bigwedge_{\substack{\forall (e_1,e_2) \in E^2_{spec} \\ \forall t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{j \in [t+1, t+B_{min}]} \overline{Y}_{e_2,j}) \tag{69}$$

If the first event in a pair is taking place at time t, then the second event must take place within the next $(B_{max}+1)$ times:

$$\bigwedge_{\substack{\forall (e_1,e_2)\in E^2_{spec}\\ \forall t\in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{k\in[t,t+B_{max}+1]} Y_{e_2,k}) \tag{70}$$

Link Events Constraints This constraint specifies a certain number of event groups and imposes that all events within an event group must be held simultaneously. Let EG_{spec} denote this set of sets of events:

All events within an event group must be held at the same times:

$$\bigwedge_{\substack{\forall eg\in EG_{spec}\\ t\in T\\ e_j,e_k\in eg}} (Y_{e_j,t} \Leftrightarrow Y_{e_k,t}) \tag{71}$$

If the constraint is declared a soft one, we may apply a similar technique that was presented when soft cardinality constraint were shown: create an auxiliary variable which implies every clause and insert a soft unit clause containing that auxiliary variable along with the appropriate weight. However, with this encoding only the sum of deviations cost function can be encoded, which is the only cost function used in the instances for this constraint.

Preassign Resource Constraints We define decision variables which indicate whether an event is using a resource at a time. If an event is using a resource at some time, the event must take place at that time:

$$\bigwedge_{\substack{\forall e\in E_{spec}\\ t\in T\\ r\in R^e_{spec}}} (Y_{e,t} \Rightarrow Y_{e,t,r}) \tag{72}$$

In the specification of the general XHSTT, this constraint is given when events are defined, rather than a separate constraint.

Preassign Time Constraints Similar to Preassign Resource Constraints, certain events have a fixed schedule. For example, an external professor is available only on Monday from 8:00-10:00.

This consist of adding a series of unit clauses of the appropriate $Y_{e,t}$.

Assign Resource Constraints Each event requires a certain amount of resources in order to be scheduled. These resources can be teachers, classes, rooms, etc. For example, in order for a math lesson to take place a math teacher, a room, and a projector are needed. It might also be the case that two teachers are needed, e.g. one lecturer and one as an assistant. This has been implemented into the general HSTT specification as follows:

Each event has a number of *roles*. To each of these *roles* exactly one resource of a specific resource type must be assigned. The *role* names within an event must be unique, but different events may have the same *roles* requiring different types of resources. For example, an event might require the following roles with the appropriate resource types given in parenthesis: 'Teacher' (teacher), 'Assistant' (teacher), 'Class' (class), 'Seminar room' (room). This constraint merely requires that a resource of a given type must be assigned. For the given *role*, a variable $M^{role}_{e,t,r}$ is created, which indicates whether event e at time t is using resource r to fulfill the given *role*. The constraint is encoded as follows:

If an event is taking place, it's specified *role* must be fulfilled:

$$\bigwedge_{\substack{e\in E_{spec}\\ t\in T}} (Y_{e,t} \rightarrow exactly\_1[M^{role}_{e,t,r} : r \in R_{spec\_resource\_type}]) \tag{73}$$

If a resource has been chosen to fulfill an event's role at some time, mark that resource as used by the event at that time:

$$\bigwedge_{\substack{e\in E_{spec}\\ t\in T\\ r\in R_{spec\_resource\_type}}} (M_{e,t,r}^{role} \rightarrow Y_{e,t,r}) \tag{74}$$

The previous two encodings hold individually for each Assign Resource Constraint. The next encoding is done after all constraints of type Assign Resource Constraints and is in a sense a global constraint:

Avoid Split Assignments Constraint This constraint applies to the specified *role* and to a specified resource type. We create auxiliary variables $V_{e,r}^{role}$ which indicate whether an event e is using a resource r to fulfill its *role* at some point in time:

$$\bigwedge_{\substack{e\in E_{spec}\\ R_{spec\_resource\_type}\\ t\in T}} (M_{e,t,r}^{role} \rightarrow V_{e,r}^{role}) \tag{75}$$

The constraint is now encoded as:

$$\bigwedge_{e\in E_{spec}} (atMost\_1[V_{e,r}^{role} : r \in R_{spec\_resource\_type}]) \tag{76}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead.

Prefer Resources Constraints Similar to before, this constraint applies for a specified *role*. The encoding relies on auxiliary variables create in ARC:

$$\bigwedge_{\substack{\forall e\in E_{spec}\\ \forall t\in T}} (atLeast\_1[M_{e,t,r}^{role} : r \in R \setminus R_{spec}]) \tag{77}$$

Limit Workload Constraints We do not provide the general formulation, but rather focus on an important special case which is used in the instances. For each resource assigned to a subevent (solution resource sr), we calculate it's workload as:

$$Workload(sr) = Dur(subevent) * Workload(subevent)/Dur(event) \tag{78}$$

Where the workload of an subevent is by default equal to the duration of the event, but can be specified differently in the definition of the event. If events all have their default values for their workload ($workload(e) = duration(e)$) (which is the case in the instances), then the encoding can be significantly simplified. The observation here is that the formula simplifies to the case where every unit of time in which a resource is busy counts as one workload unit, if the resource does not have a preassigned workload in which case it is preassigned to the event. However, for the purpose of encoding, we may treat all resources as not having preassigned workloads, but subtract from the given minimum and maximum workload by the constraint by an amount equal to the preassigned workload minus one and do so for each event in which the resource has a preassigned workload. The constraint is now simply encoded as:

$$\bigwedge_{r\in R_{spec}} (atLeast\_Work_{min}[X_{t,r} : t \in T] \wedge atLeast\_Work_{max}[X_{t,r} : t \in T]) \tag{79}$$

Special Cases In this section we look into important special cases which may simplify the encodings significantly.

If an Assign Resource Constraint is given and all of the resources it references behave the same, then instead of encoding Assign Resource and Avoid Clashes Constraints for those resources, we may use the following encoding:

$$\bigwedge_{t \in T} (atMost\_h[Y_{e,t} : e \in E_{spec}]) \tag{80}$$

Where E_{spec} are events that require the mentioned resources and h is number of resources of the described kind. This case arises in EnglandStPaul and FinArtificialSchool instance and allows us to encode these two instances, even though we do not model Assign Resource Constraints in general.

If there is only one *role* per resource type specified in the requirements of an event, then the encoding of the auxiliary variables in Assign Resource Constraints may be avoided.

If the resources specified in Assign Resource Constraints are not subjected to Limit Idle Constraints and assigning more than one resource to an event may be feasible, then a simpler encoding may be used for ARC, in which *atLeast_1* is used instead of using *exactly_1*. This case happens typically in instances which require the assignment of rooms. If two rooms are assigned to an event, in the solution we would simply pick only one. However, this cannot be applied in general e.g to teachers.

Another problem with ARC is that certain symmetries may arise, increasing the solution time. For example, if we have two ARCs, each with their specified *role*. If these two roles both use the same resource type and no further constraints are imposed on these resources, then we may swap their assignments of resources and still get the same in(feasible) solution, which is undesirable. Therefore, encoding a sorting is very useful and can be done efficiently since the unary representation is used.

In some cases, by knowing the semantics of each constraint, simpler encodings can be produced. This is encountered in SpainInstance in which a large amount of Spread Events Constraints are encoded which state that lessons can have at most one starting point in two consecutive days. However, this is not trivial to specify in the general HSTT specification and will produce a large number of clauses, which could be avoid if a special encoding for such a constraint is encoded.

Another interesting case is the encoding of $K_{e,t,d}$. These are created in order to comply with the formal specification of XHSTT. In some cases, it suffices to encode $K_{e,t,d}$ as (i is an integer):

$$K_{e,t,d} \leftrightarrow (\bigwedge_{i \in [0..d-1]} Y_{e,t+i}) \wedge \overline{Y}_{e,t+d} \tag{81}$$

This encoding is much more desirable when it is possible and we can use it e.g. in the ItalyInstance1, where the general encoding took around 50 hours to compute the optimal solution, while with the change shown above took around 10 hours. If the encoding is used, other constraints might be affected, such as Split Events Constraint and need to be changed accordingly. However, in our current implementation these cases need to be done by hand.

6 SMT approach

Two SMT approaches have previously been investigate in [?] and [?] (linear arithmetic and bitvectors) and the authors reported that the approaches were not very competitive in their current state. Here we investigate a different SMT approach.

With this approach, we start with a relaxed version of a XHSTT instance by omitting the soft constraints. After a solution has been found for this simplified problem, the solution is examined with respect to the original problem and soft constraint violations as detected. These violated constraints are encoded as maxSAT and are inserted into the relaxed formula and the solution process is initiated again. This is done iteratively until a solution is found such that all constraints that have not been added are satisfied. In this case, the solution found is optimal. The idea is that perhaps not all constraints from the original XHSTT instance are needed to find the optimal solution, but only a subset of them, and that it will be able to find the solution faster with less constraints.

All of the constraints are encoded as described in the previous section, except for the soft cardinality constraints, essentially making all soft constraints $atLeast\_0$ or $atMost\_n$ (where n is the number of literals in that constraint). When it is concluded that the original constraint is violated, the degree of the cardinality constraint is changed by one step (e.g. $atLeast\_i$ will become $atLeast\_(i+1)$ and $atMost\_j$ will become $atMost\_(j+1)$). This change is performed by inserting the appropriate clauses. Eventually, if the constraint keep getting violated from iteration to iteration, the complete cardinality encoding will be inserted.

There are some exceptions. When the constraint $atMost\_0$ is violated and the cost function linear with respect to the number of violating literals, we insert unit clauses $(w\ \overline{x}_i)$ for each violating literal, where w is the constraint weight and x_i is the violating literal.

The other exceptions are the Prefer Times Constraint and Limit Idle Times Constraint. These constraint typically employ $atMost\_0$ cardinality constraints and when a violation is detected, we not only insert clauses as described above, but also insert clauses for each literal that lies within the same day of the violation.

Note that when a solution of cost c is found for a relaxed version of a XHSTT instance, c is a lower bound for the original XHSTT problem. An outline of the algorithm is given in Algorithm 1. With ub and lb we denote the upper and lower bound respectively, with $maxSATcost$ the cost of the solution with respect to the maxSAT problem currently analyzed, while $XHSTTcost$ is the cost with respect to all constraints of the problem.

Algorithm 1: SMT for XHSTT algorithm outline

```
begin
    $I_{relax} \longleftarrow encodeHardConstraints(I_{original})$
    $ub = \infty$
    $lb = 0$
    $bestAssignment = \emptyset$
    $globally\_solved = false$
    while $globally\_solved = false$ do
        $a = maxSATsolve(I_{relax})$
        $cla = encodeViolationsNotEncoded(a)$
        if $lb < maxSATcost(a)$ then
            $lb = cost(a)$
        if $ub > XHSTTcost(a)$ then
            $ub = XHSTTcost(a)$
        if $cla = \emptyset \vee lb = ub$ then
            $globally\_solved = true$
        else
            $I_{relax} = I_{relax} \cup cla$
```

6.1 Technical details

We developed a SMT algorithm which is based on an upper bounding maxSAT algorithm. In order to explain our approach, we first explain the Linear algorithm for maxSAT.

With regard to the traditional decision problem, the problem of solving a SAT instance while fixing certain variables is known as "solving under assumptions". This can be done by having the solver first "branch" on the fixed variables and then continue doing a regular SAT search. However, this kind of technique cannot be directly used for maxSAT because the underlying formula is being changed during the solution process. We elaborate on this further below.

We use the the Linear maxSAT algorithm (Algorithm 2) [**?**] which makes repeated calls to a SAT solver and after each call adds constraints which ask for a better solution than the previous one. The optimal solution is obtained when the SAT solver returns $false$. We opted for a linear algorithm rather than a core guided approach because authors in [**?**] reported significantly better results using a linear algorithm solver rather than core guided solvers.

Algorithm 2: Linear algorithm for maxSAT

```
begin
    $P \longleftarrow maxSAT\ formula$
    $c = \infty$
    $bestAssignment = \emptyset$
    while $isSatisfiable(P)$ do
        $bestAssignment = satisfiableAssignment(P)$
        $c \longleftarrow cost(P, bestAssignment)$
        $P = P \cup (\sum_{i \in K} softConstraint(i) < c)$
```

The original maxSAT formula gets changed because bounds are added at each iteration, in addition to *learned* clauses which are added to direct the search (see [**?**] for clause learning). It is not straightforward to remove the added clauses at later stages of the algorithm, because clauses are learned with respect to other clauses (including other learned clauses) and removing some clauses may therefore invalidate previously learned clauses. To the best of our knowledge, no maxSAT solver supports this kind of search. An alternative is to restart the solver after each call, losing possibly valuable learned clauses and bounds. This motivated us to investigate a different approach: instead of restarting between calls, we keep the modified formula intact. Thus, each call to the solver depends on all previous calls due to the bounds and learned clauses. When querying the solver with a new set of assumptions, it will attempt to report the best possible solution, but only if it is better than all of the previously computed solutions. To this end, we modified the linear algorithm in the open-source maxSAT solver Open-WBO [**?**]. A different approach related to ours is presented in [**?**] for lower bounding maxSAT algorithms.

In our SMT approach, each time a new optimal solution with respect to the currently considered soft constraints is found it is examined and new soft constraints are inserted, meaning that the sum of the weights of soft constraints changes and the previously inserted clauses regarding the cardinality constraints (third line in the *while* loop of 2) should be invalidated. In order to invalidate them, we performed the following: when inserting the clauses, we add them as usually, but add a negative literal $\overline{b}$ to each clause. During the current iteration, this literal is treated as an assumption which assigns it $false$ ($b = true$). When the cardinality constraints inserted need to be invalidated, this is simply done by inserting a hard unit clause ($\overline{b}$), which forces the assignment of $b = false$, making all the clauses added as cardinality constraints for the previous iteration satisfiable (effectively invalidating them).

We above is summarized in in Algorithm 3 and 4.

Algorithm 3: SMT for XHSTT algorithm outline

```
begin
    $I_{relax} \longleftarrow encodeHardConstraints(I_{original})$
    $ub = \infty$
    $lb = 0$
    $bestAssignment = \emptyset$
    $globally\_solved = false$
    while $globally\_solved = false$ do
        $v \longleftarrow createNewVar()$
        $a = modifiedMaxSATsolve(I_{relax}, v)$
        $cla = encodeViolationsNotEncoded(a)$
        if $lb < maxSATcost(a)$ then
            $lb = cost(a)$
        if $ub > XHSTTcost(a)$ then
            $ub = XHSTTcost(a)$
        if $cla = \emptyset \vee lb = ub$ then
            $globally\_solved = true$
        else
            $I_{relax} = I_{relax} \cup cla$
        $I_{relax} = I_{relax} \cup (\overline{v})$
```

Algorithm 4: Modified Linear algorithm for SMT, using v as an input variable

```
begin
    $v \longleftarrow input\ variable\ given\ as\ parameter$
    $P \longleftarrow maxSAT\ formula$
    $c = \infty$
    $bestAssignment = \emptyset$
    while $isSatisfiableUnderAssumption(P, \overline{v})$ do
        $bestAssignment = satisfiableAssignmentUnderAssumption(P, \overline{v})$
        $c \longleftarrow cost(P, bestAssignment)$
        $cla = encodeAsClauses(\sum_{i \in K} softConstraint(i) < c)$
        $i = 0$
        while $i < |cla|$ do
            $cla[i] = cla[i] \vee v$
            $i++$
        $P = P \cup cla$
```

7 Computational Results

We have set several goals in order to evaluate the maxSAT approach for XHSTT and they are as follows:

- Compare the performance of different maxSAT solvers on XHSTT instances.
- Compare different cardinality constraint encodings for XHSTT.
- Compare maxSAT with Integer Programming for XHSTT.
- Compare maxSAT with our SMT approach (Section 6).
- See how well our maxSAT approach would do if it was used in the International Timetabling Competition 2011.

7.1 Benchmark instances and Computing Environment

We evaluated our approach on XHSTT benchmark instances which can be found on the repository of the International Timetabling Competition 2011 (ITC 2011) [3]. We used the XHSTT-2014 benchmark set, which contains instances that were careful selected by the ITC 2011 over the years and are meant to be interesting test beds for researchers. Additionally, we included every instance used in the competition (these two sets of instances overlap). This way we took into consideration all relevant XHSTT instances, to the best of our knowledge.

In total we can model efficiently with maxSAT 27 out of 39 (70%) instances. We have a specific modeling for resource assignments (Assign Resource Constraints and related constraints) for two cases (FinArtificialSchool, EnglandStPaul, see Section 5.5), but for other instances with resource assignments our current model is not practical. Unfortunately, in this case the number of produced variables and clauses is very large, and until now we could not come up with a more efficient encoding for these constraints. Thus, for the remaining 12 instances, we currently do not have an appropriate model and could not have experimented with them.

In the instances, the number of times ranges from 25 to 125, number of resources from 8 to 99, number of events from 21 to 809 with total event duration from 75 to 1912. These numbers vary heavily from instance to instance. We do not provide detailed information, but direct the interested reader to [**?**,**?**].

We have submitted XHSTT maxSAT instances to the maxSAT Competition 2014 and they have been used since. Over the few years they have proven to be challenging instances for maxSAT solvers. After the submission, some XHSTT instances have been slightly changed. The maxSAT encodings used in this paper can be found here: (`www.dbai.tuwien.ac.at/user/demir/xHSTTtoSAT_instances.tar.gz`).

MaxSAT experiments were done on a benchmark server with a AMD Opteron Processor 6272 2.1GHz with two processors. Each processor has each eight physical cores and each core puts at disposal two logical cores (per hyperthreading). The machine has a total of 224 GB of RAM (14 x 16GB). When experimenting we initiated the solving of 16 instances in parallel.

IP experiments were performed on a machine with an Intel Core i5-4210U Processor with 2.7 GHz and 4 GB of RAM. The reason why experiments were performed on different machines is because the solvers require different operating systems. When experimenting we solved a single instance at a time. For both IP and maxSAT each solver was run with a single thread.

ITC 2011 issued a benchmark tool which is designed to test how fast a machine performs operations relevant for timetabling. The tool estimates how long a XHSTT solver should run on the machine at hand in order for it to be equivalent to 1000 seconds on ITC's computer (the computational time limit for ITC's second phase of the competition). The intent is to provide grounds for determining some normalized time across different platforms. For our maxSAT benchmark server

[3] http://www.utwente.nl/ctit/hstt/itc2011/welcome/

the suggested time was around 1250 seconds. Given that our maxSAT approach is an exact approach we decided to allocated roughly ten times more time, for a total of four hours. For IP, the equivalent time is around 2.2 hours.

7.2 Notation

In tables we shall note the cost function for instances as (x, y), where x is the infeasibility value (sum of the cost functions of hard constraints) and y is the objective value (sum of the cost functions of soft constraints). For example, (3, 35) denotes that the infeasibility value is 3 and the objective value is 35. If the infeasible value is equal to zero, we say that the solution is feasible, otherwise it is infeasible.

7.3 Solvers

We chose to experiment with maxSAT solvers WPM3 [?], Open-WBO [?], and Optiriss (a combination of the Riss framework [?] and Open-WBO [?]). The first two solvers were selected because they were the best solvers for timetabling instances in the Industrial Weighted Partial maxSAT category in the maxSAT Competition 2016, and since Open-WBO was used in Optiriss we decided to include it as well.

Both Optiriss and Open-WBO allows its users to configure the solvers by selecting among several maxSAT algorithms and parameters. We used the default configuration for Open-WBO and the two configurations of Optiriss (*Optiriss-def* and *Optiriss-inc*) that were used in the maxSAT Competition 2016. In addition to this, we used the same configuration but have set the solvers to use the Linear maxSAT algorithm [?] (see Algorithm 2) because this algorithm already previously showed good performance for XHSTT in [?] (this can be done by adding the parameter *-algorithm=1* to either solver). Therefore, if we consider different configuration of solvers as stand alone solvers themselves, we experimented with a total of seven solvers.

7.4 Evaluation of different maxSAT solvers

We compare the performance of different maxSAT solvers on XHSTT instances. In order to do so, we used a similar ranking system as the ITC 2011. We run all solvers on each instance for four hours and record the solution. For each instance we compute the rank for each solver. The rank is a number between one and seven and it represents how well the solver did relative to other solvers. For a given instance, the best solver has rank one, the second best has rank two, etc. Solvers can share the same rank in case of ties. In Table 1 and Table 2 we show the results and the ranking of solvers for this comparison.

Based on these results, we conclude that the default configuration of Open-WBO has the best average rank. However, the average rank of 2.16 indicates that there is no clear winner as the results are not uniform across instances. While overall Open-WBO performs the best on average, we can see that for a number of instances it gets outperformed by other solvers. Therefore, we decided to select k solvers with complement each other instead of determining a single winner. In order words, we wish to select k solvers such that if we run them in parallel and take the best result, we obtain good results across all instances. Formally, we would like select k solvers in order to minimize the combined rank $\frac{\sum_{i \in I} min(rank(s,i) : s \in S)}{|I|}$, where I is the set of instances, S is the set of selected solvers, and $rank(i, s)$ is the rank of solver s on instance i.

In order to determine which solver to chose and how many (the parameter k) we modeled the described problem as a maxSAT optimization problem and used Open-WBO to compute the optimum solution for every k. We show the optimum combined rank for every choice of k as a pair (k, rank): (1, 2.2), (2, 1.4), (3, 1.2), (4, 1.1), (5, 1), (6, 1), (7, 1). Based on these results, we chose $k = 4$

in order to keep the combined rank close to one. One solution for $k = 4$ is Open-WBO (def), Open-WBO (lin), Optiriss (inc), and Optiriss (default-linear). These four solvers are the best maxSAT solvers for XHSTT according to our criteria. Each of them have their own strengths and weaknesses and they will be used for further experimentation. We note that there are other combinations of four solvers which achieve the same combined rank, but we have arbitrarily chosen this combination among these ones.

7.5 Evaluation of (Soft) Cardinality constraint encodings

We experiment with different (soft) cardinality constraint encodings and their impact on the solution. During our initial experiments we noticed that changing Assign Time Constraints encoding independently from other constraints had significant impact during the search (using a "bad" encoding for ATC leads to noticeably worse solutions) and because of this we chose to give it special treatment in the encoding selection phase. We believe this is because the encoding of this constraint is very important due to the fact that it is a very fundamental one for timetabling and has (arguably) the most impact on other constraints. Therefore, selecting the best encoding for it is crucial.

We denote the encoding configuration used for an instance as a pair X-Y: X is used for Assign Time Constraints, and Y for other constraints. In the case of the combinatorial encoding, the bit adder encoding is used instead only in situations when it would produce too many clauses and variables ($n \geq 50 \vee (n \geq 42 \wedge k \geq 5)$, where n is the number of literals and k is the cardinality of the constraint). The selected encodings are used to encode both hard and soft cardinality constraints. We selected four different encoding configurations (abbreviations: CN - Cardinality Networks, C - Combinatorial, and S - Sequential): S-C, CN-C, S-S, and CN-CN. The last two can simply be abbreviated with simply S and CN, respectively. The first configuration was initially submitted to the maxSAT Competition 2014 and was used in Section 7.4.

We run the best solvers (determined in Section 7.4) with the same time limit of four hours on each instance with each encoding configuration. We consider each pair of solver and encoding configuration as a single solver and rank them for each instance as in Section 7.4. We present the results and rankings in Table 3 and 4.

As in Section 7.4 we wish to select the k pairs of solvers and encoding configurations which complement each other the most. We show the optimum combined rank for every choice of k as a pair (k, rank): (1, 4.23), (2, 2.52), (3, 1.92), (4, 1.6), (5, 1.3), (6, 1.2), (7, 1.12), (8, 1.04), ($n \geq 9$, 1). Based on these results, we chose $k = 4$ as before. One solution for $k = 4$ is Open-WBO (lin) and (def) with (CN, CN, -), Open-WBO (def) with (CN, C, A), and Optiriss (incremental) with (SS, SS, -). These pairs are the best combinations of maxSAT solvers and cardinality constraint encodings for XHSTT according to our criteria.

7.6 Evaluation of a maxSAT approach versus an Integer Programming approach

We compare our maxSAT approach with an existing Integer Programming approach [**?**]. For comparison purposes we used the best k pairs of solvers and encoding configurations determined in Section 7.5. These comparisons are performed as in previous section. The results and rankings are given in Table 5 and 6. We included the comparison with the combined maxSAT solutions as well.

Based on the results, we conclude that maxSAT is competitive with IP. In particular, when comparing the maxSAT configurations individually with IP, two of them (Optiriss(inc)-S and Open-WBO-CN) achieve a better average ranking than IP. When we consider the combined rank, maxSAT outperforms IP in all but five cases. However, we would like to remind the reader that this only the case on the instances we were able to model with maxSAT (see Section 7.1).

An interesting point for maxSAT which we would like to emphasize is that maxSAT solvers are constantly being developed, are in some cases open source (e.g. Open-WBO), and are not so heavily engineered as the commercial IP solver Gurobi in [**?**], but still manage to provide competitive results.

7.7 Evaluation of a pure maxSAT approach versus a SMT approach

We have implemented the SMT approach described in Section 6 by modifying the maxSAT solver openWBO. The implementation was done for a subset of instances. In this section, we give a comparison of results of this SMT approach with a pure maxSAT approach using the same maxSAT solver. We compared with Open-WBO(lin)-S-C as it was the closest maxSAT formulation to our SMT approach described in Section 6. We have run the solvers for four hours and the results obtained are given in Tables 7 and 8. Overall the pure maxSAT approach shows better results, although the average ranks do not differ by a large amount.

From the experiments the SMT approach is significantly outperformed by the pure maxSAT approach. We believe this is because most constraints are important and influence the final solution, in addition to XHSTT problems being challenging to solve optimally.

7.8 Evaluation of maxSAT approach on ITC 2011

In this section we compare with the results obtained during the second phase of the International Timetabling Competition 2011. During this round the time limit was set to 1000 seconds. We run our approach with a normalized amount of time (see Section 7.1 and show the results in Tables 9 and 10.

Our approach provides competitive results with the heuristics solvers used in the competition. Any individual maxSAT configuration would rank second. If we would consider the combined rank of maxSAT solver, then a clear first place would be achieved. However, in the comparison we have only included instances which we were able to model with maxSAT (see Section 7.1), leaving out five instances.

8 Conclusion

In this paper, we have shown that the general High School Timetabling Problem [?] (XHSTT) can indeed be modeled as a weighted partial maxSAT problem, despite the generality of the specification. We presented a complete and detailed mathematical description of the XHSTT problem and its maxSAT modeling in the general sense as required by the specification, but also presented several alternative encodings for special cases. Different solvers and (soft) cardinality constraints were used and evaluated in order to find the most suitable combination for benchmark XHSTT instances. Additionally, a SMT approach has been developed. Our results show that our approach is competitive with Integer Programming, as well as with the heuristic solvers used in ITC 2011. The generated maxSAT instances encode practical and large timetabling problems and have been submitted to the maxSAT Competition 2014. They have been used since then and have been challenging and useful benchmarks for maxSAT solvers.

For future work, there are a number of issues we would like to investigate. Developing portfolio approaches for both cardinality constraint encoding and solver selection could be an interesting research direction, as well as devising more advanced SMT algorithms. Furthermore, it is worth to study if encodings can be still optimized to better suit particular instances.

9 Acknowledgements

The work was supported by the Vienna PhD School of Informatics and the Austrian Science Fund (FWF): P24814-N23. We would also like to thank anonymous reviewers for their helpful comments.

instance/solver	WBO(lin)	WBO(def)	WPM3	Optiriss (inc)	Optiriss (def)	Optiriss (def-linear)	Optiriss (inc-linear)
Italy1	12	12	12	12	12	46	46
Italy4	7809	777	1270	779	779	9899	9899
Kosova	29946	1103	1117	23374	23374	25530	25530
SAwoodlands	251	790	4030	-	-	-	-
SALewitt	349	0	0	39	39	446	446
Brazil1	41	79	70	75	75	41	41
Brazil2	29	30	28	33	33	27	27
Brazil3	39	102	110	105	105	40	40
Brazil4	157	155	154	161	161	171	171
Brazil5	160	125	137	121	121	152	152
Brazil6	248	192	194	196	196	245	245
Brazil7	470	269	254	260	260	531	531
FinArtificial	9	96	4006	267	267	14	14
FinCollege	1056	207	407	186	186	1564	1564
FinElementarySchool	3	3	3	3	3	3	3
FinHighSchool	183	123	131	128	128	268	268
FinSecondarySchool	336	622	781	633	633	411	411
FinSecondarySchool2	2464	179	415	181	181	2742	2742
GreeceAigio	2033	2303	2320	1419	1419	3300	3300
GreeceHighSchool1	0	0	0	0	0	0	0
GreecePatras	888	0	1073	230	230	1241	1241
GreecePreveza	974	1051	1040	187	187	1184	1184
GreeceUni3	89	251	185	308	308	108	108
GreeceUni4	143	344	344	327	327	139	139
GreeceUni5	0	0	0	0	0	0	0

Table 1. Comparison of maxSAT solvers.

instance/solver	WBO(lin)	WBO(def)	WPM3	Optiriss (inc)	Optiriss (def)	Optiriss (def-linear)	Optiriss (inc-linear)
Italy1	1	1	1	1	1	2	2
Italy4	4	1	3	2	2	5	5
Kosova	5	1	2	3	3	4	4
SAwoodlands	1	2	3	4	4	4	4
SALewitt	3	1	1	2	2	4	4
Brazil1	1	4	2	3	3	1	1
Brazil2	3	4	2	5	5	1	1
Brazil3	1	3	5	4	4	2	2
Brazil4	3	2	1	4	4	5	5
Brazil5	5	2	3	1	1	4	4
Brazil6	5	1	2	3	3	4	4
Brazil7	4	3	1	2	2	5	5
FinArtificial	1	3	5	4	4	2	2
FinCollege	4	2	3	1	1	5	5
FinElementarySchool	1	1	1	1	1	1	1
FinHighSchool	4	1	3	2	2	5	5
FinSecondarySchool	1	3	5	4	4	2	2
FinSecondarySchool2	4	1	3	2	2	5	5
GreeceAigio	2	3	4	1	1	5	5
GreeceHighSchool1	1	1	1	1	1	1	1
GreecePatras	3	1	4	2	2	5	5
GreecePreveza	2	4	3	1	1	5	5
GreeceUni3	1	4	3	5	5	2	2
GreeceUni4	2	4	4	3	3	1	1
GreeceUni5	1	1	1	1	1	1	1
average	2.52	2.16	2.64	2.48	2.48	3.24	3.24

Table 2. Ranking of maxSAT solvers.

instance/solver-encoding	α-S	α-CN-C	α-CN	α-S-C	β-S	β-CN-C	β-CN	β-S-C	θ-S	θ-CN-C	θ-CN	θ-S-C	γ-S	γ-CN-C	γ-CN	γ-S-C
Italy1	12	12	12	12	12	12	12	12	12	12	12	12	12	29	12	46
Italy4	11028	7385	14264	7809	535	861	527	777	579	702	538	779	12485	8247	12879	9899
Kosova	26321	29330	29973	29946	1109	1119	1091	1103	1101	1155	1056	23374	27191	32432	32133	25530
SAwoodlands	816	248	265	251	-	0	823	790	-	-	-	-	-	-	-	-
SALewitt	465	213	336	349	61	0	66	0	44	0	55	39	1696	277	334	446
Brazil1	41	41	41	41	59	79	69	79	56	86	52	75	41	41	41	41
Brazil2	38	49	43	29	16	84	75	30	13	34	27	33	34	50	52	27
Brazil3	28	40	49	39	75	117	66	102	68	119	70	105	50	63	63	40
Brazil4	160	182	195	157	140	168	138	155	228	152	143	161	181	190	211	171
Brazil5	166	176	175	160	116	150	122	125	124	136	126	121	183	175	188	152
Brazil6	276	262	279	248	162	184	160	192	138	211	165	196	252	289	295	245
Brazil7	546	605	581	470	254	258	226	269	229	262	231	260	524	591	614	531
FinArtificial	10	16	8	9	8	148	15	96	8	263	19	267	8	14	17	14
FinCollege	644	479	556	1056	233	222	173	207	204	172	182	186	593	587	591	1564
FinElementarySchool	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
FinHighSchool	130	149	132	183	35	40	30	123	31	39	26	128	179	162	149	268
FinSecondarySchool	307	290	256	336	648	657	593	622	622	623	623	633	322	289	281	411
FinSecondarySchool2	971	819	965	2464	195	53	181	179	168	165	180	181	1212	1153	1408	2742
GreeceAigio	2211	2281	2207	2033	2036	2318	767	2303	-	694	-	1419	-	3164	-	3300
GreeceHighSchool1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GreecePatras	879	885	949	888	0	0	125	0	1763	199	-	230	2258	1307	-	1241
GreecePreveza	744	932	1082	974	171	1084	158	1051	2392	143	2320	187	2494	1227	-	1184
GreeceUni3	84	109	97	89	181	220	231	251	193	326	194	308	89	116	100	108
GreeceUni4	146	142	141	143	198	334	218	344	215	295	217	327	149	138	132	139
GreeceUni5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3. Comparison of selected solvers with different cardinality constraints. Abbreviations: α = Open-WBO(lin), β = Open-WBO(def), θ = Optiriss(inc), γ = Optiriss(inc-lin)

instance/solver-encoding	α-S	α-CN-C	α-CN	α-S-C	β-S	β-CN-C	β-CN	β-S-C	θ-S	θ-CN-C	θ-CN	θ-S-C	γ-S	γ-CN-C	γ-CN	γ-S-C
Italy1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	3
Italy4	13	9	16	10	2	8	1	6	4	5	3	7	14	11	15	12
Kosova	10	12	14	13	5	6	2	4	3	7	1	8	11	16	15	9
SAwoodlands	6	2	4	3	8	1	7	5	8	8	8	8	8	8	8	8
SALewitt	13	7	10	11	5	1	6	1	3	1	4	2	14	8	9	12
Brazil1	1	1	1	1	4	7	5	7	3	8	2	6	1	1	1	1
Brazil2	8	10	9	4	2	14	13	5	1	7	3	6	7	11	12	3
Brazil3	1	3	4	2	10	13	7	11	8	14	9	12	5	6	6	3
Brazil4	7	12	14	6	2	9	1	5	16	4	3	8	11	13	15	10
Brazil5	11	13	12	10	1	8	3	5	4	7	6	2	14	12	15	9
Brazil6	13	12	14	10	3	5	2	6	1	8	4	7	11	15	16	9
Brazil7	12	15	13	9	4	5	1	8	2	7	3	6	10	14	16	11
FinArtificial	3	6	1	2	1	10	5	9	1	11	8	12	1	4	7	4
FinCollege	14	9	10	15	8	7	2	6	5	1	3	4	13	11	12	16
FinElementarySchool	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
FinHighSchool	9	11	10	14	4	6	2	7	3	5	1	8	13	12	11	15
FinSecondarySchool	5	4	1	7	13	14	9	10	10	11	11	12	6	3	2	8
FinSecondarySchool2	10	8	9	14	7	1	6	4	3	2	5	6	12	11	13	15
GreeceAigio	7	8	6	4	5	10	2	9	13	1	13	3	13	11	13	12
GreeceHighSchool1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
GreecePatras	5	6	8	7	1	1	2	1	11	3	13	4	12	10	13	9
GreecePreveza	5	6	9	7	3	10	2	8	14	1	13	4	15	12	16	11
GreeceUni3	1	6	3	2	8	11	12	13	9	15	10	14	2	7	4	5
GreeceUni4	7	5	4	6	9	15	12	16	10	13	11	14	8	2	1	3
GreeceUni5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
average	6.60	6.76	7.04	6.44	4.36	6.64	4.23	6.00	5.44	5.72	5.52	6.28	8.20	8.12	8.96	7.64

Table 4. Ranking of selected solvers with different cardinality constraints. Abbreviations: α = Open-WBO(lin), β = Open-WBO(def), θ = Optiriss(inc), γ = Optiriss(inc-lin)

instance/solver	WBO(lin)-CN	WBO(def)-CN-C	WBO-CN	Optiriss(inc)-S	best-maxSAT	IP
Italy1	12	12	12	12	12	15
Italy4	14264	861	527	579	527	8686
Kosova	29973	1119	1091	1101	1091	(2746, 154438)
SALewitt	336	0	66	44	0	0
SAwoodlands	265	0	823	-	0	(390, 343)
Brazil1	41	79	69	56	41	41
Brazil2	43	84	75	13	13	19
Brazil3	49	117	66	68	49	27
Brazil4	195	168	138	228	138	225
Brazil5	175	150	122	124	122	131
Brazil6	279	184	160	138	138	240
Brazil7	581	258	226	229	226	304
FinArtificial	8	148	15	8	8	(25, 71)
FinCollege	556	222	173	204	173	(201, 1394)
FinElementarySchool	3	3	3	3	3	3
FinHighSchool	132	40	30	31	30	155
FinSecondarySchool	256	657	593	622	256	157
FinSecondarySchool2	965	53	181	168	53	2360
GreeceAigio	2207	2318	767	-	767	800
GreeceHighSchool1	0	0	0	0	0	0
GreecePatras	949	0	125	1763	0	0
GreecePreveza	1082	1084	158	2392	158	17
GreeceUni3	97	220	231	193	97	24
GreeceUni4	141	334	218	215	141	25
GreeceUni5	0	0	0	0	0	2

Table 5. Comparison of maxSAT solvers with Integer Programming.

instance/solver	WBO(lin)-CN	WBO(def)-CN-C	WBO-CN	Optiriss(inc)-S	best-maxSAT	IP
Italy1	1	1	1	1	1	2
Italy4	5	3	1	2	1	4
Kosova	4	3	1	2	1	5
SALewitt	4	1	3	2	1	1
SAwoodlands	2	1	3	4	1	5
Brazil1	1	4	3	2	1	1
Brazil2	3	5	4	1	1	2
Brazil3	2	5	3	4	2	1
Brazil4	3	2	1	5	1	4
Brazil5	5	4	1	2	1	3
Brazil6	5	3	2	1	1	4
Brazil7	5	3	1	2	1	4
FinArtificial	1	3	2	1	1	4
FinCollege	4	3	1	2	1	5
FinElementarySchool	1	1	1	1	1	1
FinHighSchool	4	3	1	2	1	5
FinSecondarySchool	2	5	3	4	2	1
FinSecondarySchool2	4	1	3	2	1	5
GreeceAigio	3	4	1	5	1	2
GreeceHighSchool1	1	1	1	1	1	1
GreecePatras	3	1	2	4	1	1
GreecePreveza	3	4	2	5	2	1
GreeceUni3	2	4	5	3	2	1
GreeceUni4	2	5	4	3	2	1
GreeceUni5	1	1	1	1	1	2
average	2.84	2.84	2.04	2.48	1.20	2.64

Table 6. Ranking of maxSAT solvers and Integer Programming.

instance/solver	SMTmaxSAT	WBO(lin)-S-C
Italy1	223	12
Brazil1	69	41
Brazil2	97	29
Brazil3	60	39
Brazil4	146	157
Brazil5	193	160
Brazil6	206	248
Brazil7	511	470
FinCollege	254	1056
FinHighSchool	136	183
FinSecondarySchool	742	336
FinSecondarySchool2	321	2464

Table 7. Comparison of maxSAT and the developed SMT approach (Section 6)

instance/solver	SMTmaxSAT	WBO(lin)-S-C
Italy1	2	1
Brazil1	2	1
Brazil2	2	1
Brazil3	2	1
Brazil4	1	2
Brazil5	2	1
Brazil6	1	2
Brazil7	2	1
FinCollege	1	2
FinHighSchool	1	2
FinSecondarySchool	2	1
FinSecondarySchool2	1	2
average	1.58	1.41

Table 8. Ranking of maxSAT and the developed SMT approach (Section 6)

instance/solver	WBO(lin)-CN	WBO(def)-CN-C	WBO-CN	Optiriss(inc)-S	best-maxSAT	GOAL	HySST	Lectio	HFT
Italy4	1246	21549	22698	558	558	454	6926	651	2636379
Brazil2	96	75	78	88	75	(1, 62)	(1, 77)	38	(6, 190)
Brazil3	114	81	46	86	46	124	118	152	(30, 283)
Brazil4	167	235	-	261	235	(17, 98)	(4, 231)	(2, 199)	(67, 237)
Brazil6	189	427	163	170	163	(4, 227)	(3, 269)	230	(23, 390)
FinElementarySchool	3	3	3	3	3	4	(1, 4)	3	(30, 73)
FinSecondarySchool2	441	1825	406	629	406	1	23	34	(31, 1628)
GreeceAigio	4289	3197	-	1960	1960	13	(2, 470)	1062	(50, 3165)
GreeceUni3	220	112	193	231	112	6	11	(30, 2)	(15, 190)
GreeceUni4	334	155	215	218	155	7	21	(36, 95)	(237, 281)
GreeceUni5	0	22	0	0	0	0	4	(4, 19)	(11, 158)

Table 9. Comparison of maxSAT solvers in the ITC 2011 second round.

instance/solver	WBO(lin)-CN	WBO(def)-CN-C	WBO-CN	Optiriss(inc)-S	best-maxSAT	GOAL	HySST	Lectio	HFT
Italy4	4	6	7	2	2	1	5	3	8
Brazil2	5	2	3	4	2	6	7	1	8
Brazil3	4	2	1	3	1	6	5	7	8
Brazil4	1	2	8	3	2	6	5	4	7
Brazil6	3	5	1	2	1	7	6	4	8
FinElementarySchool	1	1	1	1	1	2	3	1	4
FinSecondarySchool2	5	7	4	6	4	1	2	3	8
GreeceAigio	5	4	7	3	3	1	6	2	8
GreeceUni3	5	3	4	6	3	1	2	8	7
GreeceUni4	6	3	4	5	3	1	2	7	8
GreeceUni5	1	3	1	1	1	1	2	5	4
average	3.63	3.45	3.72	3.27	2.09	3.00	4.09	4.09	7.09

Table 10. Ranking of maxSAT solvers in the ITC 2011 second round.