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), time slots, 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. In addition, timetabling requirements vary from country to country and because of this many variations of HSTT exist. Recently, researchers have proposed a general HSTT problem formulation in an attempt to standardize the problem from different countries and school systems.

In this paper, for the first time we provide a new detailed 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 present experimental results and compare to other approaches, using both artificial and real-world instances, all of which were taken from the Third International Timetabling Competition 2011 (ITC 2011) benchmark repository. Different solvers and (cardinality constraint) encodings are proposed and experimentally evaluated. Our approach gives competitive results and in some cases outperforms the winning algorithm from the timetabling competition.

Keywords: High school timetabling, maxSAT, SAT encodings, cardinality constraints

1 Introduction

In this paper, we describe a modeling of the high school timetabling problem (HSTT) as a maximum propositional satisfiability problem (maxSAT). By doing so, we are able to find solutions for many instances which were proposed by the International Timetabling Competition 2011 [16], and in some cases manage to outperform the winning algorithm of the competition, GOAL.

The problem of timetabling is to coordinate resources (e.g. rooms, teachers, students) with time slots in order to fulfill certain goals or events (e.g. lectures).

Timetabling is encountered in a number of different domains. Every educational institution, airport, public transport system, etc requires some form of timetabling. The difference between a good and a bad timetable can be significant, but constructing timetables by hand can be time consuming, very difficult, error prone, and in some cases impossible. Therefore, developing high quality algorithms which would automatically do so is of great importance. In this work, we focus on HSTT. Respecting constraints is very important for this problem, as timetables directly contribute to the quality of the educational system and satisfaction of students and staff. Every timetabling decision affects hundreds of students and teachers for prolonged amounts of time, since each timetable is usually used for at least a semester.

Unfortunately, HSTT is hard to solve and even simple variants of HSTT are NP-complete problems [11]. Apart from the fact that problems that need to be solved can be very large and have many different constraints, high school timetabling requirements vary from country to country and therefore many variations of the timetabling problem exist. Because of this, it was unclear what the state of the art was regarding solution techniques, as comparing algorithms was difficult.

Recently researchers have proposed a general high school timetabling problem formulation [17] in an attempt to standardize the problem from different countries and school systems and this formulation has been endorsed by the Third International Timetabling Competition 2011 (ITC 2011) [16] [17]. This was a significant contribution, as now algorithms can be compared on standardized instances, that were proposed from different researchers [15].

The winner of the competition was the group GOAL, followed by Lectio and HySST. All of the algorithms were based on heuristics. In GOAL, an initial solution is generated, which is further improved by using Simulated Annealing and Iterated Local Search, using seven different neighborhoods [8]. Lectio uses an Adaptive Large Neighborhood Search [20], while HySST uses a Hyper-Heuristic Search [13]. Recently, Authors [21] used Integer Programming (IP) in a Large Neighborhood Search algorithm, and [19] introduced a two phase IP algorithm for another timetabling problem, but have managed to adjust the method for a number of HSTT instances. Other approaches recently proposed include fix and optimize [9], hybrid approach [22], an IP formulation [23] and ejection chains [14].

All of the best algorithms on the competition were heuristic algorithms and this is why introducing a new complete method (our approach) is important. Some advantages are being able to provide proofs of optimality or infeasibility, calculate lower bounds as well as an opportunity to hybridize algorithms, as well as create valuable benchmarks for maxSAT solvers. Even though significant work has been put into HSTT, optimal solutions for most instances are still not known and this is still an active research area.

In this paper, we investigate the formulation of HSTT as maxSAT. There is a natural connection between timetabling and SAT. HSTT as itself has many logic based characteristics and as such some of its constraints can easily be encoded as SAT. This has motivated us to investigate how efficient can a SAT formulation for HSTT be. However, due to the generality of the specification that we use, devising a complete model is not a trivial task, because as we will see later, some of the constraints are quite complex. In addition to formulating a general formulation, one needs to take care of important special cases which arise in practice and can significantly simplify the encoding.

In [5] 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 HSTT and cannot be modeled in CTT. All of the constraints of CTT with one exception can be modeled as HSTT. Because of this, new and more generalized encodings must be explored in order to model HSTT.

The main contributions of this paper are as follows:

– We show that HSTT can be modeled as Weighted Partial maxSAT problem, despite the fact that HSTT is very general and has many different constraints, both hard and soft versions. All constraints are included in their general formulations, as well as important alternative encodings for special cases.
– We investigate empirically the performance of our model using both artificial and real-world instances, all of which were taken from the Third International Timetabling Competition 2011 benchmark repository. A comparison with the winning algorithm from ITC 2011 is given and the results show that our approach outperforms it in some cases.
– We experiment with different cardinality constraint encodings and compare them to one another, in the context of HSTT. Moreover, we give some insight in what makes HSTT difficult for the maxSAT approach.
– We show how a combination of the maxSAT approach and GOAL can lead to further improvements for larger instances.
– We create maxSAT instances based on HSTT. These instances were submitted to the maxSAT Competition 2014 and have proven to be challenging benchmarks for maxSAT solvers.

Some of the results of this paper have been presented in LaSh 2014 workshop which has no formal proceedings.

The rest of the paper is organized as follows: in the next section, we give a more detailed look into the problem description which serves as an introduction for Section 3, where the detailed presentation of our approach in modeling HSTT as maxSAT is given. In Section 4, we provide computational results obtained on artificial and real life problems, which includes comparisons of different encodings and solvers. Finally, we give concluding remarks and ideas for future work.

2 Problem Description

In our research we consider the general formulation of the High School Timetabling problem, as described in [17].

The general High School Timetabling formulation specifies three main entities: times, resources, and events. Times refer to discrete time slots which are available, such as Monday 9:00-10:00, Monday 10:00-11:00, etc. 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 and a specific student group (both considered resources) and two time slots. Events can be divided into subevents which is done in cases where longer events need to be split into multiple shorter ones (e.g. a four hour Mathematics lesson can be split into several subevents of duration one, two or three).

Constraints impose limits on what kind of assignments are feasible or desired. These may constraint 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. We describe the constraints in more details in the next section when we present the SAT formulations.

Each constraint has a nonnegative cost function associated with it, which penalizes assignments that violate it. It is important to differentiate between hard and soft constraints. Hard constraints are constraints that define the feasibility of the solution and are required for the solution to make sense, while soft constraints define desirable situations, which define the quality of the solution. Therefore, the cost function consists of two parts: infeasibility value (hard constraints) and objective value (soft constraints) part. The final cost function infeasibility and objective value parts are calculated as the sum of the cost functions of hard and soft constraints, respectively. The goal is to first minimize the infeasibility and then minimize the objective function value part. If the unfeasible value is equal to zero, we say that the solution is feasible, otherwise it is unfeasible. The exact way these two are calculated will be discussed in the next section.

3 Our Approach - Modeling HSTT as maxSAT

3.1 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 $(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$ unfeasible and $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$ feasible.

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 cardinaltiy 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 [18], [6]), each typically requiring a different amount of auxiliary variables and clauses. In Section 3.3 we describe the ones used in our implementation in more detail.

3.2 Constraints

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.

Each HSTT constraint has a so called "points of application" and each point generates a number of deviations. Cost of the constraint is obtained by applying a cost function on the set of deviations produced and multiplying it by a weight (e.g. a cost function may simply be the sum of all deviations). For example, a constraint that specifies that a lesson must be spread over the course of three days (in the specification this constraint is called Spread Events Constraint), a point of application would be each lesson specified and a deviation for a lesson would be calculated as $max(0, 3 - S_{lesson})$ where S_{lesson} is the number of days in which the lesson is currently scheduled to take place (e.g. if the complete lesson is being hold only on Monday, the deviation is two).

Our approach supports cost functions of sums of (squares of) deviations. The HSTT specification allows for other cost functions as well, but we do not have an encoding for them currently. Fortunately, only two instances use nonsupported cost functions.

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 HSTT as hard clauses and we do not distinguish between two different unfeasible solution in terms of quality. By doing so we simplify the computation, possibly offering a faster algorithm.

E, T, and R are sets of events, times, and resources, respectively. Each constraint applies to a subset of them and will be denoted by E_{spec}, T_{spec}, and 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.

Assign Time Constraints Every event must be assigned a given amount of times. For example, if a lecture lasts for two hours, two time slots must be assigned to it.

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:

$$\bigwedge_{\forall e \in E} (exactly\_d[Y_{e,t} : t \in T]) \tag{1}$$

Avoid Clashes Constraint 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.

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:

$$\bigwedge_{\substack{\forall e \in E \\ t \in T \\ r \in R}} (Y_{e,t,r} \Rightarrow Y_{e,t}) \tag{2}$$

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

$$\bigwedge_{\substack{\forall r \in R_{spec} \\ t \in T}} (at\_Most\_1[Y_{e,t,r} : e \in E(r)]) \tag{3}$$

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

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:

$$\bigwedge_{\substack{\forall r \in R \\ t \in T}} (X_{t,r} \Leftrightarrow \bigvee_{e \in E(r)} Y_{e,t,r}) \tag{4}$$

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

$$\bigwedge_{\forall r \in R_{spec}} (atMost\_0[X_{t,r} : t \in T_{spec}]) \tag{5}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead. Points of application are resources and deviations are calculated as the number of times a resource is assigned to an unavailable time.

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 time slots 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.

In the formal specification of HSTT, 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 slot of Monday* and *first time slot 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 time slots 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 $(t-1)$:

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ t \in T}} (Y_{e,t} \wedge \overline{Y}_{e,(t-1)} \Rightarrow S_{e,t}) \tag{6}$$

If a starting time for event e has been assigned at time t, then the corresponding event must also take place at that time:

$$\bigwedge_{\substack{\forall e \in E \\ t \in T}} (S_{e,t} \Rightarrow Y_{e,t}) \tag{7}$$

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

$$\bigwedge_{\forall e \in E_{spec}} (atLeast\_A_{min}[S_{e,t} : t \in T] \wedge atMost\_A_{max}[S_{e,t} : t \in T]) \tag{8}$$

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[1]:

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T}} (S_{e,t} \Rightarrow \bigvee_{d_{min} \leq d \leq d_{max}} K_{e,t,d}) \tag{9}$$

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 $[d_{min}, d_{max}]$):

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T \\ d \in D}} K_{e,t,d} \Rightarrow \bigwedge_{i \in [0,d-1]} Y_{e,(t+i)} \tag{10}$$

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

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T \\ d \in D}} (K_{e,t,d} \Rightarrow \bigwedge_{d_{min} \leq g \leq d_{max}} \bigwedge_{i \in [0,d-1] \wedge (i \neq 0 \vee g \neq d)} \overline{K}_{e,t+i,g}) \tag{11}$$

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:

$$K_{e,t,d} \wedge Y_{e,t+d} \Rightarrow S_{e,t+d} \tag{12}$$

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 slot on Monday.

The constraint is encoded as:

$$\bigwedge_{\forall e \in E_{spec}} (atMost\_0[\star : t \in T \setminus T_{spec}]) \tag{13}$$

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

[1] 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.

Distribute Split Events Constraint 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:

There must be at least d_{min} starting times with given duration d:

$$\bigwedge_{\forall e \in E_{spec}} (atLeast\_d_{min}[K_{e,t,d} : \forall t \in T]) \tag{14}$$

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

$$\bigwedge_{\forall e \in E_{spec}} (atMost\_d_{max}[K_{e,t,d} : \forall t \in T]) \tag{15}$$

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

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. 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 slot t. Formally,

$$\bigwedge_{\substack{eg \in EG_{spec} \\ t \in T}} (Z_{eg,t} \Leftrightarrow \bigvee_{e \in eg} S_{e,t}) \tag{16}$$

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. 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):

$$\bigwedge_{\substack{\forall tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atLeast\_d_i^{min}[Z_{eg,t} : t \in tg_i]) \tag{17}$$

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

$$\bigwedge_{\substack{\forall tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atMost\_d_i^{max}[Z_{eg,t} : t \in tg_i]) \tag{18}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead. Points of application are event groups and deviations are calculated as the number by which the events group falls short of the minumum or exceeds the maximum.

Limit Busy Times Constraints This constraints imposes limits on the number of times a resource can become busy within certain a 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. Again, we first encode auxiliary variables:

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

$$\bigwedge_{\substack{\forall r \in R \\ \forall tg \in TG_{spec}}} (B_{tg,r} \Leftrightarrow \bigvee_{t \in T} X_{t,r}) \tag{19}$$

The constraint is now encoded as:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R_{spec}}} (B_{tg,r} \Rightarrow atLeast\_b_{min}[X_{t,r} : t \in tg] \wedge atMost\_b_{max}[X_{t,r} : t \in tg]) \tag{20}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead. Points of application are resources and each resource generates multiple deviations (one for each time group) which calculated as the number by which the events group falls short of the minimum or exceeds the maximum.

Limit Idle Times Constraints This constraint specifies the minimal 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 must not have any idle times. To encode the constraint, a three different type 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 slot in time group tg. It is important to emphasize that these variables are created with respect to time groups and that the index for time denotes the number of a time slot within a group times. Formally:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (G^{tg}_{t,r} \Leftrightarrow \bigvee_{i\ before\ t\ \wedge\ i \in tg} X_{i,r}) \tag{21}$$

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (H^{tg}_{t,r} \Leftrightarrow \bigvee_{i\ after\ t\ \wedge\ i \in tg} X_{i,r}) \tag{22}$$

The first and last time slot 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.

Variable $I^{tg}_{t,r}$ indicates that a resource is idle at time t with respect to time group tg (set of times) if 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,

$$\bigwedge_{\substack{tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (I^{tg}_{t,r} \Leftrightarrow \overline{X}_{t,r} \wedge G^{tg}_{t,r} \wedge H^{tg}_{t,r}) \tag{23}$$

We now encode the constraint:

There must be at least $idle_{min}$ idle times during a time group:

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R}} (atLeast\_idle_{min}[I^{tg}_{t,r} : t \in tg]) \tag{24}$$

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

$$\bigwedge_{\substack{\forall tg \in TG_{spec} \\ r \in R}} (atMost\_idle_{max}[I^{tg}_{t,r} : t \in tg]) \tag{25}$$

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

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.

There must be at least b_{tg}^{min} busy time groups:

$$\bigwedge_{\forall r \in R_{spec}} (atLeast\_b_{tg}^{min}[B_{tg,r} : tg \in TG_{spec}]) \tag{26}$$

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

$$\bigwedge_{\forall r \in R_{spec}} (atMost\_b_{max}^{tg}[B_{tg,r} : t \in TG_{spec}]) \tag{27}$$

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

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 maximal separations between the two events. The constraint specifies a set of pairs of events to which it applies.

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:

$$\bigwedge_{\substack{\forall (e_1,e_2) \in E_{spec}^2 \\ \forall t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{i \in [0,t]} \overline{Y}_{e_2,i}) \tag{28}$$

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} time slots:

$$\bigwedge_{\substack{\forall (e_1,e_2) \in E_{spec}^2 \\ \forall t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{j \in [t+1, t+B_{min}]} \overline{Y}_{e_2,j}) \tag{29}$$

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

$$\bigwedge_{\substack{\forall (e_1,e_2) \in E_{spec}^2 \\ \forall t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{k \in [t, t+B_{max}+1]} Y_{e_2,k}) \tag{30}$$

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 together.

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:

$$\bigwedge_{\substack{\forall eg \in EG_{spec} \\ t \in T \\ e_j, e_k \in eg}} (Y_{e_j,t} \Leftrightarrow Y_{e_k,t}) \tag{31}$$

If the constraint is declared a soft one, we may apply a similiar 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. Points of application are event groups and deviations are calculated as the number of times in which at least one of the events within the event group is taking place, but not all of them.

Preassign Resource Constraints Specified events must use specified predefined resources. For example, a math lecture for class A requires Alice as a math teacher in order to take place.

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:

$$\bigwedge_{\substack{\forall e \in E \\ \forall t \in T \\ \forall r \in R}} (Y_{e,t,r} \Rightarrow Y_{e,t}) \wedge \bigwedge_{\substack{\forall e \in E_{spec} \\ t \in T \\ r \in R^e_{spec}}} (Y_{e,t} \Rightarrow Y_{e,t,r}) \tag{32}$$

In the actually specification of the general HSTT, this constraint is given when events are defined, rather than a seperate constraint.

Preassign Time Constraints Similar to Preassign Resource Constraints, certain events have a fixed schedule. For example, an external professor is available only on 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, a resource 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:

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T}} (Y_{e,t} \rightarrow exactly\_1[M^{role}_{e,t,r} : r \in R_{spec\_resource\_type}]) \tag{33}$$

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:

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T \\ r \in R_{spec\_resource\_type}}} (M^{role}_{e,t,r} \rightarrow Y_{e,t,r}) \tag{34}$$

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:

A resource may fulfill at most one role at any given time:

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T \\ r \in R}} (atMost\_1[M^{role}_{e,t,r} : role \in ARC_{roles}]) \tag{35}$$

Avoid Split Assignments Constraint Events are frequently broken down into subevents, each of which has the same resource requirements. This constraint imposes that for the specified *role*, only one resource should be used across all of the subevents. For example, a lecture should always take place in the same lecture room, regardless of which room is chosen. 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:

$$\bigwedge_{\substack{e \in E_{spec} \\ R_{spec\_resource\_type} \\ t \in T}} (M_{e,t,r}^{role} \rightarrow V_{e,r}^{role}) \tag{36}$$

The constraint is now encoded as:

$$\bigwedge_{e \in E_{spec}} (atMost\_1[V_{e,r}^{role} : r \in R_{spec\_resource\_type}]) \tag{37}$$

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

Prefer Resources Constraints The Assign Resources Constraint specifies that for each required resource type, a resource must be assigned. This constraint imposes further constraints on which resource should be assigned. For example, Assign Resource Constraint might require a room to be specified to a lecture, but Prefer Resource Constraint states that seminar rooms are more preferable. Similar to before, this constraint applies for a specified *role*. The encoding relies on auxiliary variables create in ARC:

$$\bigwedge_{\substack{\forall e \in E_{spec} \\ \forall t \in T}} (atLeast\_1[M_{e,t,r}^{role} : r \in R \setminus R_{spec}]) \tag{38}$$

Limit Workload Constraints Specified resource cannot exceed a specified workload amount. For example, teachers may work no more than 20 hours per week. 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:

$$Workload(sr) = Dur(subevent) * Workload(subevent)/Dur(event) \tag{39}$$

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 substract 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:

$$\bigwedge_{r \in R_{spec}} (atLeast\_Work_{min}[X_{t,r} : t \in T] \land atLeast\_Work_{max}[X_{t,r} : t \in T]) \tag{40}$$

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:

$$\bigwedge_{t \in T} (atMost\_h[Y_{e,t} : e \in E_{spec}]) \tag{41}$$

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 FinlandArtificialSchool instance and without this case and another case described further it would not have been possible to encode within reasonable amounts of memory.

If there is only one *role* per resource type specified in the requirements of an event, then the encoding of the auxiliary variables in ARC 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 un(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 HSTT. In some cases, it suffices to encode $K_{e,t,d}$ as (i is an integer):

$$K_{e,t,d} \leftrightarrow (\bigwedge_{i \in [0..d-1]} Y_{e,t+i}) \wedge \overline{Y}_{e,t+d} \tag{42}$$

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.

3.3 Cardinality Constraints - details

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 [18] 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 [18] is closely related to unary numbers. The unary number representation for an integer n as given in [7] is:

$$\bigwedge_{\forall i \in [1, n-1]} (u_i \Rightarrow u_{i-1}). \tag{43}$$

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 [18]. 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 to the way it is 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 unfeasible. 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:

$$\bigwedge_{\substack{\forall i \in [0..n) \\ \forall j \in [0..,kmax) \\ (i+1 \geq k_{max} \vee j \leq i)}} (S_{i,j} \Rightarrow S_{i+1,j}) \tag{44}$$

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:

$$\bigwedge_{\substack{\forall i \in [1..n) \\ \forall j \in [1..,kmax) \\ (i+1 \geq k_{max} \vee j \leq i)}} (x_i \wedge S_{i-1,j-1} \Rightarrow S_{1,j}) \tag{45}$$

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):

$$\bigwedge_{\forall i \in [0..n)} (x_i \Rightarrow S_{i,0}) \tag{46}$$

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

$$\bigwedge_{\substack{\forall i \in [0..n) \\ \forall j \in [0..,kmax-1) \\ (i+1 \geq k_{max} \vee j \leq i)}} (\overline{S_{i,j}} \Rightarrow \overline{S_{i+1,j+1}}) \tag{47}$$

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:

$$\bigwedge_{\forall j \in [0..,kmax)} (\overline{S_{i,j}} \Rightarrow x_i) \tag{48}$$

General cases:

$$\bigwedge_{\substack{\forall i \in [0..n-1) \\ \forall j \in [0..,kmax) \\ (i+1 \geq k_{max} \vee j \leq i)}} (\overline{S_{i,j}} \wedge S_{i+1,j} \Rightarrow x_{i+1}) \tag{49}$$

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 48, 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 47. 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:

$$(S_{n-1,k_{min}-1}) \tag{50}$$

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

$$\bigwedge_{\forall i \in [k_{max}-1..n)} (\overline{x_i} \vee \overline{S_{(i-1),(k_{max}-1)}}) \tag{51}$$

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 51.

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

Cardinality Networks Cardinality Networks are described in [6]. 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 [6]. The number of auxiliary variables and clauses required for this encoding is $O(nlog^2k)$.

3.4 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:

$$\bigwedge_{j \in P} (A_j \rightarrow atLeast\_j[x_i : x_i \in X]) \wedge \bigwedge_{j \in P} (w(j)(A_j)) \tag{52}$$

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 [18] 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 $atMost\_k_{max}$ (50 and 51). 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 $atMost\_k_{max}$):

$$\bigwedge_{\forall i \in [1..k_{min})} (w_i(S_{(n-1),(i-1)})) \tag{53}$$

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:

$$(w_0(S_{(n-1),0})) \tag{54}$$

$$\bigwedge_{\forall i \in [1..k_{min}]} (w_i(\overline{S_{(n-1),(i-1)}} \vee S_{(n-1),i})) \tag{55}$$

Cardinality Networks Cardinality Networks [6] can be used to model soft constraints and this has been done in [5]. However, when doing so, since every output must be penalized[2], they require more auxiliary variables and degenerate into Sorting Networks [10] from which they offer improvements, meaning the number of auxiliary variables and clauses goes up to $O(nlogn)$.

3.5 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 $w(j) = c * j$,

[2] 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.

where c is some constant. In this case, instead of using any of the previously described encodings, we encode the following soft unit clauses:

$$\bigwedge_{x_i \in X} ((c)(x_i)) \tag{56}$$

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.

4 Computational Results

4.1 Benchmark instances

We have conducted experimental evaluations on benchmark instances from HSTT which can be found on the repository of the ITC 2011 [2]. Instances which were suggested by the competition as test beds, as well as the ones used in the competition have been chosen (these two sets intersect).

In the instances, the number of time slots ranges from 25 to 60, number of resources from 20 to 120, number of events from 200 to 1000 with total event duration from 300 to 1500. These numbers vary heavily from instance to instance. We do not provide detailed information regarding these instances, but direct the interested reader to [2] [15] [17].

We made our generated maxSAT instances available online (`www.dbai.tuwien.ac.at/user/demir/xHSTTtoSAT_instances.tar.gz`). These instances have been submitted and used in the maxSAT Competition 2014.

4.2 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 unfeasible value is equal to zero, we say that the solution is feasible, otherwise it is unfeasible.

4.3 Solvers

We experimented with different maxSAT solvers in order to solve encodings obtained by our approach. We considered the following maxSAT solvers, all of which showed good performances in the maxSAT Competition 2013 [1] in the Industrial Weighted Partial Max-SAT category (either in the complete or incomplete track): sat4j-maxsat, optimax, WPM2*[3], WPM1, Munsat, ISAC [12]. The solvers can be downloaded from [4].

4.4 Evaluation of (Soft) Cardinality constraint encodings

We experimented with different (soft) cardinality constraint encodings and their impact in the solution. We denote the encodings used for an instance as a triplet of encodings (X, Y, Z): X is used for AssignTimeConstraints, Y for other constraints and Z is used as a backup encoding only in situations were the combinatorial encoding (if present) would produce too large encodings. The selected encodings are used to encoding both hard and soft cardinality constraints.

During our initial experiments, we noticed that changing AssignTimeConstraints 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 is a very fundamental one for timetabling and has (arguably) the most impact on other constraints and therefore selecting the best encoding for it is crucial.

We experimented with the following encodings: [4]:

– (CN, C, A)
– (S, C, A)
– (S, C, CN)
– (S, S, -)
– (CN, CN, -)
– (S, A, -)
– (S, CN, A)

4.5 Experimental plan

We divided the experimental plan into three phases, because testing all combinations of solvers, instances, and encodings for extended periods of time is too computationally expensive. In each phase experiments will be done and based on the results the next phase will continue.

First Phase In the first phase, we chose the only two solvers from the maxSAT Competition 2013 incomplete track (sat4j-maxsat and optimax), ten instances, all encodings described previously, and allocated two hours for each instance. Instance have different sizes (from small to large) and come from different countries. We ran the experiments on the same machine with eight cores, each instance being given a single core and four instances were solved at a time. The results of the first phase are given in Tables 1 and 2 (values indicate the cost of the violation of soft constraints, with the exception of the value $'-1'$ which means that the solver did not produce any solution within the given timelimit):

For the next phase, based on the results, we selected (S, C, A) and (S, S, A). The reasons for this are given below.

[3] We use the asterisk to denote that WPM2 solver we use has been modified to output the best solution found during a given time frame. We thank Carlos Ansótegui for providing this version.

[4] Abbreviations: CN - Cardinality Networks, C - Combinatorial, A - Bit Adder, S - Sequential.

Sat4j 2h	(CN, C, A)	(S, C, A/CN)	(S, S, -)	(CN, CN, -)	(S, A, -)	(S, CN, -)
Brazil1	48	48	48	47	**42**	45
Brazil2	86	**60**	66	94	80	76
Brazil6	341	**247**	330	380	339	357
FinArtificial	257	59 (252)	30	63	-1	**22**
FinHighSchool	1079	845	**489**	617	1585	537
GreecePatras2010	1487	**826**	1619	2149	10745	2027
GreeceUni4	191	172	148	**122**	507	152
Italy1	73	54	77	**12**	231	-1
Italy4	20230	**18139**	20327	21109	-1	21894
Kosova1	92775	96011	20436	13572	-1	**11279**

Table 1. First phase results for sat4j-maxsat

Optimax 2h	(CN, C, -)	(S, C, A/CN)	(S, S, -)	(CN, CN, -)	(S, A, -)	(S, CN, -)
Brazil1	-1	**41**	46	-1	-1	44
Brazil2	39	36	**27**	-1	34	-1
Brazil6	-1	-1	-1	-1	-1	-1
FinArtificial	-1	-1	-1	-1	-1	-1
FinHighSchool	-1	-1	-1	-1	-1	-1
GreecePatras2010	-1	**0**	-1	-1	-1	-1
GreeceUni4	-1	-1	-1	-1	-1	-1
Italy1	13	**12**	**12**	**12**	14	-1
Italy4	-1	-1	-1	-1	-1	-1
Kosova1	-1	-1	-1	-1	-1	-1

Table 2. First phase results for optimax

Since the results obtained for Italy4 and KosovaInstance are far from competitive for all of the encodings (best solution known to this date are 40 and 3, respectively), we opted to discard them from our decision process for the next phase. (CN, C, A) is worse in each case when compared to (S, C, A) and is therefore discarded. A similar argument holds for (S, C, CN) and (S, A, -), even though the latter does better in one case. In addition, since optimax could not provide solutions for most of the instances, we temporarily exclude it as well for our decision process.

For the remaining encodings, we assigned a score for each result, which corresponds to its relative performance with respect to other encodings. Smaller scores indicated better performance, with 1 being the best possible score. For cases when the results are close we treat them as equal (e.g. a result of 48 and 47 are treated equal), because we believe the difference is not significant enough to justify a clear winner, unless this behavior is consistent. The obtained scores are given in Table 3.

Based on the average score, we chose to proceed with (S, C, A) and (S, S, A).

Second phase For the second phase, we use the same ten instances as before, but this time use the two best encodings from the previous step, use complete and incomplete solvers mentioned in

-	(S, C, A/CN)	(S, S, -)	(CN, CN, -)	(S, CN, -)
Brazil1	2	2	2	1
Brazil2	1	2	4	3
Brazil6	1	2	4	3
FinArtificial	3 (4)	2	3	1
FinHighSchool	4	1	3	2
GreecePatras2010	1	2	3	4
GreeceUni4	3	2	1	2
Italy1	2	3	1	4
Sum	17 (18)	16	21	20

Table 3. Rankings of solvers based on solutions

Section 4.3, and extend the computational time to four hours. The used the same computer as in the previous stage. Tables 4, 5 and 6 summarize the results:

Sat4j 4h	(S, C, A/CN)	(S, S, -)
Brazil1	**46**	48
Brazil2	**55**	58
Brazil6	**247**	326
FinArtificial	53	**13**
FinHighSchool	782	**438**
GreecePatras2010	**826**	1619
GreeceUni4	169	**148**
Italy1	**34**	42
Italy4	**18139**	20327
Kosova1	96011	**20399**

Table 4. Second phase results for sat4j

WPM1, Munsat and ISAC are not included in the tables above because they could not provide solutions within the allocated time limit for the instances (with one exception: Italy1 where the solution given was 12, optimal).

In this phase, we wish to choose the best solver along with the best encoding. Based on the results, we selected sat4j-maxsat as the best solver. Below we discuss our decisions in more detail.

For the same reasons as in the previous phase, we disregard the results obtained for Italy4 and KosovaInstance.

Optimax managed to provide very good solutions for some instances, but most instances could not be solved. A similar situation occurs with WPM2*.

An advantage for sat4j-maxsat is that it always found a feasible solution, which is important for practical reasons. Another interesting property of the solver is that improvements are made in a number of smalls steps and the solver can be stopped at any time and provide a solution, whereas

Optimax 4h	(S, C, A/CN)	(S, S, -)
Brazil1	**41**	46
Brazil2	36	**27**
Brazil6	-1	-1
FinArtificial	-1	-1
FinHighSchool	-1	-1
GreecePatras2010	**0**	**0**
GreeceUni4	-1	-1
Italy1	**12**	**12**
Italy4	-1	-1
Kosova1	-1	-1

Table 5. Second phase results for optimax

WPM2 4h	(S, C, A/CN)	(S, S, -)
Brazil1	**41**	43
Brazil2	78	**33**
Brazil6	688	**679**
FinArtificial	**4006**	-1
FinHighSchool	103	**100**
GreecePatras2010	**244**	-1
GreeceUni4	-1	-1
Italy1	**12**	460
Italy4	**126537**	-1
Kosova1	-1	-1

Table 6. Second phase results for WPM2

in the other two solvers, improvements are done in large steps and stopping the solver right before the next improvement may be very impactful for the solution. The downside is that in the limited time provided, for some instances the other two solvers provide much better solutions.

Since both encodings performed comparably good with respect to each other in the case of sat4j-maxsat, we opted to keep both of them for the final phase.

Third phase In the final phase, we test the performance of sat4j-maxsat on all instances, using (S, S, A) and (S, C, A) encodings for 24 hours. All tests were performed on an Intel Core i3-2120 CPU @ 3.30GHz with 4 GB RAM, each instance being given a single core and four instances were solved at a time. Results are given in Table 7:

Overall, the (S, C, A) encoding performs better, even though it got outperformed in a few examples and therefore $(S, S, -)$ should not be excluded from future instances.

In the following section, we discuss the results in greater detail.

Name	(S, C, A)	(S, S, -)
BrazilianInstance1	**(0, 38)***	(0, 40)
BrazilianInstance2	(0, 32)	**(0, 22)**
BrazilianInstance4	**(0, 205)**	(0, 214)
BrazilianInstance5	**(0, 117)**	(0, 169)
BrazilianInstance6	**(0, 230)**	(0, 267)
BrazilianInstance7	**(0, 400)**	(0, 481)
SouthAfricaLewitt2009	(0, 26)	**(0, 12)**
SouthAfricaWoodlands	**(0, 0)***	**(0, 0)***
FinlandCollege	(0, 1523)	**(0, 900)**
FinlandHighSchool	**(0, 289)**	(0, 387)
FinlandSecondarySchool	**(0, 252)**	(0, 358)
FinlandArtificialSchool	(0, 47)	**(0, 3)**
GreecePatras2010	**(0, 331)**	(0, 1587)
GreeceWesternUniversity4	**(0, 121)**	(0, 148)
GreeceHighSchool	**(0, 0)***	**(0, 0)***
KosovaInstance	(0, x)	(0, x)
EnglandStPaul	(0, x)	(0, x)
ItalyInstance1	(0, 17)	**(0, 12)***
ItalyInstance4	(0, 12825)	(0, 20007)

Table 7. Results obtained with sat4j-maxsat after 24 hours.

4.6 Discussion of results and additional comparisons

So far, we compared different solvers and encodings to one another. In this section, we give a comparison with GOAL [8], the winning algorithm of the International Timetabling Competition 2011 (ITC 2011).

GOAL is a stochastic local search algorithm. It starts for a (possibly unfeasible) initial solution generated by KHE [3] and iteratively performs local improvements, using a number of different neighborhoods. We ran GOAL with its default parameters on the same machine used in the third phase with the same 24 hour time budget.

The results obtained are summarized in Table 8. Values marked with an asterisk * are known to be optimal (further information on how the optimality proof was obtained can be found on ITC's web page on individual instances [2]):

There might be differences in the results obtained by GOAL in the competition and obtained by our 24 hour runs, because in the competition competitors in the final phase were given one month to use whatever available resources to provide the best results. We focus here on the comparison with the winner of ITC 2011 competition, because we think that this gives an idea of how good our approach performs in a limited amount of time compared to one of best existing approaches for this problem. For some of the instances, better upper bounds were obtained after the competition by GOAL and other approaches without time or resource limitations.

Our approach outperformed GOAL in 12 instances. Three of the instance had been solved to optimality, namely BrazilianInstance1 and South African instances. For the others, part to the success on the Brazilian instances can be attributed to the fact that our approach quickly finds a

Name	(S, C, A)	(S, S, -)	GOAL
BrazilianInstance1	**(0, 38)***	(0, 40)	(0, 54)
BrazilianInstance2	(0, 32)	**(0, 22)**	(1, 42)
BrazilianInstance4	**(0, 205)**	(0, 214)	(16, 95)
BrazilianInstance5	**(0, 117)**	(0, 169)	(4, 121)
BrazilianInstance6	**(0, 230)**	(0, 267)	(4, 195)
BrazilianInstance7	**(0, 400)**	(0, 481)	(11, 230)
SouthAfricaLewitt2009	**(0, 0)***	(0, 12)	(0, 18)
SouthAfricaWoodlands	**(0, 0)***	**(0, 0)***	(2, 13)
FinlandCollege	(0, 1523)	**(0, 900)**	(1, 5)
FinlandHighSchool	(0, 289)	(0, 378)	**(0, 14)**
FinlandSecondarySchool	(0, 252)	(0, 358)	**(0, 83)**
FinlandArtificialSchool	(0, 47)	**(0, 3)**	(3, 6)
GreecePatras2010	(0, 331)	(0, 1587)	**(0, 0)***
GreeceWesternUniversity4	(0, 121)	(0, 148)	**(0, 5)**
GreeceHighSchool	**(0, 0)***	**(0, 0)***	**(0, 0)***
KosovaInstance	(0, x)	(0, x)	**(0, 5)**
EnglandStPaul	**(0, x)**	**(0, x)**	(3, 48)
ItalyInstance1	(0, 17)	**(0, 12)**	(0, 19)
ItalyInstance4	(0, 12825)	(0, 20007)	**(0, 57)**

Table 8. Comparison of maxSAT approach with GOAL.

feasible solution, while GOAL in these cases does not and its heuristics could not escape infeasibility. Even though our encoding cannot capture the square of sums cost function for soft constraints of EnglandStPaul and KosovaInstance, it does not affect our findings of feasible solutions, which gives immediately a better solution than the unfeasible solution for EnglandStPaul of GOAL. For these instances we did not provide the objective function obtained, since only hard constraints had been considered and the resulting objective value is essentially random. GOAL outperformed our method in 6 instances.

In a previous iteration of our approach, we used a less general encoding which was still sufficient for ItalyInstance1 (see the notes in the previous section regarding $S_{e,t}$ and $K_{e,t,d}$) and have calculated the optimal solution of the instance (objective value 12) in around 10 hours. However, this is not automatized and must be done by hand currently, but it is an interesting way to point out that the more general the encoding is, the harder it is to solve, since the layers of abstraction add additional complexity. With the general encoding, it took around 50 hours to obtain the optimal solution in contrast to the previous 10 hours.

We took a closer look into the instances and solution obtained for instances which were not solved well compared to GOAL. It appears that Limit Idle Times constraint is difficult for maxSAT, as in these cases this constraint was dominating the cost.

Overall, we conclude that our approach is competitive. Our approach can be used to provide competitive solutions when compared to GOAL and in the case of smaller instances optimal solution can be calculated. The results obtained in cases where encodings could be constructed are promising and we believe that further research in this direction will prove to be fruitful.

In the next subsection we discuss instances which have been used in ITC 2011, but have not been encoded by our approach.

Instances not encoded Assigning resources to events in the general case is problematic for our maxsat approach. The encodings become very large because of the way we implemented the general encoding and the search space drastically increases as after assigning a timeslot and duration to a subevent the solver needs to assign a resource as well. For some special cases these constraints can be encoded as discussed in 3.5. Due to previously mentioned reasons, Spanish, Australian, Dane, Netherlands instances were not encoded.

KosovaInstance and EnglandStPaul contain the square of the sum of deviations cost function for some of its soft constraints, which is currently not supported, as discussed in the previous section. However, this does not affect the generation of a feasible solution, which was done successfully in both cases.

4.7 Combination of approaches

In the selected instances our approach quickly (with a few seconds, with one exception being the FinlandArtificial instance were it takes half an hour) finds an initial feasible solution which is then gradually improved, while for some cases GOAL does not manage to find an feasible solution after its search. This has motivated us to investigate how well would GOAL perform if we would use our method to generate a feasible initial solution and leave the optimization to GOAL. We have implemented this approach and Table 9 summarizes the results, where the column *Combination* displays the results obtained. Instances in which the difference was marginal to previously obtained results were excluded for clarity:

Name	(S, C, A)	(S, S, -)	GOAL	Combination
BrazilianInstance1	**(0, 38)***	(0, 40)	(0, 54)	(0, 48)
BrazilianInstance2	(0, 32)	**(0, 22)**	(1, 42)	(0, 37)
BrazilianInstance4	(0, 205)	(0, 214)	(16, 95)	**(0, 142)**
BrazilianInstance5	(0, 117)	(0, 169)	(4, 121)	**(0, 106)**
BrazilianInstance6	(0, 230)	(0, 267)	(4, 195)	**(0, 171)**
BrazilianInstance7	(0, 400)	(0, 481)	(11, 230)	**(0, 210)**
SouthAfricaLewitt2009	**(0, 0)***	(0, 12)	(0, 18)	*(0, 470)*
SouthAfricaWoodlands	**(0, 0)***	**(0, 0)***	(2, 13)	(0, 71)
FinlandCollege	(0, 1523)	(0, 900)	(1, 5)	**(0, 14)**
FinlandArtificialSchool	(0, 47)	**(0, 3)**	(3, 6)	(0, 12)
KosovaInstance	(0, x)	(0, x)	**(0, 5)**	*(0, 1059)*
EnglandStPaul	(0, x)	(0, x)	(3, 48)	**(0, 138)**

Table 9. Comparison of results with the combination of GOAL and maxSAT approach

In cases where GOAL previously could not find a feasible solution, the combination approach was significantly more successful than GOAL. However, for two cases, namely SouthAfricaLewitt2009 and KosovaInstance, GOAL quickly got stuck in a local optimal with the initial solution provided. Overall, the combination approach seems to be useful, especially for larger instances.

4.8 Further analysis on cardinality constraints

Resulting maxSAT instances depend on the way we choose encodings for the cardinality constraints. We now take a look into the number of variables and clauses for each selected instances and cardinality encoding used. Table 10 shows this information.

Encoding/Instance	(S, S, -)	(S, C, A)	(S, CN, -)	(S, A, -)	(CN, C, A)	(CN, CN, -)
BrazilInstance1	(7614; 24989)	(5537; 21026)	(15929; 48048)	(14668; 49921)	(8669; 28532)	(19061; 55554)
BrazilInstance2	(18970; 60948)	(13885; 56391)	(40966; 119964)	(42741; 144184)	(22486; 78049)	(49567; 141622)
BrazilInstance6	(43092; 139046)	(31207; 127458)	(100368; 293330)	(95934; 324045)	(51137; 177698)	(120298; 343570)
FinlandArtificialSchool	(71018; 262510)	(58333; 307250)	(113823; 376180)	(111783; 404845)	(78790; 361175)	(146280; 445065)
FinlandHighSchool	(87772; 329322)	(61148; 355265)	(180374; 566921)	(172818; 623155)	(115530; 498945)	(234756; 710601)
Patras2010	(155972; 465784)	(90161; 643803)	(478325; 1317415)	(434521; 1438590)	(147494; 794651)	(535658; 1468263)
UniversityInstance4	(95640; 304196)	(60613; 538773)	(250889; 707300)	(243880; 821837)	(124541; 706977)	(314817; 875504)
ItalyInstance1	(25286; 85749)	(15914; 68367)	(33834; 115963)	(33834; 115963)	(30469; 105474)	(56103; 163900)
ItalyInstance4	(268116; 1192308)	(201690; 1376526)	(526698; 1877364)	(561489; 2217024)	(412821; 1931858)	(737829; 2432696)
KosovaInstance1	(962166; 3464743)	(768302; 3197313)	(1573136; 4836452)	(1329281; 4667662)	(986627; 3736796)	(1791461; 5375935)

Table 10. Number of variables and clauses for each encoding and instance.

In the previous experiments, (S, C, A) and $(S, S, -)$ performed the best among the encoding. It is also the case that they are also produce smallest instances in terms of clauses. This motivated us to investigate how well would the maxsat approach perform if each time we encode a cardinality constraint we select the encoding with the least amount of clauses. We did this for all instances and ran the experiments again under the same circumstances as in the third phase. Table 11 shows comparison between the third phase and this new idea:

Unfortunately, it is not always the case that the smallest possible encoding is the best, as in some cases it performs better and in other cases it performs worse. However, we believe that smaller encodings are generally preferred since in most of the cases the smallest encoding had a similar number of clauses compared to the two selected encodings, but the exact boarder when an encoding becomes better than another is unclear. Possibly a lot of different factors have impact on the encoding, such as redundant clauses which artificially increase the number of clauses, other previously encoded constraints may make some clauses redundant, etc.

4.9 Discussion and Lessons Learned

In this section we would to elaborate some lessons learned and interesting situations encountered during working on this research work.

Name	(S, C, A)	(S, S, -)	(smallest encoding)
BrazilianInstance1	**(0, 38)***	(0, 40)	(0, 39)
BrazilianInstance2	(0, 32)	**(0, 22)**	(0, 27)
BrazilianInstance4	**(0, 205)**	(0, 214)	(0, 206)
BrazilianInstance5	**(0, 117)**	(0, 169)	(0, 172)
BrazilianInstance6	**(0, 230)**	(0, 267)	(0, 240)
BrazilianInstance7	(0, 400)	(0, 481)	**(0, 368)**
SouthAfricaLewitt2009	**(0, 0)***	(0, 12)	(0, 14)
SouthAfricaWoodlands	**(0, 0)***	**(0, 0)***	**(0, 0)***
FinlandCollege	(0, 1523)	**(0, 900)**	(0, 1218)
FinlandHighSchool	**(0, 289)**	(0, 378)	(0, 523)
FinlandSecondarySchool	**(0, 252)**	(0, 358)	(0, 301)
FinlandArtificialSchool	(0, 47)	**(0, 3)**	(0, 7)
GreecePatras2010	**(0, 331)**	(0, 1587)	(0, 917)
GreeceWesternUniversity4	**(0, 121)**	(0, 148)	(0, 147)
GreeceHighSchool	**(0, 0)***	**(0, 0)***	**(0, 0)***
KosovaInstance	(0, x)	(0, x)	(0, x)
EnglandStPaul	(0, x)	(0, x)	(3, x)
ItalyInstance1	(0, 17)	**(0, 12)***	(0, 19)
ItalyInstance4	(0, 12825)	(0, 20007)	**(0, 18004)**

Table 11. Comparison of results with the smallest number of clauses encoding.

Solvers: We would like to reflect on how important it is to test out different SAT solvers when using SAT for applications. There are many different SAT solvers available and a number of of them must be tested in order to make conclusions on how effective is SAT for a specific application. We believe this is because each solver has its advantages and disadvantages and since application can have diverse properties and structures, results can be quite different, depending on how the chosen solver is designed. An example for this can be seen in our case. For our application, sat4j-maxsat performed the best, even though in the recent maxSAT competition it was outperformed by other solvers.

Another interesting observation is that sat4j-maxsat is known for not being designed to be as fast as possible, as it is written in Java. Nevertheless, it provided the best results. Taking a deeper look inside the internal workings of sat4j-maxsat and other solvers could prove to be beneficial for our future studies in order to try to asses why these solvers performs the way they do for our application.

As a final remark on the subject of solvers, for complex applications such as HSTT, we believe that solvers which operate exclusively by increasing lower bounds (with the objective of minimization) will likely perform worse. This is because calculating optimal solutions for such cases is usually very difficult task and therefore researchers often resort to finding nonoptimal *good* solutions. This was the case in our applications (e.g. WPM1 and munsat are lower bounding solvers, while sat4j-maxsat and optimax do not rely exclusively on lower bounds).

Special cases and concrete instances: When modeling a problem (e.g. HSTT), it is important to provide a general formulation which covers all cases. However, we would like to note that it is worthwhile to take a deeper look in special cases, as in some cases significantly simpler formulations can be provided, which may offer quicker solution times. In addition, taking a look at which concrete cases arise in instances is useful while developing a model.

We had several such cases in modeling of HSTT. When modeling the (soft) cardinality constraints, during our experimental phase the combinatorial encoding proved to be very useful, even though it is requires exponentially many clauses as the number of variables and cardinality requirements grow. There exist other more *sophisticated* encodings which did not perform so well, since the concrete instances had requirements which were frequentlys small enough for the combinatorial encoding. Similarly, identifying that $atMost$ can be converted to $atLeast$[5], that for soft constraints encoding that all literals must be set to true can simply be done by adding a series of weighted unit clauses, and other techniques described in previous sections all had positive impact on the performances.

Another important situation was when dealing with resource assignment constraints. The general formulation is correct and handles all possible resource assignments, but for most cases unpractical, because resource assignment usually has some pattern which can be exploited. For example, the FinlandArtificialInstance and EnglandStPaul instances had the requirement of assigning rooms to lectures were many rooms were identical. In FinlandArtificial, there were three types of rooms and if a lecture required a room of type A, any room of type A would be acceptable. A similar situation was present in EnglandStPaul, except all rooms were to be treated identical. Noticing these kinds of situation is very important, not only in order to reduce the size of the encoding (which in this case was significant), but also to tackle another more subtle problem which is described in the following point.

Symmetries: Identifying symmetries in the model is another very important issue. As a simple example, consider that three out of ten different objects must be chosen. We label each object with an integer and we associate a cost to each triplet representing a choice of three objects. In this scenario, $(1, 3, 5)$ represents the same choice as $(3, 1, 5)$ and thus has the same cost. This is what we call a symmetry, two $different$ solutions actually represent the same solution.

If symmetries are not properly dealt with, they may worsen the running times significantly, because solvers may not identify that different solutions are in fact identical and will go through essentially the same work multiple times. In the example above, by imposing that the triples must be ordered removes this problem, because $(1, 3, 5)$ is now the only feasible representation of those three objects. This way each solution has a unique representation. Note that this increases the complexity of the modeling, as it requires introducing new constraints which were not initially present. However, it may be worthwhile.

In our case for HSTT, we had several symmetries arising. One was described previously regarding assigning rooms to events for the FinlandArtificialInstance and EnglandStPaul. Here the problem was that for an event, one room out a certain pool of rooms needed to be assigned to it, but each room within the pool is treated as identical. When the solver was to decide which room to assign to an event, the solver had seemingly many choices, when in fact the room choice did not matter, as long as one was selected. This was solved by limiting the number of events that can take place at a time, as previously described.

Another situation was when encoding cardinality constraints. Apart from the mentioned ones, we tried a few other types as well. The main idea was similar to the simple example presented at the beginning of this section, where e.g. a cardinality constraint $exactly\_k$ would be represented as a k-tuple. The difference in encodings was how each number of the k-tuple was represented. Regardless of the representation, for each encoding a significant performance gain was obtained by enforcing

[5] E.g. instead of encoding that at most nine literals out of ten can be true, it is simpler to encode that at least one out of the ten negated literals must be true.

sorting among k-tuples, as in the example. Even though these encodings proved to be inferior to the ones currently, they clearly demonstrated the importance of symmetry breaking.

Breaking symmetries is useful for complete solvers, since it prevents the solver from revisiting the same solution multiple times. However, for incomplete solvers this may or may not be the case, as it may restrict the local search moves too much. This was for example the case in [24].

Try different (soft) cardinality encodings: There are many different cardinality encoding [18][6] and which encoding is the best in practice is unclear. While we can compare the number of auxiliary variables and clauses needed to enforce a cardinality encoding, we cannot conclude for sure which encoding will be better in practice. Therefore, when attempting to use SAT for an application, we suggest experimenting with several different (soft) cardinality encodings as the difference in performance can be significant, as observed in our experiments. We believe this is especially true if the cardinality encodings involve a smaller number of variables, like in our application (e.g. less than 30 most of the time). In addition, since cardinality constraints might be respectable part of the encoding (like in our case), this further confirms the need to experimenting with different encodings.

From our experimental results it appears to be the case that (especially for longer runs) the encoding with the least number of clauses is the best. Given that all of the encodings have the same semantics (restrict the assignments of literals), choosing the encoding with the least number of clauses is best because of implementation reasons, as with each variable (de)assignment the solver has to deal with less clauses.

Another way of examining cardinality encodings would be to consider the number of literals in the encoding. However, for our experiments, the difference between least number of clauses and least number of literals was marginal.

Important constraints: Applications with multiple constraints might have some constraints which are in a sense more important than others, but the constraint might not be always immediately obvious. Paying that constraint special attention, such as in the form of creating several different encodings for it, may prove to be useful. Naturally, this idea can be applied to all constraints, but we recommend with dealing with the most important constraint.

For HSTT, AssignTimeConstraints is a very fundamental constraint. It requires that a lesson must be scheduled for a certain number of hours. As seen in the experimental section, we have gave this constraint special treatment by experimenting with several different cardinality encodings for it. This kind of treatment proved to be useful and we believe this could possibly be the case for other complex applications.

SAT for timetabling: One of the key points of this paper is that (max)SAT is useful for HSTT. While this has also been the case for another different timetabling problem, namely the Curriculum Timetabling problem, HSTT is significantly more complex and it was unclear whether maxSAT would be useful for such a complicated problem.

We believe this is the case because there is a natural connection between SAT and timetabling, as some constraints of HSTT can naturally be represented by logical formulas.

5 Conclusion

In this paper, we have shown that the general High School Timetabling Problem [17] can indeed be modeled as a weighted partial maxSAT problem, despite the generality of the specification. We presented a complete and detailed encoding in the general sense as required by the specification, but also presented several alternative encodings for special cases. Different cardinality constraints were used and evaluated in order to find the most suitable encoding for the HSTT instances.

We evaluated the encodings by implementing and evaluating our approach on benchmark instances suggested and used by the International Timetabling Competition (ITC 2011) and compared our results with the GOAL, the winning solver of the ITC 2011. Our approach gives competitive results and outperforms GOAL for several problems. We also investigated the combination of our approach with the GOAL solver and the results for some instances could be still improved. Overall, the experimental evaluation shows that SAT solving is a robust tool for solving high school timetabling problems. We also presented in this paper several lessons learned regarding the encodings, constraints, and timetabling instances. Finally, the generated maxSAT instances encode practical and large timetabling problems and as such have been submitted to the maxSAT Competition 2014 and have proven to be challenging and useful benchmarks for maxSAT solvers.

For future work, there are a number of issues we would like to investigate. Developing new hybridization techniques (such as large neighborhood search algorithms) that utilize SAT solving is an interesting research direction. Furthermore, it is worth to study if encodings can be still optimized to better suit a particular instance.

6 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.

References

1. Eighth Max-SAT evaluation. http://maxsat.ia.udl.cat:81/13/introduction/index.html. Accessed: 2014-1-30
2. International timetabling competition 2011. http://www.utwente.nl/ctit/hstt/itc2011/welcome/. Accessed: 2014-1-30
3. The KHE high school timetabling engine. http://sydney.edu.au/engineering/it/ jeff/khe/. Accessed: 2014-9-16
4. Solvers of the eighth Max-SAT evaluation. http://maxsat.ia.udl.cat:81/13/solvers/index.html. Accessed: 2014-1-30
5. Achá, R.J.A., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and maxsat. Annals OR **218**(1), 71–91 (2014). DOI 10.1007/s10479-012-1081-x. URL `http://dx.doi.org/10.1007/s10479-012-1081-x`
6. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: O. Kullmann (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5584, pp. 167–180. Springer (2009). DOI 10.1007/978-3-642-02777-2_18. URL `http://dx.doi.org/10.1007/978-3-642-02777-2_18`
7. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In: Principles and Practice of Constraint Programming–CP 2003, pp. 108–122. Springer (2003)
8. Brito, S.S., Fonseca, G.H.G., Toffolo, T.A.M., Santos, H.G., Souza, M.J.F.: A SA-ILS approach for the high school timetabling problem. Electronic Notes in Discrete Mathematics **39**, 169–176 (2012)
9. Dorneles, A.P., de Arajo, O.C., Buriol, L.S.: A fix-and-optimize heuristic for the high school timetabling problem. Computers & Operations Research **52, Part A**(0), 29 – 38 (2014). DOI http://dx.doi.org/10.1016/j.cor.2014.06.023. URL `http://www.sciencedirect.com/science/article/pii/S0305054814001816`
10. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. JSAT **2**(1-4), 1–26 (2006). URL `http://jsat.ewi.tudelft.nl/content/volume2/JSAT2_1_Een.pdf`
11. Even, S., Itai, A., Shamir, A.: On the complexity of time table and multi-commodity flow problems. In: Foundations of Computer Science, 1975., 16th Annual Symposium on, pp. 184–193. IEEE (1975)

12. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance-specific algorithm configuration. In: H. Coelho, R. Studer, M. Wooldridge (eds.) ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings, *Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 751–756. IOS Press (2010). DOI 10.3233/978-1-60750-606-5-751. URL `http://dx.doi.org/10.3233/978-1-60750-606-5-751`
13. Kheiri, A., Ozcan, E., Parkes, A.J.: HySST: hyper-heuristic search strategies and timetabling. In: Proceedings of the Ninth International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012) (2012)
14. Kingston, J.: KHE14: An algorithm for high school timetabling. In: Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling, E. Ozcan, E. K. Burke, B. McCollum (Eds.), pp. 498–501 (2014)
15. Post, G., Ahmadi, S., Daskalaki, S., Kingston, J., Kyngas, J., Nurmi, C., Ranson, D.: An XML format for benchmarks in high school timetabling. Annals of Operations Research **194**(1), 385–397 (2012). DOI 10.1007/s10479-010-0699-9. URL `http://dx.doi.org/10.1007/s10479-010-0699-9`
16. Post, G., Di Gaspero, L., Kingston, J., McCollum, B., Schaerf, A.: The third international timetabling competition. Annals of Operations Research pp. 1–7 (2013). DOI 10.1007/s10479-013-1340-5. URL `http://dx.doi.org/10.1007/s10479-013-1340-5`
17. Post, G., Kingston, J.H., Ahmadi, S., Daskalaki, S., Gogos, C., Kyngäs, J., Nurmi, C., Musliu, N., Pillay, N., Santos, H., Schaerf, A.: XHSTT: an XML archive for high school timetabling problems in different countries. Annals OR **218**(1), 295–301 (2014). DOI 10.1007/s10479-011-1012-2. URL `http://dx.doi.org/10.1007/s10479-011-1012-2`
18. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: P. van Beek (ed.) Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3709, pp. 827–831. Springer (2005). DOI 10.1007/11564751_73. URL `http://dx.doi.org/10.1007/11564751_73`
19. Sørensen, M., Dahms, F.H.: A two-stage decomposition of high school timetabling applied to cases in Denmark. Computers & Operations Research **43**, 36–49 (2014)
20. Sørensen, M., Kristiansen, S., Stidsen, T.R.: International timetabling competition 2011: An adaptive large neighborhood search algorithm. In: Proceedings of the Ninth International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012), p. 489 (2012)
21. Sørensen, M., Stidsen, T.R.: Comparing solution approaches for a complete model of high school timetabling. Tech. rep., DTU Management Engineering (2013)
22. Sørensen, M., Stidsen, T.R.: Hybridizing integer programming and metaheuristics for solving high school timetabling. In: Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling, E. Ozcan, E. K. Burke, B. McCollum (Eds.), pp. 557–560 (2014)
23. Sørensen, M., Stidsen, T.R., Kristiansen, S.: Integer programming for the generalized (high) school timetabling problem. In: Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling, E. Ozcan, E. K. Burke, B. McCollum (Eds.), pp. 498–501 (2014)
24. Triska, M., Musliu, N.: An improved SAT formulation for the social golfer problem. Annals OR **194**(1), 427–438 (2012). DOI 10.1007/s10479-010-0702-5. URL `http://dx.doi.org/10.1007/s10479-010-0702-5`