FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SAT-Based Approaches for the General High School Timetabling Problem

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Emir Demirović
Registration Number 1228470

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

External reviewers:
Prof. Andrea Shaerf. University of Udine, Italy.
Univ.-Prof. Dr. oec. Martin Geiger. Helmut Schmidt University, Germany.

Vienna, 1st February, 2017

Emir Demirović	Nysret Musliu

Declaration of Authorship

Emir Demirović
Favoritenstrasse 9, 1040 Wien, Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 1st February, 2017

Emir Demirović

Personal Note

The topic of high school timetabling is of personal interest to me and has not been selected by chance. During my education, I have had first-hand experience with problems that suboptimal timetables bring, seen my peers struggle with similar issues, and in addition observed the pains that timetablers go through. It is timetabling that motivated me to pursue studying combinatorial optimization. The idea that various problems (in particular, timetabling) can be improved ”merely” by using already available resources in a more clever way, rather than investing into new resources, seemed compelling. Moreover, the decisions related to the mentioned resource usages can be left to a computer, which is able to examine significantly more scenarios than a human. It was a very exciting moment for me when I was given the opportunity to explore algorithms for high school timetabling as my PhD. In my thesis, I aimed to provide new scientific contributions to the timetabling community, but also offer schools (and other institutions) accessible and practical automatized software for generating timetables. The former is presented in the following text, while the latter remains to be done in the upcoming years, in order to improve the lives of hundreds of students and staff members.

Acknowledgements

I would like to wholeheartedly thank my supervisor Nysret Musliu. His expertise in combinatorial optimization and his warm personality made my PhD time very pleasant. He has been of tremendous support and has helped me to further grow as a professional in the field of combinatorial optimization. It was an extraordinary coincidence to have met Nysret just as I was starting my PhD, as our research interests aligned perfectly, and I am ever-so thankful that such a marvelous opportunity presented itself to me.

It goes without saying that I would not be the person I am today without the support of my friends and family, especially my mother. Their love has been vital for me as a person, but also as a professional. They have always been there for me, encouraging me to pursue my dreams, and providing both material and moral support, without which my thesis would not be possible.

Lastly, I would like to express my endless gratitude towards two people, who I often refer to as the heroes of my young adulthood: professors Željko Jurić and Ališer Sijarić. Although not directly influencing my PhD, both have molded my life in such a way that it is unimaginable for me now to lead life differently.

The work was supported by the Vienna PhD School of Informatics and the Austrian Science Fund (FWF): P24814-N23.

Abstract

High School Timetabling (HSTT) is a well known and widespread problem. The problem consists of coordinating resources (e.g. teachers, rooms), times, and events (e.g. lectures) with respect to various constraints. Unfortunately, HSTT is hard to solve and just finding a feasible solution for simple variants of HSTT have been proven to be NP-complete. In this work, we consider the general HSTT problem, abbreviated as XHSTT. Despite significant research efforts for XHSTT and other timetabling problems, no *silver bullet* algorithm has been found so far. Many problems have yet to be efficiently and/or optimally solved.

The main goal of this thesis is to explore the relation between propositional logic and high school timetabling, as well as related approaches. We model the complex formalism of XHSTT using Boolean variables and basic logical connectives only. We evaluated different cardinality constraint encodings, solvers, and important special cases in order to significantly simplify the modeling in practice. We note that resource assignment constraints have been considered only for special cases, rather than in general. In addition, we investigated a maxSAT-based satisfiability modulo theories (SMT) approach. Another model we studied in this work is based on bitvectors. By using a series of bitvector operations (such as *AND, OR*, and *XOR*) on the set of event bitvectors, we were able to model all constraints, with the exception of resource assignment constraints. The bitvector models serves as an efficient data structure for local search algorithms such as hill climbing and simulated annealing. To integrate maxSAT into a hybrid algorithm, we combined local search with a large neighborhood search algorithm that exploits maxSAT. Furthermore, to the best of our knowledge, it is the first time maxSAT is used within a large neighborhood search scheme.

We carried out thorough experimentation on important benchmark instances that can be found in the repository of the third international timetabling competition (ITC 2011) and compared with the state-of-the-art algorithms for XHSTT. Detailed experiments were performed in order to determine the most appropriate maxSAT solvers and cardinality constraint encodings, evaluate our SMT approach, and compare with integer programming and the ITC 2011 results. Computational results demonstrate that we outperform the integer programming approach on numerous benchmarks. We are able to obtain even better results by combining several maxSAT solvers. When compared to the leading KHE engine for XHSTT, the bitvector modeling approach provided significant improvements

for local search algorithms such as hill climbing and simulated annealing. Lastly, our large neighborhood search algorithm excelled in situations when limited computational time is allocated, being able to obtain better results than the state-of-the-art solvers and the pure maxSAT approach in many benchmarks.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

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

Unfortunately, high school timetabling is hard to solve and merely finding a feasible solution of simple variants of high school timetabling has been proven to be NP-complete [EIS75]. Apart from the fact that practical problems can be very large and have many different constraints, high school timetabling requirements vary from country to country. Due to this, many variations of the timetabling problem exist. A lot of progress has been made and HSTT is still an active field of research, even having its own specific competition in 2011 - the third International Timetabling Competition (ITC 2011).

To introduce standardization, researches have proposed a general high school timetabling problem formulation [PDGK$^+$13, PKA$^+$14] called XHSTT. This was an important step for HSTT, as before it was difficult to compare HSTT algorithms, as authors from different countries would consider similar, albeit different HSTT problems. Therefore, it was unclear what was the state-of-the-art. With the introduction of XHSTT, algorithms could now be fairly compared on well-established benchmarks. The ITC 2011 has endorsed this

formulation and attracted 17 research groups from across the globe, further encouraging and advancing research for HSTT. XHSTT is still the de facto standard for high school timetabling and it is the formulation that we consider in this thesis.

There has been significant research done for XHSTT and other timetabling problems, but many problems have yet to be efficiently or optimally solved. New methodologies and techniques are continuously being developed. Many different methods for solving XHSTT exist with most of the research focused around local search algorithms. The winner of ITC 2011, GOAL [dFST+16], used a simulated annealing and iterated local search approach. It was later outperformed by a variable neighborhood search algorithm [FS14]. Another approach, named KHE14 [Kinb], is based on ejection chains and uses procedures specialized for each XHSTT constraint to repair and improve solutions. Recently hyper-heuristics have been used successfully for XHSTT [AÖK15, KK16], where a high-level heuristic dictates the usage of low-level heuristics depending on the instance. An exact approach in the form of integer programming has been proposed in [KSS15]. A decomposition technique in [Sør13a] splits the solution process into two phases: time and resource assignment phases. Several hybrid techniques have been developed which combine integer programming and large neighborhood search [SS, FSC16a, DdAB14].

Unfortunately, despite significant research efforts, no *silver bullet* algorithm has been found so far. Each state-of-the-art algorithm has its own scenarios where it performs better than the competition. For example, KHE14 is very effective at producing high quality solutions given very limited computational time (e.g. a few minutes). However, it cannot reach (or prove) optimal solutions as some other methods even if longer times are considered. The integer programming approach is somewhat the opposite, where good or optimal solutions can be obtained when significantly more time is allocated, but is not as competitive when solutions need to be generated very quickly. The other methods typically lie in between these two extremes, where the quality of their solutions depends on the structure of the instances in question. Therefore, presenting new methods, ideas, and view points are very important for this active field of research.

1.2 Goal of the Thesis

In a broader sense, the goal of this work is to investigate innovative approaches and algorithms for XHSTT. This includes formally modeling XHSTT, developing and analyzing new methods and paradigms, and lastly implementing algorithms which, given an input of timetabling requirements, produce high quality timetables without further human intervention. By doing so, we strive to improve the state-of-the-art for XHSTT and motivate further research.

More specifically, we aim to explore the relation between propositional logic and high school timetabling, as well as related approaches. The problem of determining whether a propositional logic formula has a solution is called the satisfiability problem (abbreviated as SAT). In this work, we investigate SAT-based approaches for high school timetabling. There is an innate connection between timetabling and SAT, as it has many logic charac-

teristics and as such some of its constraints can be naturally expressed in propositional logic (e.g. if assignment X is made, then Y cannot be made). This and the fact that it has not yet been done for XHSTT motivated us to further investigate different paradigms and approaches related to propositional logic and see if the gap between SAT-based approaches and high school timetabling can be bridged. In addition, we would like to carry out a research on the use of these methods in hybrid algorithms by combining them with local search and large neighborhood search.

Thus, the main goals of this thesis are:

- Study if SAT-based approaches are suitable for XHSTT. In particular, the objective is to investigate whether XHSTT can be expressed as a propositional logic formula and if the obtained SAT problems can be addressed by existing SAT solving techniques. Furthermore, we would like to research different modeling strategies and determine their impact on the solution process.
- Investigate other SAT-related modeling approaches for XHSTT, such as bitvector modeling. Compare the models and determine which one is the most appropriate. In addition, analyze whether the developed models can be exploited in local search algorithms.
- Examine the possibility of hybridizing the developed SAT-based approaches with local search and large neighborhood search. Use domain-specific knowledge to augment the SAT solving algorithm by initially focusing the search in areas that are most likely to contain high quality solutions, and gradually increasing the allowed scope of search until the complete search space is exhaustively explored.
- Compare the developed approaches with the state-of-the-art for XHSTT on important benchmark instances from the literature.

These goals are addressed in further text, with the first three points having their own dedicated chapters. In each chapter we provide a detailed comparison with the state-of-the-art to assess our approach.

1.3 Main Contributions

The main contributions of this thesis are the following:

- We modeled XHSTT as a propositional logic formula, using Boolean variables and basic logical connectives only. Hard constraints are translated into propositional Boolean formulas (SAT). To account for the soft constraints, the model is extended with the use of Partial Weighted maxSAT. We evaluated different cardinality constraint encodings, solvers, and important special cases in order to significantly simplify the modeling in practice. We note that resource assignment constraints

have been considered only for special cases, rather than in general. In addition, we investigated a maxSAT-based satisfiability modulo theories (SMT) approach. The experimental results on numerous benchmarks prove that our maxSAT approach provides competitive results, outperforming the state-of-the-art complete approach based on integer programming.

- We provided a bitvector modeling of XHSTT. A single bitvector is associated with each event (e.g. a lecture). Each bit can take the values zero or one, indicating if the event assigned to it is taking place at a discrete time associated with the bit. By using a series of bitvector operations (such as *AND, OR*, and *XOR*) on the set of event bitvectors, we were able to model all XHSTT constraints, with the exception of resource assignment constraints. One of the key points of the bitvector modeling is that modern processors have built-in support for bitvector operations, allowing us to efficiently compute constraint violations. The resulting bitvector model is mainly used as an efficient data structure for the representative of XHSTT for local search algorithms. In addition, SMT solvers that support the bitvector theory can be used to generate XHSTT solutions.

- We developed a hybrid algorithm which exploits maxSAT for XHSTT. We combined local search and large neighborhood search to solve XHSTT instances which are modeled as maxSAT (as described in Chapter 4). Iteratively, starting from an arbitrary solution, a part of the solution is *destroyed* by using one of the two *neighborhood vectors*, and is then *repaired* by maxSAT. We have experimented with several variants in order to show the importance of each component of the algorithm. Our approach outperformed the state-of-the-art solvers on important benchmarks. Furthermore, to the best of our knowledge, it is the first time maxSAT is used within a large neighborhood search scheme.

1.4 Publications

Some of the results presented in this thesis have been included in the following publications:

- **Journals**
  - **MaxSAT-Based Large Neighborhood Search For High School Timetabling**,
    Emir Demirović, Nysret Musliu,
    Computers & Operations Research, 78:172–180, 2017.
  - **Modeling High School Timetabling With Bitvectors**,
    Emir Demirović, Nysret Musliu,
    Annals of Operations Research, doi:10.1007/s10479-016-2220-6, 2016.
- **Conferences and Workshops**
  - **Solving High School Timetabling with Satisfiability Modulo Theories**,
    Emir Demirović, Nysret Musliu,
    Proceedings of PATAT-14, pages 142-166.
  - **Modeling High School Timetabling as Partial Weighted maxSAT**,
    Emir Demirović, Nysret Musliu,
    LaSh 2014: The 4th Workshop on Logic and Search, a SAT/ICLP workshop at FLoC 2014.

Chapter 4 is a significant extension of the workshop paper and is planned to be published as a journal paper. In addition, Chapter 3 will be the basis of another journal publication.

1.5 Structure of This Work

In the next chapter we introduce the general high school timetabling problem XHSTT, both informally and formally. In Chapter 3 we provide an overview of the state-of-the-art for XHSTT. The most important works are selected and their contents are discussed in enough detail to understand the general idea of the work. This is followed by the main contributions of this thesis: Chapters 4, 5, and 6. Each of these chapters presents a distinct approach to XHSTT and answers one of the goals of the thesis. A comparison to the state-of-the-art is given immediately after presenting the details of the approach in order to demonstrate its effectiveness. In Chapter 4 we present our modeling of XHSTT as a propositional logic formula, evaluate various encodings, solvers, techniques, an SMT approach, and compare to integer programming. A bitvector model proposed as an alternative way of representing XHSTT is discussed in Chapter 5. In Chapter 6 we present our large neighborhood search algorithm which exploits maxSAT to solve XHSTT. Lastly, in Chapter 7 we give a conclusion and discuss future work.

CHAPTER 2

Problem Description

In this chapter we describe the general high school timetabling problem XHSTT. We give a simple informal introduction, followed by a detailed formal problem statement, which is a translation of the problem definition discussed in [PDGK+13, PKA+14] into a more formal language.

2.1 Informal Description

High school timetabling has been studied extensively in the past. However, a lot of work has been done exclusively for country-specific educational systems, which resulted in many different timetabling formulations. Thus, it was difficult to compare algorithms and clearly determine the state-of-the-art. To solve this issue and encourage timetabling research, researchers agreed on a standardized formulation called XHSTT [PDGK+13, PKA+14]. This formulation was general enough to be able to model education systems from different countries and was endorsed by the International Timetabling Competition 2011. This is the formulation used in our work.

XHSTT specifies three main entities: times, resources, and events. Times refer to the available discrete time units, such as Monday 9:00-10:00 and Monday 10:00-11:00. Resources correspond to, for example, available rooms, teachers, students. 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 the teacher and the student group are resources) and two units of time (two *times*). Events are to be scheduled into one or more *solution events* or *subevents*. For example, a mathematics lecture with a total duration of four hours can be split into two subevents with a duration of two hours each, but can also be scheduled as a single subevent with a duration of four hours. Constraints may restrict the durations of subevents.

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

We now give an informal overview of all the constraints in XHSTT (as reported in [PKA+14]). There is a total of 16 constraints (excluding preassignments of times or resources to events, which are not listed explicitly). The constraints apply to a specified subsets of events, resource, or times.

Constraints related to events:

1. Assign Time Constraints – events must be assigned a certain number of times. Typically all events are included in this constraint.
2. Split Events Constraints – the duration and amount of subevents must be within certain limits. Distribute Split Events Constraints (below) gives further control on subevents.
3. Distribute Split Events Constraints – the number of subevents with a particular duration must be within certain bounds.
4. Prefer Times Constraints – stated times are preferred over others for specified events.
5. Avoid Split Assignments Constraints – subevents derived from the same events should be assigned the same resources.
6. Spread Events Constraints – events must be spread over the week.
7. Link Events Constraints – events must take place simultaneously.
8. Order Events Constraints – certain events may only take place after other events have finished.

Constraints related to resources:

1. Assign Resource Constraints – resources must be assigned to events.

2. Prefer Resources Constraints – certain resources are preferred over other ones for given events.

3. Avoid Clashes Constraints – resources cannot be used by two or more subevents at the same time.

4. Avoid Unavailable Times Constraints – resources cannot be used at certain times.

5. Limit Idle Times Constraints – resources must have their number of idle times for specified days lie between given values.

6. Cluster Busy Times Constraints – resources' activities must all take place within a minimum and maximum number of days.

7. Limit Busy Times Constraints – resources within specified days must have their number of busy times lie between given values.

8. Limit Workload Constraints – resources must have their workload lie between given values.

We now proceed with the formal description of these constraints.

2.2 Formal Description

In this section we give our mathematical formulation of the problem description given in [PDGK$^+$13, PKA$^+$14]. The main entities of XHSTT are events (e.g. lessons that need to be scheduled). Each event has a predetermined duration (e.g. a mathematics lesson has a duration of two hours) and requirements for resources (e.g a lesson requires one teacher and one room). Events are split into one or more subevents, as previously described in Section 2.1. Note that an event may be "split" into a single subevent. The resources can be predetermined or left open to the solver to assign them. Every resource requirement in an event has two additional properties: its *role* and *workload*. These two properties are described in Section 2.2.3 under Assign Resource Constraints and Limit Workload Constraints. In XHSTT terminology, subevents and roles correspond to *solution events* and *event resources*, respectively.

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

Events, resources, and times can be grouped into any number of event, resource, or time groups. Groups are used when defining constraints.

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

2.2.1 Variables

1. $X_{e,i,t} = 1$ if the subevent i of event e takes place at time t, and $X_{e,i,t} = 0$ otherwise.

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

3. $M_{e,i}^{k,r} = 1$ if the subevent i of event e is assigned resource r for the role k, and $M_{e,i}^{k,r} = 0$ otherwise. We constrain the assignments further by stating that for any given e, k, i, there can be at most one r such that $M_{e,i}^{k,r} = 1$. The importance of these variables will become apparent when modeling resource assignment constraints.

4. $X_{r,t} = 1$ if resource r is being used at time t by any event and $X_{r,t} = 0$ otherwise.

A solution to XHSTT is an assignment of values to all variables. The solution quality is evaluated as a sum of constraint violations. These are discussed in detailed in Section 2.2.3.

2.2.2 Functions

1. $duration(e)$ computes the duration of e, but this depends on whether e is an event or a subevent. If e is an event, it returns the total duration for an event e. In this case the value is fixed within an given instance. If e is a subevent, it returns the duration of the subevent. Note that for subevents this value is determined by the end solution and is not fixed up front, although constraints may impose restrictions.

2. $subevents(e)$ returns the set of subevents that event e has been divided into. For similar reasons as above, the set is not fixed and depends on the final solution, although the sum of the durations of subevents of e must be equal to *duration(e)*.

3. $events(R)$ computes the set of events to which resource R has been assigned to.

4. $step(x) = 1$ if $x > 0$, otherwise $step(x) = 0$.

5. $bound\_violation(x, a, b) = |a - x| * step(a - x) + |x - b| * step(x - b)$. If the value x lies in the interval $[a, b]$, it will evaluate to zero. Otherwise, it will evaluate to how far x is from the interval. This function is important as it is frequently used in XHSTT, as a common constraint is that a certain property value should be within a given interval (hard constraint), or as close to it as possible (soft constraint).

6. $equal(a, b) = 1 - step(a - b) - step(b - a)$. Evaluates to one if the arguments a and b are equal and zero otherwise.

2.2.3 Constraints

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

A *point of application* for a constraint is the object to which the constraint applies to. For example, for prefer times constraint, a point of application is a single event. An integer value called the *deviation* is computed for each constraint's point of application, which represents how severe the violation of the constraint is at that point. The procedure for computing the deviation is unique and is described for each constraint individually.

A cost function is applied to the deviation and the value produced is multiplied by a weight in order to obtain the cost of the particular point of application. There are three cost functions that can be applied to the deviation: linear, quadratic, and step. The linear function makes no changes to the deviation, the quadratic function squares the deviation, while the step function evaluates to 1 if the deviation is nonzero and evaluates to 0 otherwise. The cost of a constraint is the sum of its deviations.

Constraints are labeled as either hard or soft. The goal is to minimize the sum of the costs of hard constraints and then minimize the sum of the costs of soft constraints. Note that no constraint is predefined as hard or soft, as this is left for the instance modeler to determine based on specific timetabling needs.

In the following, we describe the constraints and the way the deviation is computed at a single point of application. Auxiliary functions used in the constraint are given immediately below the equation which describes the deviation computation. Recall that in order to compute the cost of a particular constraint, one calculates the deviation for each point of application and applies a cost function to it, sums up all these values, and multiplies them by the given constraint's weight. Each constraint defines a number of sets (e.g. E_{spec} and TG_{spec}) and constants, as appropriate.

Assign Time Constraints

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

$$deviation_{atc}(e) = (\sum_{i \in subevents(e)} \sum_{t \in T} X_{e,i,t}) - duration(e) \tag{2.1}$$

Split Events Constraints

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

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

$$deviation_{splitec}(e) = bound\_violation(|subevents(e)|, d_{min}^{splitec}, d_{max}^{splitec}) + a_{splitec}(e) \quad (2.2)$$

$$a_{splitec}(e) = \sum_{i \in subevents(e)} bound\_violation(duration(i), A_{min}^{splitec}, A_{max}^{splitec}) \quad (2.3)$$

Distribute Split Events Constraints

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

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

$$deviation_{dsec}(e) = bound\_violation(a_{dsec}(e), A_{min}^{dsec}, A_{max}^{dsec}) \quad (2.4)$$

$$a_{dsec}(e) = \sum_{i \in subevents(e)} equal(duration(i), d) \quad (2.5)$$

Prefer Times Constraints

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

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

$$deviation_{ptc}(e) = \sum_{i \in subevents(e)} \sum_{t \in \text{notPrefTimes}} S_{e,i,t} * duration(i) \tag{2.6}$$

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

Spread Events Constraints

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

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

$$deviation_{spreadec}(eg) = \sum_{tg \in TG_{spec}} bound\_violation(a_{spreadec}(eg, tg), A_{min}^{spreadec}, A_{max}^{spreadec}) \tag{2.7}$$

$$a_{spreadec}(eg, tg) = \sum_{e \in eg} \sum_{i \in subevents(e)} \sum_{t \in tg} S_{e,i,t} \tag{2.8}$$

Avoid Clashes Constraints

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

As mentioned in the beginning of the section, each resource requirement for an event e has an additional property: its *role* and its desired resource type. For example, a sport lesson may require two resources of type *teacher* with the following roles: *senior* and *junior*. For a particular role k, all requests for resources with the role k must be of the same resource type.

When an event is split into multiple subevents (including the case of a single subevent), each subevent is assigned a (possibly different) resource to satisfy the role requirement. Once a resource is assigned to a subevent, it remains in its use during the complete duration of the subevent. As noted in Section 2.2.1, the variables $M_{e,i}^{k,r}$ determine the role resource assignments to subevents. At most one resource may be assigned to satisfy the role requirements for a subevent. If a resource is preassigned to an event, then the corresponding variables $M_{e,i}^{k,r}$ are fixed with respect to the reassignment. Otherwise,

the solver needs to assign resources to events, which is done via the assign resource constraints.

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

$$deviation_{acc}(r) = \sum_{t \in T} step(a_{acc}(r,t)) * a_{acc}(r,t) \tag{2.9}$$

$$a_{acc}(r,t) = (\sum_{e \in events(r)} \sum_{i \in subevents(e)} \sum_{k \in roles(e)} M_{e,i}^{k,r} * X_{e,i,t}) - 1 \tag{2.10}$$

The function $a_{acc}(r,t)$ calculates how many events are using resource r at time t.

Limit Idle Times Constraints

This constraint specifies the minimum and maximum number of times in which a resource can be idle during times in specified time groups. For example, a typical constraint is to impose that students should not have any idle times. An idle time for a resource r within a time group tg is a time $t \in tg$ in which r is not busy, but is busy at some other times before and after t within tg.

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

$$deviation_{litc}(r) = bound\_violation(a_{litc}(r), A_{min}^{litc}, A_{max}^{litc}) \tag{2.11}$$

$$a_{litc}(r) = \sum_{tg \in TG_{spec}} \sum_{t \in tg} I(tg,t,r) \tag{2.12}$$

$$I(tg,j,r) = before(tg,j,r) * (1 - X_{r,j}) * after(tg,j,r) \tag{2.13}$$

$$after(tg,j,r) = step(\sum_{t \in tg \wedge t > j} X_{r,t}) \tag{2.14}$$

$$before(tg,j,r) = step(\sum_{t \in tg \wedge t < j} X_{r,t}) \tag{2.15}$$

Avoid Unavailable Times Constraints

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

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

$$deviation_{autc}(r) = \sum_{t \in UT} X_{r,t} \tag{2.16}$$

Cluster Busy Times Constraints

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

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

$$deviation_{cbtc}(r) = bound\_violation(b_{cbtc}(r), A^{cbtc}_{min}, A^{cbtc}_{max}) \tag{2.17}$$

$$b_{cbtc}(r) = \sum_{tg \in TG_{spec}} a_{cbtc}(tg, r) \tag{2.18}$$

$$a_{cbtc}(tg, r) = step(\sum_{t \in tg} X_{r,t}) \tag{2.19}$$

Limit Busy Times Constraints

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

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

$$deviation_{lbtc}(r) = \sum_{tg \in TG_{spec}} (bound\_violation(a_{lbtc}(tg, r), A^{lbtc}_{min}, A^{lbtc}_{max}) * step(a_{lbtc}(tg, r)) \tag{2.20}$$

$$a_{lbtc}(tg, r) = \sum_{t \in tg} X_{r,t} \tag{2.21}$$

Link Events Constraints

Certain events must be held at the same time. For example, sport lessons for all classes of the same year must be held simultaneously.

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

$$deviation_{lec}(eg) = \sum_{t \in T} step(a_{lec}(eg, t)) * (1 - equal(a_{lec}(eg, t), |eg|)) \tag{2.22}$$

$$a_{lec}(eg, t) = \sum_{e \in eg} \sum_{i \in subevents(e)} X_{e,i,t} \tag{2.23}$$

Order Events Constraints

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

Formally, it imposes that two specified events must take place one after the other with a minimum A^{oec}_{min} and maximum A^{oec}_{max} times between them. The point of application is a pair of events and the deviation for a single pair of events $ep = \langle e_1, e_2 \rangle$ is calculated as follows:

$$deviation_{oec}(ep) = bound\_violation(b_{oec}(e_2) - a_{oec}(e_1), A^{oec}_{min}, A^{oec}_{max}) \tag{2.24}$$

$$b_{oec}(e_2) = min\{t : i \in subevents(e_2) \wedge S_{e_2,i,t} = 1\} \tag{2.25}$$

$$a_{oec}(e_1) = max\{t + duration(i) : i \in subevents(e_1) \wedge S_{e_1,i,t} = 1\} \tag{2.26}$$

Assign Resources Constraints

This constraint specifies that resources must be assigned to certain events. Assign Resource Constraints (ARC) is defined with a specific role k and merely states that resource assignment requests with the role k must be fulfilled, while other constraints may limit the assignments of resource to certain roles. For example, a lesson might require a room to take place, leaving the solver to decide which room it will use.

Formally, it imposes that resources of the specified type must be assigned to specified events to fulfill the role k. Let R_{type} denote the set of resources of the specified type. The point of application is an event and the deviation for a single event e is calculated as follows:

$$deviation_{arc}(e,k) = \sum_{i \in subevents(e)} (1 - a_{arc}(e,i,k)) * duration(i) \tag{2.27}$$

$$a_{arc}(e,i,k) = \sum_{r \in R_{type}} M_{e,i}^{k,r} \tag{2.28}$$

Prefer Resources Constraints

This constraint specifies for certain events and roles which resources are allowed (hard constraint) or preferred (soft constraint). For example, for the role *senior teacher* only a subset of the teachers may be considered.

Formally, let *notPrefResources* denote the set of resources which are not preferred for the role k. The point of application is an event and the deviation for a single event e is calculated as follows:

$$deviation_{arc}(e,k) = \sum_{i \in subevents(e)} a_{arc}(e,i,k) * duration(i) \tag{2.29}$$

$$a_{arc}(e,i,k) = \sum_{r \in \text{notPrefResources}} M_{e,i}^{k,r} \tag{2.30}$$

Events to which no resource has been assigned are effectively ignored by this constraint.

Avoid Split Assignments Constraints

This constraint specifies that for certain events and roles a single resource must (or should) be used throughout the complete duration of the events. For example, a mathematics lecture for class c must always take place in the same room, but the room itself is not determined before hand.

Formally, it limits the number of different resources that may be used through the duration of an event for the role k. The point of application is an event and the deviation for a single event e is calculated as follows:

$$deviation_{asac}(e,k) = step(a_{asac}(e,k)) * a_{asac}(e,k) \tag{2.31}$$

$$a_{asac}(e,k) = (\sum_{i \in subevents(e)} \sum_{r \in R} M_{e,i}^{k,r}) - 1 \tag{2.32}$$

Events to which no resource has been assigned are effectively ignored by this constraint.

Limit Workload Constraints

This constraint imposes that specified resources must have a certain workload. For example, a teacher must teach six hours per week, but he or she is not preassigned to any events.

The *workload(e, r)* is real number defined for every pair of events and resources. The value can be explicitly stated for the pair (e, r). If it is not, it assumes the value of the workload of the event, which is either given or equal to its duration, depending on whether it was specified or not.

Formally, it limits the minimum W_{min} and W_{max} workload of a resource. Let *role(e)* be the set of roles required by event e. The point of application is a resource and the deviation for a single resource r is calculated as follows:

$$deviation_{lwc}(r) = bound\_violation(\lceil b_{lwc}(r) \rceil, W_{min}, W_{max}) \tag{2.33}$$

$$b_{lwc}(r) = \sum_{e \in E_{spec}} \sum_{i \in subevents(e)} \sum_{k \in role(e)} M_{e,i}^{k,r} * a_{lwc}(e,i,r) \tag{2.34}$$

$$a_{lwc}(e,i,r) = \frac{duration(i) * workload(e,r)}{duration(e)} \tag{2.35}$$

With this constraint we conclude the formal description of XHSTT.

CHAPTER 3

State-of-the-Art for XHSTT

The aim of this chapter is to provide an overview of the most important XHSTT-based research. As noted in Chapter 2, there has been done a lot of work for high school timetabling. However, it is difficult to compare the works, as different authors considered similar, albeit different, timetabling problems. This motivated researchers to devise a standardized formulation for high school timetabling: one which would be general enough to accommodate all the diverse constraints and requirements. Thus, XHSTT [PDGK$^+$13, PKA$^+$14] was introduced. In the light of this, our work is based on XHSTT and we review state-of-the-art research on timetabling related to XHSTT. For a survey on other timetabling work, we direct the interested reader to [Pil14, Sch99]. We note that we do not include the work done in this thesis (Chapters 4, 5, and 6) in our overview.

We classified the surveyed works based on the solution approach. We now list all of the papers and provide brief descriptions of their content to give an overview of the state-of-the-art methods and techniques used. Afterwards, we allocate a section to each of the selected papers and describe them in more detail.

- Stochastic local search algorithms
  - GOAL [dFST$^+$16] – the winner of the ITC 2011. The algorithm generates an initial solution using KHE [kina] (described later on) and then combines iterative local search with simulated annealing to improve the solution. Today this approach is outperformed by its successors (see next two points).
  - Variable neighborhood search (VNS) – these algorithms [FS14] share a common search pattern: iteratively, based on the currently examined solution, a random solution is chosen from one of the available neighborhoods, after which a decent method is applied. The resulting solution is then accepted to take place instead of the current one if certain criteria is fulfilled. The main difference between the VNS algorithms is the acceptance criteria and the neighborhoods.

  - Late acceptance hill climbing [FSC16b] – a local search algorithm which accepts solutions based on their comparison with the *k-th* previous solution, rather than comparing with the best solution found so far.

- Hyper-heuristics [KOP, AÖK15, KK16] are a general domain independent strategy which is able to automatically adapt for a wide range of problems. Hyper-heuristics are high-level heuristics that are used to *generate* or *select* other low-level heuristics. They are usually equipped with a mechanism that allows self-tuning to the problem in question. Such approaches were considered for XHSTT, where a set of low-level heuristics was defined (e.g. moving an event from one time to another, or exchanging the assigned times of two events) and hyper-heuristics determined the way these low-level heuristics were used. In Section 3.2, we narrow our focus to hyper-heuristics that gave good results for XHSTT.

- KHE algorithms [Kinb] represent a set of complex algorithms for high school timetabling, all of which are distributed and implemented with KHE. According to its website [kina], KHE is an "open source ANSI C software library, written by Jeff Kingston, whose main aim is to provide a fast and robust foundation for solving instances of high school timetabling problems". All of the algorithms are based on local search, but are significantly different from the other ones presented here, which is why a separate category has been dedicated to them. Additionally, other algorithms (e.g. the VNS approach [FS14] and our work in Chapter 6) use (some) of these methods. Here we highlight only the (arguably) most important parts and refer the interested reader for more details to [Kinb]. The main focus of this work is to provide high quality solutions within very limited time (a few minutes).

  - Initial solution generation [Kinc] is done in several phases, according to hierarchical timetabling: structural, time assignments, and resource assignments phase. This alone allows other algorithms to provide further optimization (for example, simulated annealing and VNS algorithms). Ejection chains can be used to augment the process, as explained below, resulting in a fast and effective algorithm.
  - Polymorphic ejection chains [Kinb] focus directly on repairing *defects* (violations of constraints). Constraint violations are examined individually and specialized repairing procedures are developed for most constraints. While repairing one defect, new defects might be introduced, and the process is repeated recursively until there are no new defects, or other criteria are met (for example, the maximum chain length is reached).

- Decomposition of XHSTT [Sør13a] divides the solution process into two phases: time and resource assignments. The solution to the first phase is passed to the second phase as a parameter. Information about the second phase is incorporated into the first phase to ensure that good solutions can be found using the decomposition. In some cases the decomposition guarantees optimality.

- Integer programming (IP) [KSS15] models XHSTT as an integer program and then a commercial solver is used to generate solutions. Solving is done in two phases. The first phase optimizes hard constraints, after which soft constraints are added. This is the only exact algorithm for XHSTT, apart from the one presented in this thesis. By exploring the complete search space, it guarantees to find an optimal solution if given sufficient time. Typically longer running times are required when compared to the other (heuristic) approaches.
- Hybrid approaches combine metaheuristics with complete algorithms. The motivation is that complete methods perform well when considering a small number of variables. In an iterative fashion, a metaheuristic algorithm is used to select small subproblems which the complete method can efficiently solve. The complete method determines the best assignments for variables that are related to a *neighborhood*, while the other variables are fixed with respect to the current solution. For example, if the neighborhood consists of a single teacher, than the complete algorithm will calculate the best assignment for the given teacher's events, without changing the assignments of the other events. All algorithms presented here use integer programming as their complete method of choice.
  - Matheuristic [SS, Sør13b] - an adaptive large neighborhood search approach. The size of the neighborhood changes during the execution of the algorithm, depending on the estimate of how easy the neighborhoods are to solve (based on the *IP gap*). In addition, when selecting a neighborhood, preference is given to the ones that have previously been able to improve the solution.
  - VNS and IP [FSC16a] - a hybrid algorithm that uses a large neighborhood search algorithm to further improve the initial solution obtained by the VNS approach [FS14]. The neighborhoods consist of a set of randomly chosen resources. The size of the neighborhood is adjusted at run-time, depending on the estimate of the algorithm's performance.
  - Fix-and-Optimize [DdAB14] – an algorithm used to solve a special case of XHSTT. Neighborhoods consisting of k teachers or classes are considered, where k is initially set to two and increased with time.

In the following we look at each algorithm in more detail.

3.1 Simulated Annealing and Variable Neighborhood Search

In this section we describe the winner of the ITC 2011 and its successors following a similar algorithm theme. The common element these algorithms share is that they start from an initial solution and optimize through a sheer amount of randomized local search moves, using several neighborhoods. The initial solution is generated using KHE (see its description in Section 3.3.1) and here we describe the optimization algorithm.

3.1.1 GOAL

This algorithm [dFST+16] was the winner of the ITC 2011, providing the best solutions (at that time) on most instances. Today it is outperformed by its successors [FS14, FSC16a], but we still believe it is worth mentioning in order to get an overview of how algorithms for XHSTT evolved. The main part of the algorithm is simulated annealing. Afterwards, iterated local search is used to provide final improvements to the best solution found.

Simulated Annealing

Simulated Annealing (SA) was first proposed in [Kir84] as a probabilistic metaheuristic based on an analogy with the physical process of heating a material and then cooling it to decrease defects.

An important concept for SA is the neighborhood of a solution. Abstractly speaking, given a solution s, the neighborhood of s is a set of other solutions that are related to s according to some specified criteria. Typically the neighborhood solutions are somehow similar to S. The neighborhoods used are described later on.

The cost difference of two solutions s_1 and s_2 is denoted as $\Delta(s_1, s_2)$. A negative value for $\Delta(s_1, s_2)$ states that s_1 has a lower cost than s_2, meaning s_1 is a better solution in terms of the objective cost function (for a minimization problem). In SA, given a current solution s_0, a random solution s_x is drawn from the neighborhood of s_0, $N(s_0)$. If $\Delta(s_x, s_0) \leq 0$, then the newly generated solution s_x is *accepted*, that is, it takes the place of the current solution s_0, $s_0 = s_x$. Otherwise if $\Delta(s_0, s_x) > 0$, the solution is accepted with probability $e^{-\frac{\Delta}{T}}$, where T is the *temperature*. Initially during SA, T is a set to a high value and as the algorithm iterates the temperature is decreased (for example, $T = T * \alpha$, where $\alpha \in (0, 1)$). A high (low) value for the temperature T makes the probability of acceptance higher (lower), encouraging diversification (intensification). After T reaches a very low value, *reheating* takes place, increasing the value of T. This process is repeated iteratively until a stop criteria is met. In the case of GOAL, temperature decreasing is done after each m iterations, reheating is done $SA_{reheating}$ times, and there are several different neighborhoods ($N_k(s_0)$ is the *k-th* neighborhood of s_0) from which a solution can be drawn from. The Algorithm is outlined in Algorithm 3.1.

Iterated Local Search

Iterated local search (ILS) was first proposed in [LMS03] as a metaheuristic that perturbs and optimizes solutions in local optimum. In GOAL, the starting point of the algorithm is a solution generated by simulated annealing. Iteratively, $n_{perturb}$ perturbations are performed and a random-non-ascend method is then applied to lead the solution to a (possibly different) local optimum. The non-ascent algorithm, unlike simulated annealing, systematically enumerates neighborhoods solutions, with the order of exploration being changed randomly. It only accept solutions whose quality is at least the same as the current solution. A perturbation consists of the unconditional acceptance of a solution

Algorithm 3.1: Simulated Annealing for XHSTT

input: Initial solution $s_{initial}$, initial temperature $T_{initial}$, number of reheats $reheats_{max}$, number of iterations before cooling n_{max}, cooling rate α.
output: Optimized solution s_{best}.

```
1  begin
2      s_best ⟵ s_initial; s_cur ⟵ s_initial; n_iter ⟵ 0; T_cur ⟵ T_initial; reheats ⟵ 0
3      while reheats < reheats_max ∧ there is time left do
4          while n_iter < n_max do
5              n_iter ⟵ n_iter + 1
6              k ⟵ selectNeighborhood()
7              s_new ⟵ selectNeighbor(N_k(s_cur))
8              Δ ⟵ cost(s_new) − cost(s_cur)
9              if Δ < 0 ∨ e^(−Δ/T_cur) > random(0,1) then
10                 s_cur ⟵ s_new
11                 if cost(s_cur) < cost(s_best) then
12                     s_best ⟵ s_cur
13                 end
14             end
15         end
16         T_cur ⟵ α * T_cur
17         n_iter ⟵ 0
18         if T_cur < T_min then
19             reheats ⟵ reheats + 1
20             T_cur ⟵ T_initial
21         end
22     end
23     return s_best
24 end
```

from the KC or RTT neighborhoods (see next section). The number $n_{perturb}$ is set to an initial value and is increased if no improvements were found after a certain number of iterations. It is reset to its initial value when a improving solution is found or when its reaches a threshold value. The ILS algorithm used in GOAL is outlined in Algorithm 3.2.

Neighborhoods

The following neighborhoods were used in GOAL:

- Event Swap (ES): The times of two events are swapped.
- Event Move (EM): An event is moved from its current time to another.

Algorithm 3.2: Iterated Local Search for XHSTT

input: Initial solution $s_{initial}$, initial number of perturbations $n_{perturb}$, threshold value $n_{max}^{perturb}$.
output: Optimized solution s_{best}.

begin
  $s_{best} \longleftarrow descentPhase(s_{initial})$
  $n_{perturb} \longleftarrow n_{initial}$; $n_{iter} \longleftarrow 0$
  while *there is time left* **do**
    $n_{iter} \longleftarrow n_{iter} + 1$
    $s_{new} \longleftarrow s_{best}$
    for $j = 1..n_{perturb}$ **do**
      $s_{new} \longleftarrow selectNeighbor(N_{KC}(s_{new}) \cup N_{RRT}(s_{new}))$
    end
    $s_{new} \longleftarrow descentPhase(s_{new})$
    if $cost(s_{new}) < cost(s_{best})$ **then**
      $s_{best} \longleftarrow s_{new}$
      $n_{perturb} \longleftarrow n_{initial}$
      $n_{iter} \longleftarrow 0$
    end
    if $n_{iter} = n_{max}^{iter}$ **then**
      $n_{iter} \longleftarrow 0$
      $n_{perturb} \longleftarrow (n_{perturb} + n_{initial})$ **mod** $n_{max}^{perturb}$
    end
  end
  return s_{best}
end

- Event Block Move (EBM): Similar as ES, but if the move would cause the events to overlap, they are instead assigned times such that the second event starts immediately after the first one.
- Resource Swap (RS): The resources of two events are swapped. Only one resource per event is changed.
- Resource Move (RM): A resource of an event is replaced by another.
- Kempe Move (KM): Two times are chosen and a number of *chains* are considered. A chain is a series of event moves (EM), where two consecutive events share a common resource. The best chain with respect to the objective function is then selected as a neighbor.
- Reassign Resource Times (RRT): A set of events which all share a same common resource are selected and every possible permutation of their times among each

other is considered. The best permutation with respect to the objective function is chosen as a neighbor.

Note that the last last two neighborhoods are very computationally expensive when compared to the others. KM and RRT are used in the iterative local search algorithm for diversification purposes. KM is, in addition, used for optimization, but seldom (only 2% of the moves are KM). With this we conclude the description of GOAL and its main components.

3.1.2 Variable Neighborhood Search

The variable neighborhood search (VNS) method was proposed in [MH97]. It is similar to iterated local search, with the difference being in the way the solution is perturbed. A hierarchy of the neighborhoods is defined and one neighborhood is set as the active one. The active neighborhood is used to perturb the solution. Initially, the first neighborhood is set as the active neighborhood. In every iteration in which the descent phase did not result in a better solution than the best one found so far, the next neighborhood (according to the hierarchy) is set as the active neighborhood. When a better solution is found or when all of the neighborhoods have been used, the active neighborhood is reset to the the first one. The algorithm has been applied to XHSTT [FS14] and we give its outline in Algorithm 3.3. We refer to it as the Basic VNS (BVNS).

Algorithm 3.3: Basic Variable Neighborhood Search Algorithm for XHSTT

```
   input: Initial solution s_initial
   output: Optimized solution s_best
 1 begin
 2     while there is time left do
 3         k ⟵ 1
 4         while k ≤ k_max do
 5             s_new ⟵ selectNeighbor(N_k(s_best))
 6             s_new ⟵ descentMethod(s_new)
 7             if cost(s_new) < cost(s_best) then
 8                 s_best ⟵ s_new
 9                 k ⟵ 1
10             else
11                 k ⟵ k + 1
12             end
13         end
14     end
15     return s_best
16 end
```

For XHSTT, the neighborhoods are the same as in GOAL, but without the RRT neighborhood. Their hierarchy is in the order as listed previously. In addition to BVNS, three other variations have been implemented for XHSTT. We describe them in the context for XHSTT as follows:

- Reduced VNS (RVNS) - similar as BVNS, but without the descent phase.
- Sequential Variable Neighborhood Descent (SVND) - similar as BVNS with the restriction that only the *k-th* neighborhood is considered in the descent phase, rather than allowing all neighborhoods (k is the currently active neighborhood, the one that is used the perturb the solution).
- Skewed VNS (SNVS) - as SVND, but in addition to the restriction to the neighborhoods during the descent phase, a relaxed rule is used to accept the solution. The relaxed rule takes into consideration the cost of the solution, as well as the distance from the best solution. To calculate the distance between two solutions s_1 and s_2, they are first represented as strings of length n, where n is the number of events. The *i-th* element of the string contains information on the times and resources associated with the *i-th* event. The distance $\delta(s_1, s_2)$ is the hamming distance[1] of the two strings. When evaluating a solution s'' for acceptance, the formula $f(s'') - \delta(s'', s_{best})$ is used instead of just its cost $f(s'')$.

Overall, RVNS performs poorly when compared to GOAL and other VNS algorithms. Its performance is somewhat expected, as it is unlikely better solutions will be found by randomly going through neighborhoods without more sophisticated guidance. Another reason is that exploring large and expensive neighborhoods may slow down the search in cases when improvements can be found using simple and smaller neighborhoods.

The other three VNS algorithms, BVNS, SVND, and SNVS, all perform better than GOAL on average. Among the three, the results are very similar, although SVNS typically gives better results. This is attributed to the ability to easily escape local optima, as more diverse solutions are explored when compared to GOAL. Further improvements to VNS are made by hybridizing with integer programming (see Section 3.6.2).

However, it is worth noting that for *easy* instances, another VNS approach [BFT+12b] (not discussed here) performs better than the ones presented. The approach in [BFT+12b] uses Simulated Annealing instead of a descent method for VNS. The reason is that it is "easier" to find improvements for easy instances with SA than by the descent method, due to the diversification part of SA. According to the results report in [FS14], VNS-SA was better in 5 out of 19 instances.

[1] The hamming distance of two strings of the same size is the number of positions in which they have different elements.

3.1.3 Late Acceptance Hill Climbing

Late acceptance has first been proposed in [BBb]. In traditional hill climbing, a solution is accepted after a local search move if its objective value is strictly better (lower for minimization, greater for maximization) than the current solution. In late acceptance hill climbing, a parameter k is defined. When deciding whether to accept a solution, it is accepted if it has a better objective value than the *k-th* previous solution. Therefore, it can be seen as a generalization of hill climbing for $k > 1$. Late acceptance hill climbing provides a simple way of introducing diversification and extending hill climbing while retaining a simple acceptance criteria.

After a certain number of iterations, improving moves will naturally become less frequent. Therefore, all of the previous k solutions will be identical and the method degenerates into a standard hill climbing method. To combat this situation, the authors in [FSC16b] have introduced a variant called *stagnation-free* late acceptance hill climbing (sf-LAHC). With sf-LAHC, after no improvement has been made in the last $f(k)$ iterations, the list of previous solutions is reset to the value it had the last time a improvement was made. For simplicity the authors chose f to be a linear function. The algorithm is outlined in Algorithm 3.4.

When combined with simulated annealing (SA-sf-LAHC) and applied to XHSTT, the method outperformed GOAL (Section 3.1.1) by a large margin, but it is still not competitive when compared to the VNS approach (Section 3.1.2). Even though late acceptance is not the state-of-the-art from XHSTT, we believe it is still worthwhile to consider, as it is a relatively new and simple technique that demonstrated good performance. Therefore, we believe it is a promising approach for XHSTT.

3.2 Hyper Heuristics

Hyper-heuristics can be divided into two categories, depending on whether their focus is on *generating* or *selecting* heuristics. In the following we focus on the latter, as it has been successfully applied for XHSTT.

The general idea is to have two sets of heuristics: low-level and high-level. Low-level heuristics are problem specific and range from simple to more complicated local search moves. For timetabling, as examples of a low-level heuristic, we may consider the neighborhoods discussed in GOAL (Section 3.1.1), such as exchanging the time assignment for two events, or moving a single event from one time to another. These also include procedures which perturb the search space with the aim of diversifying the solution. High-level heuristics define strategies to make use of a large number of low-level heuristics (e.g. in [KK16], authors consider 15 different low-level heuristics). In addition, one may also consider several different acceptance strategies, such as the ones used in hill climbing or simulated annealing. Ideally, hyper-heuristics can be used for many different problems and are not domain specific.

Algorithm 3.4: Stagnation-Free Late Acceptance Hill Climbing for XHSTT

input: Initial solution $s_{initial}$, maximum number of iterations before reset $n_{iter}^{stagnation}$
output: Optimized solution s_{best}

begin
  $n \longleftarrow f(k)$
  // p = $\langle p_0, p_1, ..., p_(n-1)\rangle$
  $p_i \longleftarrow cost(s_{initial})\ \forall i \in \{0, 1, ..(n-1)\}$
  $p'_i \longleftarrow p_i$
  $s_{best} \longleftarrow s_{initial}$
  $s_{cur} \longleftarrow s_{initial}$
  $i \longleftarrow 0$
  while *there is time left* **do**
    $s_{new} \longleftarrow selectNeighbor(N(s_{cur}))$
    $v \longleftarrow i$ **mod** k
    if $cost(s_{new}) < p_v$ **then**
      $s_{cur} \longleftarrow s_{new}$
      if $cost(s_{new}) < cost(s_{best})$ **then**
        $s_{best} \longleftarrow s_{cur}$
        $p' \longleftarrow p$
        $i \longleftarrow 0$
      end
    end
    $p_v \longleftarrow cost(s_{cur})$
    $i \longleftarrow i + 1$
    if $i = n_{iter}^{stagnation}$ **then**
      $p \longleftarrow p'$
      $i \longleftarrow 0$
    end
  end
  return s_{best}
end

In [AÖK15], the authors used a hyper-heuristic approach with nine low-level heuristics. Iteratively, the high-level heuristics would select one of the low-level heuristics and apply it to the current solution. The selection of a particular heuristic depends on three criteria: previous performance, pair-wise dependency with the previously used heuristic, and the time passed since the last time it was used.

Hyper-heuristics based on hidden Markov chains have been studied in [KK16]. They used 15 different low-level heuristics. Iteratively, a *sequence* is selected and applied. A sequence is an ordered list of low-level heuristics and the heuristics are applied with respect to their ordering. This is one of the main differences from previous hyper-heuristics approaches (e.g. [KOP, AÖK15]) which can somewhat be seen as hyper-heuristics dealing with sequences of length one. In [AÖK15], this has partially been addressed by the pair-wise dependency measure between heuristics. The length of the chain and the low-level heuristics are selected stochastically, as given by the hidden Markov model. The probabilities are set to an initial value and are adjusted at run-time. Each time a sequence has been applied, its performance is recorded and the probabilities are modified accordingly. Therefore, better performing heuristics have a higher chance of being selected.

These approaches are very flexible for several reasons. Heuristics are adapted to the instances that is being solved, rather than being preselected, tuning the algorithm at run-time to the instance being solved. It also allows the consideration of sequences of heuristics, which is beneficial as certain heuristics only perform well when paired with other heuristics. This alleviates two problems. First, certain heuristics may be inefficient on some instances, but very efficient on others. Therefore, with this approach, the algorithm is more likely to spend most of its time on the most appropriate heuristics for the instance. Second, given its robust tuning mechanism, it removes the burden that the algorithm designer faces when handling low-level details.

3.3 KHE

In this section we describe local search algorithms implemented in and distributed with KHE, an open source software library. The main focus is to provide fast and robust algorithms that can produce quality solutions in low amount of time (a few minutes). Due to these desirable features and its availability as open source, they have been used in other XHSTT works as well (for example, see Sections 3.1 and 6.1.3).

We aim to discuss only the (arguably) most important parts of the algorithms and refer the interested reader for more details to [Kinb]. We describe generating initial solutions, as they are used in works described in Sections 3.1 and 3.6.2, and polymorphic ejection chains, as they are the main features of KHE14 [Kinb]. The algorithm KHE14 in a sense incorporates ejection chains in the initial solution generation method to provide high quality solutions. We now proceed to describe these two parts and refer to interested reader to [Kinb] for full details.

3.3.1 Initial Solution Generation

The initial solution generation is done in three phases, according to hierarchical timetabling [Kinc]: structural, time assignment, and resource assignment phase. As mentioned above, all of these phases can be improved with the use of polymorphic ejection chains (see next section).

The structural phase can be seen as a preparation phases for the next stage. Event are split into subevents, according to the *Split Events Constraint*. Subevents that share certain logical connections are grouped into *nodes*. The logical connections are influences by several factors: the original event from which the subevent is derived from, preassigned resources, and the relation of the subevent to other subevents with respect to spread and split events constraints. Subevents are assigned to other subevents if they are constrained by the link event events constraints. This ensures that the subevents will be scheduled in the same time, avoiding violations of the link events constraints. Assigning subevents to other subevents supports *hierarchical timetabling*, a methodology where smaller parts of the timetable are constructed and then incorporated into the complete timetable. During this phase, the set of possible times and resources are determined for each event, according to preferred times or resource constraints. Additionally, a certain level of *regularity* is enforced. While it does not directly contribute to the objective function, the author in [Kinb] claims that good solution are more likely to be found if regularity is maintained, although it is difficult to evaluate experimentally.

Times are assigned to subevents in the *times assignment phase*. A number of *layers* are created, which can be seen as a subset of subevents which share preassigned resources. A complex heuristic algorithm based on the maximum matching problem is solved to determine the time assignments for each layer. Afterwards, resources are assigned in the *resource assignment phase*. The algorithms in this phase are based on *bin packing*, constructive algorithms guided by a maximum matching algorithm, data structures to ensure the satisfaction of certain resource assignment constraints, and other techniques.

3.3.2 Polymorphic Ejection Chains

The idea is to analyze defects (constraint violations) and use specialized procedures to repair them depending on what kind of defect has been encountered. These procedures are called *augment functions*. This approach is different from the ones presented in Section 3.1, as constraint violations are handled by procedures tailored to individual constraints, rather than by attempting to perform a (random) move from one of the neighborhoods (e.g. as in the VNS approaches). Thus, this approach is more focused on specific problems in the solution and attempts to solve them one by one.

The repair procedure for a single defect is as follows. An augment function is called to repair the constraint violation. During this repair, a number of other defects can be introduced. If no other defects have been produced and the initial defect has been fixed, the procedure is successful and terminates. If one *significant* defect has appeared, its appropriate augment function is called, recursively repeating this procedure. A new

defect is said to be *significant* if its removal would reduce the objective function below the value before the repair procedure has been initiated. Otherwise, if two or more significant defects have been introduced, the procedure stops and the solution is reverted to before the repair attempt, hence the name *ejection chain*. In principle, all of the newly produced defects could be recursed on, forming an *ejection tree*, but this is done only in special cases, as it is likely to be unsuccessful.

Additionally, there are two ways that can be used to limit the size of the chain: imposing that the chain cannot be larger than a fixed constraint, or by constraining that entities cannot be visited more than once in the same chain.

The augment functions are designed for each constraint in XHSTT. They typically attempt to move the entity in question (event or resource) to every other possible time. The move can be a simple event move or kempe chain as described in Section 3.1.1 (with the additional option of *ejecting* other entities, that is, unassigning their time), or other constraint specific moves.

In this section our goal was to give the reader the main ideas concerning the KHE algorithms. For more details, we refer the interested reader to [Kinb] as the main KHE algorithm description, where additional references are given.

3.4 Decomposition

In this approach XHSTT is decomposed into two phases [Sør13a]: time and resource assignments. The motivation is that it might be simpler to solve each phase separately, rather than handling the whole problem at once. Information about the second phase is incorporated into the first phase to ensure that good solutions can be found using the decomposition. If certain conditions are met, the decomposition is exact, preserving optimality. Unfortunately, the decomposition method is not developed enough to compete with the state-of-the-art results. Nevertheless, for two instances it outperforms the pure IP approach, indicating that it is possibly a good research direction for future research. We note that similar approaches, where times are assigned before resources, have been effective for the KHE algorithms (see Section 3.3) and that a similar approach has been used for another related timetabling problem [SD14]. We briefly sketch the main ideas and refer the interested reader to [Sør13a] for more details.

In the first phase, times are assigned to events. The key element of the decomposition is the way additional constraints are inserted into the first phase to take into consideration the objective function of the second phase. Therefore, a "good" solution of the first phase is likely to transition into a "good" solution to the second phase. This is accomplished by considering a bipartite graph for each time, in addition to the subset of the constraints of the first phase. The bipartite graph has two sets of nodes: subevents and resources. Subevent nodes are linked with resource nodes if the corresponding resource can be assigned to the given subevent. Some of the resource constraints from XHSTT can be directly modeled by in the bipartite graph, while other not. Therefore, a solution to

the first phase is a lower bound to the original problem (with an exception, which we do not describe here). *Hall's Theorem* provides a sufficient and necessary condition for the bipartite graph to have a matching for all subevent nodes to resource nodes. An assignment of resources to subevents corresponds to a matching. In the other direction, a matching *resembles* an assignment. The reason why it only *resembles* an assignment is because not all constraints of the problem are taken into account. (Soft) Constraints are modeled to guide the solution towards one which satisfies the conditions of Hall's theorem. Lastly, the graph's contribution to the objective function is modeled by adding its lower bound value via the theorem on the *lower bound for minimum weight maximum matching* [SD14]. It is important to note that the complete bipartite graph is not directly modeled, but only the parts necessary to capture the modeling related to the two theorems. This concludes the first phase, which is allocated 95% of the computational time. After a solution is found to the first phase, it is passed to the simpler second phase, which performs resource assignments considering all of the constraints with respect to the already assigned times. A solution to the original problem can be built based on the solutions of the two phases. Under certain post-conditions the decomposition guarantees to preserve the optimal solution.

3.5 Integer Programming

In [KSS15], the authors model XHSTT as an integer program and then use *Gurobi*, a commercial IP solver, to compute solutions for XHSTT instances. The solving is done in two phases. In the first phase only hard constraints are considered. After the optimal solution has been computed for the first phase (note that it might include hard constraint violations), soft constraints are added and the cost of hard constraints is fixed to the previously calculated optimal value. This captures the lexicographical objective function of XHSTT, where hard constraints are significantly more important than soft ones. The integer programming approach is exact (or complete), typically requiring longer computational times (e.g. days), but can provide good results along with lower bounds and proofs of optimality.

The basic variables for the IP model $x_{se,t,er,r}$ are Boolean, representing whether subevent se is taking place at time t while using resource r for its event resource[2] er. The formulation create every possible subevent for an event and the appropriate $x_{se,t,er,r}$ variables are defined. For example, given an event with duration four, subevents with the following durations are generated: 1, 1, 1, 1, 2, 2, 3, and 4. Boolean variables u_{se} indicate whether a subevent se is *active* or not, meaning whether any times have been assigned to it. Additional constraints are imposed to ensure that only the correct combinations of subevents may be used. The XHSTT constraints, along with other variables, are defined by using these variables as basic building blocks. A solution to the instance is an assignment of values to the $x_{se,t,er,r}$ variables. We direct the interested reader to [KSS15] for full details about the modeling.

[2]event resources correspond to the *role resource requirement*, described in Section 2.

3.6 Hybrid Approaches

Metaheuristic and complete approaches both have distinct advantages and disadvantages. Metaheuristics can produce good solution in low amount of time, but only consider a subset of the search space. Complete (or exact) algorithms, on the other extreme, search through the complete search space and are guaranteed to find the best possible solution, but are likely to take significantly more time. Therefore, a natural idea is to combine the two extremes into something that can offer the best of both worlds. This is what *hybrid* algorithm aim for.

All of the algorithms presented in this category share a common theme: fix the values of a subset of variables and use integer programming to optimize for the remaining variables. This process is done iteratively. The variables are fixed with respect to a *neighborhood* and the best solution found so far. In this case, a neighborhood typically consists of a number of elements (e.g. two teachers) and all variables that are not related to those elements are fixed according to their values in the best solution found so far, leaving the complete algorithm to determine the best assignments for the remaining variables. Afterwards, based on the obtained solution, parameters are adjusted (e.g. the neighborhood size is increased). The algorithms differ in the way the high level heuristic chooses to fix the subset of variables (the neighborhood generation). An outline is given in Algorithm 3.5.

Algorithm 3.5: Generic Hybrid Algorithm for XHSTT

input: Initial solution $s_{initial}$, problem formulation P, variables X
output: Optimized solution s_{best}

```
1  begin
2      s_best ⟵ s_initial
3      while there is time left do
4          n ⟵ generateNeighborhood()
5          X' ⟵ fixVariables(s_best, n)
6          s_new ⟵ solveFixed(P, X')
7          if cost(s_new) < cost(s_best) then
8              s_best ⟵ s_new
9          end
10         adjustParameters()
11     end
12     return s_best
13 end
```

3.6.1 Matheuristic

This approach is an adaptive large neighborhood search (ALNS) algorithm, whose structure resembles Lectio [SKS], the second placed algorithm from ITC 2011. It has

been experimentally shown that better results can be obtained than with a pure integer programming approach, if one considers shorter running times. An initial solution is constructed according to a greedy algorithm. In each iteration, a neighborhood is chosen. The neighborhood choice is biased towards neighborhoods that have previously performed well. The neighborhood is used to determine which variables are to be fixed. The IP solver then optimizes the solution with the remaining unfixed variable. Afterwards, the neighborhood size for the next iteration is adjusted according to the IP gap calculated: if the IP gap is smaller (greater) than g_{min} (g_{max}) the size of the neighborhood is increased (decreased).

There are four different neighborhood types to chose from, each applying to a certain set of elements (events, resources, times, or time groups). Once a neighborhood type is selected, a neighborhood is computed using in a semi-randomized procedure which is as follows. Initially the neighborhood consists of a randomly selected element (e.g. an event selected at random). New elements are added from the remaining set of elements iteratively, one by one, based on a measure of their relatedness to the previously selected elements for the neighborhood. The size of the neighborhood determines the number of elements to be chosen. Once a neighborhood S is computed, all the values of decision values *not related* to S are fixed with respect to the currently best solution found so far, and the IP solver is called to optimize with the remaining variables. The relatedness measure depends on the element in question.

- Event: let $R(e)$ be the set of resources that could possibly be assigned to event e. The relatedness of a particular event e and the set of previously selected events E is given as $|R(e) \cap (\cup_{j \in E} R(j))|$.
- Resource: similar as for events (above), except $E(r)$ is considered, where $E(r)$ is the set of events to which resource r could possibly be assigned to.
- Time: only the next time (chronologically speaking) is considered related. Therefore, if n is the size of the neighborhood, n consecutive times are selected in the end.
- Time Group: the number of times shared with the previously selected time groups.

To conclude for this algorithm describe, we note that it is important to establish a notion of relatedness between the decision variables that are selected for the neighborhood, as this makes it more likely that a better solution may be found.

3.6.2 Variable Neighborhood Search and Integer Programming

Further improvements to the VNS approach [FS14] presented in Section 3.1.2 are done by hybridizing the approach with IP. Therefore, the algorithm relies on the IP model for XHSTT [KSS15], presented in Section 3.5.

One tenth of the computational time is allocated to the VNS algorithm and the rest is improved by a matheuristic using IP. The best solution obtained by the VNS algorithm is

used as a starting point. The matheuristic randomly selects n resources as its neighborhood. The value of n changes during the algorithm execution, depending on how many times the algorithm managed to compute local optimality (optimality with respect to the selected neighborhood) in the previous iterations. The assignments of variables related to the resources in neighborhood are set free, while the others are fixed with respect to the best solution found so far, and an IP solver is used to calculate the local optimum. This is repeated until a timeout is reached.

3.6.3 Fix-and-Optimize

The fix-and-optimize approach is a large neighborhood search algorithm that is based on integer programming. It is applied to the class-teacher timetabling problem with compactness requirements problem, which when translated into XHSTT corresponds to the Brazilian instances (see [ITC]). Therefore, it deals with a subset of XHSTT. Nevertheless, we mention it in this section as the algorithm was able to produce good results for the instances it considers. The approach shares similarities with the previously mentioned hybrid approaches, but differs in a number of ways. Its initial solution is generated using an IP solver by generating a solution that satisfied all of the hard constraints (soft constraints are ignored). It then goes through different neighborhoods, allocating a fixed amount of time for each neighborhood. It iteratively considers neighborhoods of k classes until no improvements can be made, and then does the same with k teachers. Initially the parameter k is set to two and increases each time all options for that parameter are exhausted and no progress has been made. It is interesting to note that neighborhoods which considered days where deemed less efficient than the teacher or class neighborhoods in [DdAB14], but have proven to be very effective in our large neighborhood search algorithm (see Chapter 6).

3.6.4 Summary

We discussed various approaches that have shown to be effective on existing benchmarks for XHSTT. Nevertheless, more efficient computation of solutions and optimality proofs in terms of lower bounds are still active research topics. In the following chapters we show work in this direction by introducing new paradigms and methods for XHSTT, based on propositional logic and related approaches. Each chapter represents a distinct approach to XHSTT.

CHAPTER 4

Modeling High School Timetabling as Partial Weighted maxSAT

Existing algorithms for XHSTT are mostly based on heuristic (incomplete) techniques, as discussed in Chapter 3, with the only exception being the integer programming approach [KSS15]. These incomplete algorithms aim to provide upper bounds to the problem by searching through only a limited part of the search space. In this chapter, we consider a new complete (or exact) approach which, contrary to the aforementioned heuristic algorithms, exhaustively explores the whole search space.

In general, the first step in devising a complete algorithm is to precisely capture the problem definition using some mathematical formalism. We model the complex formalism of XHSTT using Boolean variables and basic logical connectives only. Hard constraints are translated into propositional Boolean formulas (SAT). To account for the soft constraints, the model is extended with the use of Partial Weighted maxSAT. By using our modeling, any solution of cost c for the Partial Weighted maxSAT formula can be directly translated into a XHSTT solution with cost c.

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

To experimentally evaluate our approach, we took all relevant XHSTT-instances (see Section 4.3.1) and out of the pool of 39 instances we were able to model 27 of them. The remaining 12 instances were not modeled because our developed maxSAT formulation in general does not support resource assignment. We have a specific modeling for resource

assignments (Assign Resource Constraints and related constraints) for two instances, but our current model is not practical for the other remaining instances.

We empirically evaluate different cardinality constraint modelings and solvers, in order to determine the best modeling for XHSTT. We show that competitive results can be obtained by modeling XHSTT as Partial Weighted maxSAT. Our method is compared to the state-of-the-art complete approach based on integer programming, and computational results demonstrate that we outperform IP on many of the used benchmarks. We are able to obtain even better results by combining several maxSAT solvers. Furthermore, we have experimented with an SMT approach, in which soft constraints are gradually added throughout the search.

It is important to note that the competing IP approach relies on using Gurobi, a highly engineered piece of software, while we use publicly available (and in some cases open source) maxSAT solvers that are not as heavily engineered and still provide good results. In addition, the maxSAT model presented here plays a crucial role in the our large neighborhood search algorithm (see Chapter 6).

It is worth mentioning that in [AN14] 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 constraints, which typically restrict the number of idle times between lessons that a teacher may have, are extremely important constraints in XHSTT and cannot be modeled in CTT. Many CTT constraints are special cases of XHSTT ones or can be adapted for XHSTT. All of the constraints in CTT but one can be modeled by XHSTT. The translation of CTT into XHSTT has been studied in [FSCS]. Due to this fact, new and more generalized encodings must be explored in order to model XHSTT.

The rest of the chapter is organized as follows: a brief description of the satisfiability problem (SAT) and its extension maxSAT, together with our maxSAT modeling of XHSTT is given in Section 4.1. Afterwards, we describe our SMT approach in Section 4.2. Detailed experimental results are given in Section 4.3, where several important questions are answered. Finally, in Section 4.4 we summarize the results of this chapter and conclude.

4.1 Modeling XHSTT as maxSAT

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

We use two approaches for solving XHSTT. The first one is to model all constraints as maxSAT and give the resulting formula to a maxSAT solver to calculate a solution. The second approach is an iterative one in which we start with a relaxed version of the original problem by omitting all soft constraints and iteratively add them during the

solution process. This second approach corresponds to an SMT approach described in Section 4.2. As is shown in Section 4.3, the first approach is more successful than the second one.

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

4.1.1 SAT and maxSAT

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

The extension of SAT considered in this work is Partial Weighted maxSAT, in which clauses are partitioned into two types: hard and soft clauses. Each soft clause is given a weight. The goal is to find an assignment which satisfies the hard clauses and minimizes the sum of the weights of the unsatisfied soft clauses. For more information about SAT and maxSAT, the interested reader is referred to [BHvMW09].

4.1.2 Cardinality Constraints

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

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

$(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$ and $(x_1, x_2, x_3, x_4) = (1, 1, 1, 1)$ would incur a penalty of 1 and 2.

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

Combinatorial Encoding

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

Bit Adders

The idea is to regard each literal as a 1-bit number and 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 result is stored in auxiliary variables a_{22} and a_{21}, which represent the final sum as a binary number. 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 does not 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 [Sin] 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 [Sin] is closely related to unary numbers. The unary number representation for an integer n as given in [BBa] is:

$$\bigwedge_{i \in [1,n-1]} (u_i \Rightarrow u_{i-1}). \tag{4.1}$$

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

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

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

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

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

$$\bigwedge_{\substack{i \in [0..n) \\ j \in [0..,kmax) \\ (i+1 \geq k_{max} \vee j \leq i)}} (S_{i,j} \Rightarrow S_{(i+1),j}) \tag{4.2}$$

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:

$$\bigwedge_{\substack{i\in[1..n)\\ j\in[1..,kmax)\\ (i+1\geq k_{max}\vee j\leq i)}} (x_i \wedge S_{(i-1),(j-1)} \Rightarrow S_{1,j}) \tag{4.3}$$

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):

$$\bigwedge_{i\in[0..n)} (x_i \Rightarrow S_{i,0}) \tag{4.4}$$

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

$$\bigwedge_{\substack{i\in[0..n)\\ j\in[0..,kmax-1)\\ (i+1\geq k_{max}\vee j\leq i)}} (\overline{S_{i,j}} \Rightarrow \overline{S_{(i+1),(j+1)}}) \tag{4.5}$$

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:

$$\bigwedge_{j\in[0..,kmax)} (\overline{S_{i,j}} \Rightarrow x_i) \tag{4.6}$$

General cases:

$$\bigwedge_{\substack{i\in[0..n-1)\\ j\in[0..,kmax)\\ (i+1\geq k_{max}\vee j\leq i)}} (\overline{S_{i,j}} \wedge S_{(i+1),j} \Rightarrow x_{i+1}) \tag{4.7}$$

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 4.6, 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 4.5. 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:

$$(S_{n-1,k_{min}-1}) \tag{4.8}$$

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

$$\bigwedge_{i\in[k_{max}-1..n)} (\overline{x_i} \vee \overline{S}_{(i-1),(k_{max}-1)}) \tag{4.9}$$

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 4.9.

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

Cardinality Networks

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

4.1.3 Soft Cardinality Constraints

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

Combinatorial Encoding

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

$$\bigwedge_{j \in P} (A_j \rightarrow atLeast\_j[x_i : x_i \in X]) \wedge \bigwedge_{j \in P} (w(j)(A_j)) \tag{4.10}$$

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 [Sin] 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 $atMost\_k_{max}$ (4.8 and 4.9). 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 $atMost\_k_{max}$):

$$\bigwedge_{i \in [1..k_{min})} (w_i(S_{(n-1),(i-1)})) \tag{4.11}$$

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:

$$(w_0(S_{(n-1),0})) \tag{4.12}$$

$$\bigwedge_{i \in [1..k_{min}]} (w_i(\overline{S}_{(n-1),(i-1)} \vee S_{(n-1),i})) \tag{4.13}$$

Cardinality Networks

Cardinality networks [ANOR] can be used to model soft constraints and this has been done in [AN14]. However, when doing so, since every output must be penalized[1], they require more auxiliary variables and degenerate into Sorting Networks [ES06] from which they offer improvements, meaning the number of auxiliary variables and clauses goes up to $O(nlog^2n)$.

4.1.4 Special Cases For Cardinality Constraints

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

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

$$\bigwedge_{x_i \in X} ((c)(x_i)) \tag{4.14}$$

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

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

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

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

4.1.5 XHSTT constraints as maxSAT

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

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

As noted in Section 2.2.3, each constraint applies to a subset of events, resources, times, and other XHSTT entities. These will be denoted by the index *spec*, e.g. E_{spec}, T_{spec}, R_{spec}. We now give the modeling of constraints described in 2 as Partial Weighted maxSAT.

Assign Time Constraints

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

$$\bigwedge_{e \in E} (exactly\_d[Y_{e,t} : t \in T]) \tag{4.15}$$

Avoid Clashes Constraint

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

$$\bigwedge_{\substack{e \in E \\ t \in T \\ r \in R}} (Y_{e,t,r} \Rightarrow Y_{e,t}) \tag{4.16}$$

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

$$\bigwedge_{\substack{r \in R_{spec} \\ t \in T}} (at\_Most\_1[Y_{e,t,r} : e \in E(r)]) \tag{4.17}$$

Avoid Unavailable Times Constraints

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

$$\bigwedge_{\substack{r \in R \\ t \in T}} (X_{t,r} \Leftrightarrow \bigvee_{e \in E(r)} Y_{e,t,r}) \tag{4.18}$$

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

$$\bigwedge_{r \in R_{spec}} (atMost\_0[X_{t,r} : t \in T_{spec}]) \tag{4.19}$$

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

Split Events Constraints

In the formal specification of XHSTT, any time can be defined as a starting time as events can be split into multiple subevents. One could regard a starting point as a time t where a lecture takes place, but has not took place at $t-1$. However, while this is true, this is not the only case when a time would be regarded as a starting time, since e.g. time $t = 5$ and $t = 6$ might be interpreted as *last time of Monday* and *first time*

of Tuesday and an event could be scheduled on both of these times, but we may regard both times as starting times. It is also worthy to note that we can also regard that as a double (block) lecture, even though it spans over two days (this was the case in the previous version of the Brazilian instances). Such a double lecture is not the ideal double lecture, but is still better than splitting the lecture into two lectures and assigning them in another fashion. Therefore, any time can in general be regarded as a starting time. Other constraints give more control over these kind of assignments.

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

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

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T}} (Y_{e,t} \wedge \overline{Y}_{e,(t-1)} \Rightarrow S_{e,t}) \tag{4.20}$$

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

$$\bigwedge_{\substack{e \in E \\ t \in T}} (S_{e,t} \Rightarrow Y_{e,t}) \tag{4.21}$$

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

$$\bigwedge_{e \in E_{spec}} (atLeast\_A_{min}[S_{e,t} : t \in T] \wedge atMost\_A_{max}[S_{e,t} : t \in T]) \tag{4.22}$$

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

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

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T}} (S_{e,t} \Rightarrow \bigvee_{d_{min} \leq d \leq d_{max}} K_{e,t,d}) \tag{4.23}$$

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

[2]Remark: We could have encoded that exactly one of the right hand sides literals must be chosen, but this is handled in the later parts of this encoding in Equation 4.25.

$$\bigwedge_{\substack{e\in E_{spec}\\ t\in T\\ d\in D}} K_{e,t,d} \Rightarrow \bigwedge_{i\in[0,d-1]} Y_{e,(t+i)} \tag{4.24}$$

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

$$\bigwedge_{\substack{e\in E_{spec}\\ t\in T\\ d\in D}} (K_{e,t,d} \Rightarrow \bigwedge_{d_{min}\leq g\leq d_{max}} \bigwedge_{\substack{i\in[0,d-1]\\ g\neq d\wedge i\neq 0}} \overline{K}_{e,t+i,g}) \tag{4.25}$$

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:

$$\bigwedge_{\substack{e\in E_{spec}\\ d_{min}\leq d\leq d_{max}\\ t\in T\wedge t+d<|T|}} K_{e,t,d} \wedge Y_{e,t+d} \Rightarrow S_{e,t+d} \tag{4.26}$$

Prefer Times Constraints

The constraint is encoded as:

$$\bigwedge_{e\in E_{spec}} (atMost\_0[\star : t \in T \setminus T_{spec}]) \tag{4.27}$$

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

Distribute Split Events Constraint

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

$$\bigwedge_{e\in E_{spec}} (atLeast\_A_{min}[K_{e,t,d} : t \in T]) \tag{4.28}$$

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

$$\bigwedge_{e\in E_{spec}} (atMost\_A_{max}[K_{e,t,d} : t \in T]) \tag{4.29}$$

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

Spread Events Constraints

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

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

$$\bigwedge_{\substack{eg \in EG_{spec} \\ t \in T}} (Z_{eg,t} \Leftrightarrow \bigvee_{e \in eg} S_{e,t}) \tag{4.30}$$

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

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atLeast\_d_i^{min}[Z_{eg,t} : t \in tg_i]) \tag{4.31}$$

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

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atMost\_d_i^{max}[Z_{eg,t} : t \in tg_i]) \tag{4.32}$$

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

Limit Busy Times Constraints

We define the auxiliary variables $B_{tg,r}$, which indicate whether resouce r is busy within time group tg. A resource is busy at a time group tg iff it is busy in at least one of the times of tg. Let TG_{spec} denote the specified set of time groups. We define the variables $B_{tg,r}$ as:

$$\bigwedge_{\substack{r \in R \\ tg \in TG_{spec}}} (B_{tg,r} \Leftrightarrow \bigvee_{t \in tg} X_{t,r}) \tag{4.33}$$

If a resource r is busy during a time group, it must be busy for at least b_{min} and at most b_{max} times during that time group. The constraint is encoded as:

$$\bigwedge_{\substack{tg \in TG_{spec} \\ r \in R_{spec}}} (B_{tg,r} \Rightarrow atLeast\_b_{min}[X_{t,r} : t \in tg] \wedge atMost\_b_{max}[X_{t,r} : t \in tg]) \quad (4.34)$$

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

Cluster Busy Times Constraints

Recall the auxiliary variables $B_{tg,r}$ defined in the previous constraint. Specified resources must have at least b_{tg}^{min} busy time groups:

$$\bigwedge_{r \in R_{spec}} (atLeast\_b_{tg}^{min}[B_{tg,r} : tg \in TG_{spec}]) \quad (4.35)$$

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

$$\bigwedge_{r \in R_{spec}} (atMost\_b_{max}^{tg}[B_{tg,r} : tg \in TG_{spec}]) \quad (4.36)$$

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

Limit Idle Times Constraints

To encode the constraint, three different types of auxiliary variables are used.

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

$$\bigwedge_{\substack{tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (G_{t,r}^{tg} \Leftrightarrow \bigvee_{i<t \ \wedge \ i \in tg} X_{i,r}) \quad (4.37)$$

$$\bigwedge_{\substack{tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (H_{t,r}^{tg} \Leftrightarrow \bigvee_{i>t \ \wedge \ i \in tg} X_{i,r}) \quad (4.38)$$

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

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

$$\bigwedge_{\substack{tg \in TG_{spec} \\ t \in tg \\ r \in R_{spec}}} (I_{t,r}^{tg} \Leftrightarrow \overline{X}_{t,r} \wedge G_{t,r}^{tg} \wedge H_{t,r}^{tg}) \tag{4.39}$$

We now encode the constraint:

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

$$\bigwedge_{\substack{tg \in TG_{spec} \\ r \in R}} (atLeast\_idle_{min}[I_{t,r}^{tg} : t \in tg]) \tag{4.40}$$

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

$$\bigwedge_{\substack{tg \in TG_{spec} \\ r \in R}} (atMost\_idle_{max}[I_{t,r}^{tg} : t \in tg]) \tag{4.41}$$

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

Order Events Constraints

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

$$\bigwedge_{\substack{(e_1,e_2) \in E_{spec}^2 \\ t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{i \in [0,t]} \overline{Y}_{e_2,i}) \tag{4.42}$$

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

$$\bigwedge_{\substack{(e_1,e_2) \in E_{spec}^2 \\ t \in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{j \in [t+1, t+B_{min}]} \overline{Y}_{e_2,j}) \tag{4.43}$$

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

$$\bigwedge_{\substack{(e_1,e_2)\in E^2_{spec}\\ t\in T_{spec}}} (Y_{e_1,t} \Rightarrow \bigwedge_{k\in[t+1,t+B_{max}+1]} Y_{e_2,k}) \tag{4.44}$$

Link Events Constraints

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

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

$$\bigwedge_{\substack{eg\in EG_{spec}\\ t\in T\\ e_j,e_k\in eg}} (Y_{e_j,t} \Leftrightarrow Y_{e_k,t}) \tag{4.45}$$

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

Preassign Resource Constraints

We define decision variables which indicate whether an event is using a resource at a time. If an event is using a resource at some time, the event must take place at that time ($R_{spec}(e)$ is the set of resources preassigned for event e):

$$\bigwedge_{\substack{e\in E_{spec}\\ t\in T\\ r\in R_{spec}(e)}} (Y_{e,t} \Rightarrow Y_{e,t,r}) \tag{4.46}$$

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

Preassign Time Constraints

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

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

Assign Resource Constraints

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

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

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

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T}} (Y_{e,t} \rightarrow exactly\_1[M^{role}_{e,t,r} : r \in R_{spec\_resource\_type}]) \tag{4.47}$$

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:

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T \\ r \in R_{spec\_resource\_type}}} (M^{role}_{e,t,r} \rightarrow Y_{e,t,r}) \tag{4.48}$$

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

Avoid Split Assignments Constraint

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

$$\bigwedge_{\substack{e \in E_{spec} \\ R_{spec\_resource\_type} \\ t \in T}} (M^{role}_{e,t,r} \rightarrow V^{role}_{e,r}) \tag{4.49}$$

The constraint is now encoded as:

$$\bigwedge_{e \in E_{spec}} (atMost\_1[V^{role}_{e,r} : r \in R_{spec\_resource\_type}]) \tag{4.50}$$

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

Prefer Resources Constraints

Similar to before, this constraint applies for a specified *role*. The encoding relies on auxiliary variables created in assign resource constraints:

$$\bigwedge_{\substack{e \in E_{spec} \\ t \in T}} (atLeast\_1[M^{role}_{e,t,r} : r \in R \setminus R_{spec}]) \tag{4.51}$$

Limit Workload Constraints

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

$$Workload(sr) = Dur(subevent) * Workload(subevent)/Dur(event) \tag{4.52}$$

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

$$\bigwedge_{r \in R_{spec}} (atLeast\_Work_{min}[X_{t,r} : t \in T] \wedge atLeast\_Work_{max}[X_{t,r} : t \in T]) \tag{4.53}$$

Special Cases For Constraints

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:

$$\bigwedge_{t \in T} (atMost\_h[Y_{e,t} : e \in E_{spec}]) \tag{4.54}$$

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

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

If the resources specified in Assign Resource Constraints are not subjected to Limit Idle Constraints and assigning more than one resource to an event may be feasible, then a simpler encoding may be used for ARC, in which $atLeast\_1$ is used instead of using $exactly\_1$. This case happens typically in instances which require the assignment of rooms. If two rooms are assigned to an event, in the solution we would simply pick only one. However, this cannot be applied in general e.g to teachers.

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

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

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

$$K_{e,t,d} \leftrightarrow (\bigwedge_{i \in [0..d-1]} Y_{e,t+i}) \wedge \overline{Y}_{e,t+d} \tag{4.55}$$

In preliminary experiments this encoding showed to be effective for some instances instead of the general encoding of $K_{e,t,d}$. If this encoding is used, other constraints might be affected, such as Split Events Constraint and need to be changed accordingly. In order

to comply with the requirements of XHSTT, we have continued further with the general encoding of $K_{e,t,d}$ (as given in Section 4.1.5 under Split Events Constraints) and have not performed detailed experiments for this special case.

4.1.6 Additional Remark

We recommend keeping the weights for soft clauses as low as possible. For example, if we would multiply the weights of the constraints in XHSTT by 100, this would not change the set of feasible solutions, but may have a negative impact on the performance of maxSAT solvers. However, this depends on the maxSAT technique used. For example, a linear upper bounding algorithm for maxSAT would see a significant drop in performance because very large cardinality constraints would be generated, but core-guided are likely to see no difference.

4.2 SMT approach

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

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

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

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

4.2.1 Technical details

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

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

We use the the Linear maxSAT algorithm (Algorithm 6.1) [LBP10] which makes repeated calls to a SAT solver and after each call adds constraints which ask for a better solution than the previous one. The optimal solution is obtained when the SAT solver returns $false$. We opted to use the linear algorithm as it was one of the algorithms that had good performance for XHSTT (see Section 4.3).

Algorithm 4.1: Linear Algorithm for maxSAT

```
1  begin
2      P ⟵ maxSAT formula
3      c = ∞
4      bestAssignment = ∅
5      while isSatisfiable(P) do
6          bestAssignment = satisfiableAssignment(P)
7          c ⟵ cost(P, bestAssignment)
8          P = P ∪ (∑_{i∈K} softConstraint(i) < c)
9      end
10 end
```

The original maxSAT formula is changed because bounds are added at each iteration, in addition to *learned* clauses which are added to direct the search (see [SLM09] for clause learning). It is not straightforward to remove the added clauses at later stages of the algorithm, because clauses are learned with respect to other clauses (including other learned clauses) and removing some clauses may therefore invalidate previously learned clauses. To the best of our knowledge, no maxSAT solver supports this kind of search. An alternative is to restart the solver after each call, losing possibly valuable learned clauses and bounds. This motivated us to investigate a different approach: instead of restarting between calls, we keep the modified formula intact. Thus, each call to the solver depends on all previous calls due to the bounds and learned clauses. When querying the solver with a new set of assumptions, it will attempt to report the best possible solution, but only if it is better than all of the previously computed solutions. To this end, we modified the linear algorithm in the open-source maxSAT solver Open-WBO [MML]. A different approach related to ours is presented in [MJML] for lower bounding maxSAT algorithms.

In our SMT approach, each time a new optimal solution with respect to the currently considered soft constraints is found it is examined and new soft constraints are inserted, meaning that the sum of the weights of soft constraints changes and the previously inserted clauses regarding the cardinality constraints (third line in the *while* loop of 6.1.2) should be invalidated. In order to invalidate them, we performed the following: when inserting the clauses, we add them as usually, but add a negative literal $\bar{b}$ to each clause. During the current iteration, this literal is treated as an assumption which assigns it $false$ ($b = true$). When the cardinality constraints inserted need to be invalidated, this is simply done by inserting a hard unit clause ($\bar{b}$), which forces the assignment of $b = false$, making all the clauses added as cardinality constraints for the previous iteration satisfiable (effectively invalidating them).

We summarized the above in Algorithm 4.2 and 4.3. With ub and lb we denote the upper and lower bound respectively, with $maxSATcost$ the cost of the solution with respect to the maxSAT problem currently analyzed, while $XHSTTcost$ is the cost with respect to all constraints of the problem. Note that when a solution of cost c is found for a relaxed version of a XHSTT-instance, c is a lower bound for the original XHSTT problem.

Algorithm 4.2: SMT for XHSTT Algorithm Outline

```
1  begin
2      I_relax ⟵ encodeHardConstraints(I_original)
3      ub = ∞
4      lb = 0
5      bestAssignment = ∅
6      globally_solved = false
7      while globally_solved = false do
8          v ⟵ createNewVar()
9          a = modifiedMaxSATsolve(I_relax, v)
10         cla = encodeViolationsNotEncoded(a)
11         if lb < maxSATcost(a) then
12             lb = cost(a)
13         end
14         if ub > XHSTTcost(a) then
15             ub = XHSTTcost(a)
16         end
17         if cla = ∅ ∨ lb = ub then
18             globally_solved = true
19         else
20             I_relax = I_relax ∪ cla
21         end
22         I_relax = I_relax ∪ (v)
23     end
24 end
```

Algorithm 4.3: Modified Linear Algorithm for SMT (using v as an input variable)

```
1  begin
2      v ⟵ input variable given as parameter
3      P ⟵ maxSAT formula
4      c = ∞
5      bestAssignment = ∅
6      while isSatisfiableUnderAssumption(P, v̄) do
7          bestAssignment = satisfiableAssignmentUnderAssumption(P, v̄)
8          c ⟵ cost(P, bestAssignment)
9          cla = encodeAsClauses(Σ_{i∈K} softConstraint(i) < c)
10         i = 0
11         while i < |cla| do
12             cla[i] = cla[i] ∨ v
13             i++
14         end
15         P = P ∪ cla
16     end
17 end
```

4.3 Computational Results

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

- Compare the performance of different maxSAT solvers on XHSTT-instances.
- Compare different cardinality constraint encodings for XHSTT.
- Compare maxSAT with Integer Programming for XHSTT.
- Compare maxSAT with the SMT approach (Section 4.2).
- Investigate how well our maxSAT approach would do if it was used in the International Timetabling Competition 2011.

4.3.1 Benchmark instances and Computing Environment

We evaluated our approach on XHSTT benchmark instances which can be found in the repository of the International Timetabling Competition 2011 (ITC 2011) [3]. We used the XHSTT-2014 benchmark set, which contains instances that were careful selected by the ITC 2011 over the years and are meant to be interesting test beds for researchers.

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

Additionally, we included every instance used in the competition (these two sets of instances overlap). This way we took into consideration all relevant XHSTT-instances, to the best of our knowledge.

In total we can model efficiently with maxSAT 27 out of 39 (70%) instances. We have a specific modeling for resource assignments (Assign Resource Constraints and related constraints) for two cases (FinArtificialSchool, EnglandStPaul, see Section 4.1.5), but for other instances with resource assignments our current model is not practical. Unfortunately, in this case the number of produced variables and clauses is very large, and until now we could not come up with a more efficient encoding for these constraints. Thus, for the remaining 12 instances, we currently do not have an appropriate model and could not have experimented with them.

In the instances, the number of times ranges from 25 to 125, number of resources from 8 to 99, number of events from 21 to 809 with total event duration from 75 to 1912. These numbers vary heavily from instance to instance. We direct the interested reader to [PAD$^+$12, PKA$^+$14] for more details regarding the instances.

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

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

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

ITC 2011 issued a benchmark tool which is designed to test how fast a machine performs operations relevant for timetabling. The tool estimates how long a XHSTT solver should run on the machine at hand in order for it to be equivalent to 1000 seconds on ITC's computer (the computational time limit for ITC's second phase of the competition). The intent is to provide grounds for determining some normalized time across different platforms. The suggested times for our maxSAT benchmark server and the IP computer were 1250 and 690 seconds, respectively. Given that both solvers are exact solvers, we decided to allocated roughly ten times more time, for a total of four hours and 2.2 hours.

4.3.2 Notation

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

4.3.3 Solvers

We chose to experiment with maxSAT solvers WPM3 [ADG], Open-WBO [MML], and Optiriss (a combination of the Riss framework [KKMS15] and Open-WBO [MML]). The first two solvers were selected because they were the best solvers for timetabling instances in the Industrial Weighted Partial maxSAT category in the maxSAT Competition 2016, and since Open-WBO was used in Optiriss we decided to include it as well.

Both Optiriss and Open-WBO allows its users to configure the solvers by selecting among several maxSAT algorithms and parameters. We used the default configuration for Open-WBO and the two configurations of Optiriss (*Optiriss-def* and *Optiriss-inc*) that were used in the maxSAT Competition 2016. In addition to this, we used the same configuration but have set the solvers to use the Linear maxSAT algorithm [LBP10] (see Algorithm 6.1) because this algorithm already previously showed good performance for XHSTT in [DM] (this can be done by adding the parameter *-algorithm=1* to either solver). Therefore, if we consider different configuration of solvers as stand alone solvers themselves, we experimented with a total of seven solvers.

4.3.4 Evaluation of different maxSAT solvers

We compare the performance of different maxSAT solvers on XHSTT-instances. In order to do so, we used a similar ranking system as the ITC 2011. We run all solvers on each instance for four hours and record the solution. For each instance we compute the rank for each solver. The rank is a number between one and seven and it represents how well the solver did relative to other solvers. For a given instance, the best solver has rank one, the second best has rank two, etc. Solvers can share the same rank in case of ties. In Table 4.1 and Table 4.2 we show the results and the ranking of solvers for this comparison.

Based on these results, we conclude that the default configuration of Open-WBO has the best average rank. However, the average rank of 2.16 indicates that there is no clear winner as the results are not uniform across instances. While overall Open-WBO performs the best on average, we can see that for a number of instances it is outperformed by other solvers. Therefore, we decided to select k solvers with complement each other instead of determining a single winner. In other words, we wish to select k solvers such that if we run them in parallel and take the best result, we obtain good results across all benchmarks. Formally, we would like select k solvers in order to minimize the combined

rank $\frac{\sum_{i \in I} min(rank(s,i):s \in S)}{|I|}$, where I is the set of instances, S is the set of selected solvers with $|S| = k$, and $rank(i, s)$ is the rank of solver s on instance i.

In order to determine which solver to chose and how many (the parameter k) we modeled the described problem as a maxSAT optimization problem and used Open-WBO to compute the optimum solution for every k. We show the optimum combined rank for every choice of k as a pair (k, rank): (1, 2.2), (2, 1.4), (3, 1.2), (4, 1.1), (5, 1), (6, 1), (7, 1). Based on these results, we chose $k = 4$ in order to keep the combined rank close to one. One solution for $k = 4$ is Open-WBO (def), Open-WBO (lin), Optiriss (inc), and Optiriss (default-linear). These four solvers are the best maxSAT solvers for XHSTT according to our criteria. Each of them have their own strengths and weaknesses and they will be used for further experimentation. We note that there are other combinations of four solvers which achieve the same combined rank, but we have arbitrarily chosen this combination among these ones.

4.3.5 Evaluation of (Soft) Cardinality constraint encodings

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

We denote the encoding configuration used for an instance as a pair X-Y: X is used for Assign Time Constraints, and Y for other constraints. In the case of the combinatorial encoding, the bit adder encoding would be used instead in situations when the encoding would produce too many clauses and variables ($n \geq 50 \vee (n \geq 42 \wedge k \geq 5)$, where n is the number of literals and k is the cardinality of the constraint. The selected encodings are used to encode both hard and soft cardinality constraints. We experimented with four different encoding configurations (abbreviations: CN - cardinality networks, C - combinatorial, and S - sequential): S-C, CN-C, S-S, and CN-CN. The last two encodings can further be abbreviated with simply S and CN, respectively. The first configuration was initially submitted to the maxSAT competition 2014 and was used in Section 4.3.4.

We run the best solvers (determined in Section 4.3.4) with the same time limit of four hours on each instance with each encoding configuration. We consider each pair of solver and encoding configuration as a single solver and rank them for each instance as in Section 4.3.4. We present the results and rankings in Table 4.3 and 4.4.

As in Section 4.3.4 we wish to select the k pairs of solvers and encoding configurations which complement each other the most. We show the optimum combined rank for every choice of k as a pair (k, rank): (1, 4.23), (2, 2.52), (3, 1.92), (4, 1.6), (5, 1.3), (6, 1.2),

(7, 1.12), (8, 1.04), ($n \geq 9$, 1). Based on these results, we chose $k = 4$ as before. One solution for $k = 4$ is Open-WBO (lin) and (def) with CN, Open-WBO (def) with CN-C, and Optiriss (incremental) with SS. These pairs are the best combinations of maxSAT solvers and cardinality constraint encodings for XHSTT according to our criteria.

4.3.6 Evaluation of a maxSAT approach versus an Integer Programming approach

We compare our maxSAT approach with an existing Integer Programming approach [KSS15]. For comparison purposes we used the best k pairs of solvers and encoding configurations determined in Section 4.3.5. These comparisons are performed as in previous section. The results and rankings are given in Table 4.5 and 4.6. We included the comparison with the combined maxSAT solutions as well.

Based on the results, we conclude that maxSAT is competitive with IP for the instances that we could model with our maxSAT approach. In particular, when comparing the maxSAT configurations individually with IP, two of them (Optiriss(inc)-S and Open-WBO-CN) achieve a better average ranking than IP. When we consider the combined rank, maxSAT outperforms IP in all but five cases.

An interesting point for maxSAT which we would like to emphasize is that maxSAT solvers are constantly being developed, are in some cases open source (e.g. Open-WBO), and are not so heavily engineered as the commercial IP solver Gurobi in [KSS15]. Nevertheless, maxSAT still manages to provide competitive results.

4.3.7 Evaluation of a pure maxSAT approach versus an SMT approach

We have implemented the SMT approach described in Section 4.2 by modifying the maxSAT solver openWBO. The implementation was done for a subset of instances. In this section, we give a comparison of results of this SMT approach with a pure maxSAT approach using the same maxSAT solver. We compared with Open-WBO(lin)-S-C as it was the closest maxSAT formulation to our SMT approach described in Section 4.2. We have run the solvers for four hours and the results obtained are given in Tables 4.7 and 4.8. Overall the pure maxSAT approach shows better results, although the average ranks do not differ by a large amount.

Based on the experiments, we conclude that the SMT approach is competitive with the pure maxSAT approach. The results indicate that it might be worthwhile to further develop the SMT approach.

4.3.8 Evaluation of maxSAT approach on ITC 2011

In this section we compare with the results obtained during the second phase of the International Timetabling Competition 2011. During this round the time limit was set

to 1000 seconds. We run our approach with a normalized amount of time (see Section 4.3.1 and show the results in Tables 4.9 and 4.10.

Our approach provides competitive results with the heuristics solvers used in the competition. Any individual maxSAT configuration would rank second. If we would consider the combined rank of maxSAT solver, then a clear first place would be achieved. However, in the comparison we have only included instances which we were able to model with maxSAT (see Section 4.3.1), leaving out five instances.

4.4 Summary

We have demonstrated that XHSTT can be modeled as Weighted Partial maxSAT despite the fact that XHSTT is very general and has many different constraints, including both hard and soft ones. All constraints are included in their general formulations, with the exception of resource assignment constraints. Important alternative encodings for special cases were discussed, which included a special case of resource assignments as well.

We investigated empirically the performance of our model using 27 out of 39 instances from the Third International Timetabling Competition 2011 benchmark repository. We compared different maxSAT solvers and cardinality constraint encodings to determine the most appropriate combinations, compared to the state-of-the-art integer programming approach, and showed how well our maxSAT approach would perform if it was submitted to the International Timetabling Competition 2011. In addition, we evaluated a maxSAT-based SMT approach.

Based on the results we conclude that our developed maxSAT approach is competitive for XHSTT, outperforming the state-of-the-art complete approach based on integer programming on many benchmarks. Experimenting with different cardinality constraints and maxSAT solvers proved to be important. Out of all of the tested combinations, we were able to isolate a few, which not only performed well, but successfully complemented each other. When compared to the state-of-the-art integer programming approach, the results are in favor for maxSAT, with the solver open-WBO and the cardinality network encoding performing better on most benchmarks. After combining several solvers, significantly better results were achieved, due to the complementary nature of the solvers and encodings. When comparing to ITC 2011, our approach would rank first or second, depending on whether or not we combine different maxSAT solvers. It is important to note that all other ITC solvers are based on metaheuristics and are expected to perform better in such large problems given limited computational time. The mentioned comparisons with integer programming and ITC's competitors considered only those instances that we can model with maxSAT. The pure maxSAT approach was overall more efficient than our developed SMT approach, although the obtained results indicated that further research in the SMT direction might be fruitful.

Lastly, our generated maxSAT instances based on XHSTT problems were submitted to the maxSAT Competition 2014 and have been used since. They have proven to be challenging benchmarks for maxSAT solvers.

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

Table 4.1: Comparison of maxSAT solvers.

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

Table 4.2: Ranking of maxSAT solvers.

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

Table 4.3: Comparison of selected solvers with different cardinality constraints. Abbreviations: α = Open-WBO(lin), β = Open-WBO(def), θ = Optiriss(inc), γ = Optiriss(inc-lin)

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

Table 4.4: Ranking of selected solvers with different cardinality constraints. Abbreviations: α = Open-WBO(lin), β = Open-WBO(def), θ = Optiriss(inc), γ = Optiriss(inc-lin)

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

Table 4.5: Comparison of maxSAT solvers with integer programming.

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

Table 4.6: Ranking of maxSAT solvers and integer programming.

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

Table 4.7: Comparison of maxSAT and the developed SMT approach (Section 4.2)

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

Table 4.8: Ranking of maxSAT and the developed SMT approach (Section 4.2)

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

Table 4.9: Comparison of maxSAT solvers in the ITC 2011 second round.

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

Table 4.10: Ranking of maxSAT solvers in the ITC 2011 second round.

CHAPTER 5

Modeling High School Timetabling with Bitvectors

In this chapter, we describe our new modeling approach for high school timetabling using bitvectors in which constraints are modeled as a series of bitvector operations. The resulting model allows efficient computation of constraint costs for local search algorithms. Additionally, it can be used to solve XHSTT with satisfiability modulo theory (SMT) solvers that support bitvector theory. We evaluate the performance for our bitvector modeling approach and compare it to the leading engine KHE for local search algorithms such as hill climbing and simulated annealing. The experimental results show that our approach can provide speed-ups for this problem. Furthermore, experimental results using SMT are given on instances from the ITC 2011 benchmark repository.

When developing HSTT algorithms, modeling aspects are very important from a practical point of view, as a good model will allow efficient implementations of the HSTT algorithms. However, for a complex problem such as general HSTT, finding good models is a challenging task because of the presence of a large number of different constraints. Therefore, the problem is two-fold: one must research good algorithmic strategies, while also having efficient data structures or models which will allow fast implementations. This is what our bitvector model aims to solve. One of the key points of the bitvector modeling is that modern processors have built-in support for bitvector operations, allowing us to efficiently compute constraint violations. Additionally, most modern processors have special operations for determining the number of bits set in an integer. These operations are called population counts or hamming weight instructions and are extensively used in the model. All of this, together with the compactness of the model, contribute to the overall effectiveness of the modeling approach.

The rest of the chapter is organized as follows. In the main Section 5.1, we describe the modeling of XHSTT as bitvectors. In Section 5.2, we present computational results. Lastly, a summary of the chapter is given in Section 5.3.

5.1 Modeling XHSTT with Bitvectors

In this section we propose a bitvector modeling for XHSTT. The main idea is to provide a simple modeling approach that can be used in different solving techniques. All of the constraint costs are obtained by using bitvector operations. We first introduce basic bitvector definitions and operations used. Then, we define the variables used for modeling XHSTT with bitvectors, followed by the description of XHSTT constraints with bitvectors.

5.1.1 Basic Bitvector Definitions

A bitvector is a vector of bits. The size of the vector is arbitrary, but fixed. Standard operations (e.g. *addition*, *and*, *or* operations on bitvectors) and predicates (e.g. equality) are defined over bitvectors and an instance consists of a conjunction of predicates. We use prefix notation, which is common for most SMT solvers, with the addition of brackets and comas in order to ease reading. For example, in infix notation one would write $(a = b)$, while in prefix notation the same expression would be written as $(= a\ b)$, while we choose to write $(= (a, b))$.

Most operations are interpreted as usual and all bitvector operands are of the same length. In the following we present some of the notations we use, in which bv_a and bv_b are bitvectors and k is a constant integer:

- $inv(bv_a)$ – the bits of bv_a are inverted (e.g. $inv(1011001) = 0100110$).
- $add(bv_a, bv_b)$ – two bitvectors are added in the same way two unsigned integers would be added (overflow might occur).
- $or(bv_a, bv_b)$ – bitwise *or* is performed on the operands. When applying *or* to each bitvector bv_a from some set BV, we use the following notation: $\bigvee_{bv_a \in BV}(bv_a)$
- $lshift(bv_a, k)$ – noncylic left shift by k is applied (e.g. $lshift(10011, 2) = 01100$).
- $rshift(bv_a, k)$ – similar to $lshift$, but uses right shifting.
- $extract(bv_a, k)$ – the *k-th* bit of bv_a is extracted.

Variables

For each event e (e.g. a lesson), we create a number of bitvectors all of length n, where n is the number of times available in the instance. The vectors along with their meanings are as follows:

- Y_e – the *i-th* bit is set (a bit is *set* if it has value 1) if the event is taking place at time i and is not set otherwise. In XHSTT terminology, Y_e covers all subevents of event e. This implies that two subevents of the same event can never clash in this representation.

- S_e – the *i-th* bit is set if the *i-th* time is declared as a starting time for event e and is not set otherwise.

- $K_{(e,d)}$ – the *i-th* bit is set if the *i-th* time is declared as a starting time of duration d for event e and is not set otherwise.

As an example of the above variables, take the following bitvectors:

7	6	5	4	3	2	1	0	(*time slot*)
0	1	1	0	0	1	1	0	(Y_e)
0	1	1	0	0	0	1	0	(S_e)
0	1	1	0	0	0	0	0	$(K_{(e,1)})$
0	0	0	0	0	0	1	0	$(K_{(e,2)})$

(5.1)

From Y_e, we see that event e (e.g. a Math lesson) is taking place at time 1, 2, 5, and 6, because those bits are set within Y_e. Similarly, times 1, 5, and 6 are labeled as starting times from S_e, meaning event e has been split into three subevents. Time 1 is labeled as a double lesson by $K_{(e,2)}$, while times 5 and 6 as lessons of duration 1 by $K_{(e,1)}$. Note that time 5 could have also been labeled as a double lesson instead of having two lessons of duration 1. Reasons for choosing one possibility over the other is regulated by constraints.

In the formal specification of XHSTT, any time can be defined as a starting time because 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 taken place at $t - 1$. However, while this is true, this cannot be the only case when a time would be regarded as a starting time, since e.g. time $t = 5$ and $t = 6$ might be interpreted as *last time of Monday* and *first time of Tuesday* and an event could be scheduled at both of these times, but clearly we must regard both times as starting times, since a double lecture does not extend over such long periods of time. Therefore, any time can in general be regarded as a starting time. It is of interest to note that the previous assignment, by the general formulation, could also be treated as a double lesson for the purpose of constraints, even though it extends over two days. Constraints give more control over these kind of assignments.

Note that our model, in order to capture the complete search space for the problem, must account for all possible combinations of the number of subevents for each event. For example, an event of duration 3 can be split into three different ways: one subevent of duration three, two subevents of durations one and two, or three subevents of duration one. Therefore, we cannot assign a bitvector for each subevent in advance because we do not know before hand in how many subevents will a particular event be split into. Due

to this we must take into account all possibilities. The equations model all these possible combinations of (nonclashing) subevents.

Formalities that are tied to starting times with regard to the specification are expressed as follows:

If a starting time for event e has been assigned at time t, then the corresponding event must also take place at that time (the set E is the set of all events):

$$\bigwedge_{e \in E} (= (or(S_e, Y_e), Y_e)) \tag{5.2}$$

When modeling with bitvectors it is common to have formulas of the form $(= (bv_a, \textit{some logical expressio}$ like the one above. This ensures that the bitvector bv_a is equal to the specified logical expression. In Equation (5.2), we encode that Y_e is equal to $(or(S_e, Y_e))$, meaning that there are no bits set in S_e which are not also set in Y_e, but it can be that some bits in Y_e are set which are not set in S_e. This is the behavior we want to capture, because if some times are declared to be starting times (the bits set in S_e), then surely the event in question must take place at those times (hence asserting the bits set in Y_e), but since they can last longer than one time it can be the case that Y_e has bits set in position where S_e does not.

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

$$\bigwedge_{e \in E} (= (or(and(Y_e, lshift(inv(Y_e), 1)), S_e), S_e)) \tag{5.3}$$

Note that the ordering of the application of inv and $lshift$ is important. With the application of $exp_1 = (and(Y_e, lshift(inv(Y_e))))$, we will obtain a bitvector which has its *i-th* bit set iff Y_e has its *i-th* bit set and its *(i-1)-th* bit is not set. Then, similarily to Equation (5.2), with the application of $= (or(exp_1, S_e), S_e)$ we ensure that S_e has bits set at least in every position as in bitvector exp_1, which is what we want to capture: every time we have the situation that a (sub)event is taking place at time i, but has not taken place at time $(i-1)$, we declare that time a starting time for said event (note that other times can be starting times too).

If time t has been set as a starting time, associate a duration with it ($D(e)$ is the set of durations that subevent of event e can take):

$$\bigwedge_{e \in E} (= ((\bigvee_{d \in D(e)} K_{(e,d)}), S_e)) \tag{5.4}$$

By setting a bit in position i in S_e we ensure that at least one $K_{(e,d)}$ will have an *i-th* bit set. Later on through Equation (5.8) we ensure that exactly one $K_{(e,d)}$ will have such bit set.

If a subevent of event e of duration d has been assigned a starting time at time t and event e is also taking place at time $t + d$, then assign time $t + d$ as a starting time ($D(e)$ is the set of possible durations subevents of e might take):

$$\bigwedge_{\substack{e \in E \\ d \in D(e)}} (= (or(and(lshift(K_{(e,d)}, d), Y_e), S_e), S_e)) \tag{5.5}$$

The formula $exp = and(lshift(K_{(e,d)}, d), Y_e)$ will result in a bitvector which has its *i-th* bit set if event e is taking place at time i and event e has been declared to have a starting point at *i-d* time of duration d. In other words, event e started at *i-d* and was declared to last d times, but after d times event e is still taking place. Therefore, we want to ensure that event e will also have a starting point at time *i*. This is then done in a similar fashion to before: $(= (or(exp, S_e), S_e))$.

When a bit in $K_{(e,d)}$ is set, ensure that the event in question must take place for d consecutive times during this specified time. In order to do this, we define a helper bitvector Y_e^d which will have its *i-th* bit set if starting from time i event e has d consecutive bits set. For example, if $Y_e = (0, 0, 1, 1, 1, 0)$, then $Y_e^3 = (0, 0, 0, 0, 1, 0)$ and $Y_e^2 = (0, 0, 0, 1, 1, 0)$ (recall that the right most bit represents time 0). Bitvector Y_e^d can be computed by taking the *and* of all of $rshift(Y_e, k)$ for $k = 0..(d - 1)$ (with $rshift(Y_e, 0) = Y_e$). We now proceed with the constraint encoding:

$$\bigwedge_{\substack{e \in E \\ d \in D(e)}} (= (or(K_{(e,d)}, and(Y_e^d, K_{(e,d)})), K_{(e,d)})) \tag{5.6}$$

The expression $exp = and(Y_e^d, K_{(e,d)})$ is a bitvector which has its *i-th* bit set if event e has d consecutive bits set starting from time i and has a starting time of duration d at time i. In order to ensure that when a bit in $K_{(e,d)}$ is set there must be d consecutive bits set in Y_e starting from time i, we encode: $(= (or(exp, K_{(e,d)}), K_{(e,d)}))$.

If an event e has a subevent of duration d starting at time i (the *i-th* bit set in $K_{(e,d)}$), make sure that no other starting time can be set within the duration of that subevent. In order to do this, we define a helper bitvector $K_{(e,d)}^k$ as:

$$\bigwedge_{\substack{e \in E \\ d \in D}} (= (\bigwedge_{i=0..k} (inv(rshift(K_{(e,d)}, i))), K_{(e,d)}^k)) \tag{5.7}$$

Bitvector $K_{(e,d)}^k$ will have its *i-th* bit set if there is no bit set at time i nor in any of the next k times in $K_{(e,d)}$. We use this helper bitvector to encode the constraint:

$$\bigwedge_{\substack{e \in E \\ d_1 \in D}} (= (and(K_{(e,d_1)}, \bigwedge_{\substack{d_2 \in D \\ d_1 \neq d_2}} (K_{(e,d_2)}^{(d_1-1)})), K_{(e,d_1)})) \tag{5.8}$$

With $exp = \bigwedge_{d_2 \in D}(K_{(e,d_2)}^{d_1-1})$ we compute a bitvector which has its *i-th* bitvector set if there is no bit set at time i in any of the $K_{(e,d_2)}$ (with $d_1 \neq d_2$) nor in any of the next $d-1$ times. Therefore, only in $K_{(e,d_1)}$ can bits in these times be set. Then, $(= (K_{(e,d_1)}, and(K_{(e,d_1)}, exp))$ ensures that if $K_{(e,d_1)}$ has a bit set at time i, it must be the case that no other $K_{(e,d_2)}$ (with $d_1 \neq d_2$) has its bit at i nor in the next $k-1$ times.

With this constraint, we conclude constraints regarding starting time definitions. We now proceed with cardinality constraint encodings followed by high school timetabling constraint encodings.

5.1.2 Cardinality Constraint Encodings

An important constraint that arises often is to determine the number of set bits in a bitvector, as well as to impose penalties if the appropriate number of bits are not set. E.g. if an event must take place for two hours, then exactly two bits in its Y_e must be set.

Let us define a unary operation $reduceBit(bv_a) = bv_a \wedge sub(bv_a, 1)$. When applied to bv_a, as the name suggests, it produces a new bitvector which has one less bit set then bv_a (for the special case $bv_a = 0$, it returns 0). For example:

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & 0 & (bv_a) \\ \wedge & 1 & 1 & 0 & 0 & 1 & 1 & (sub(bv_a, 1)) \\ \hline & 1 & 1 & 0 & 0 & 0 & 0 & (reduceBit(bv_a)) \end{array} \tag{5.9}$$

The original bitvector had three bits set, while the produced one has two bits set. The *reduceBit* operations is an important part for defining cardinality constraints.

In order to ensure that *at least k* bits are set in a bitvector, we apply *reduceBit* $k-1$ times and require that the resulting bitvector must be different from zero. For *at most k*, we apply *reduceBit* k times and constrain that the resulting bitvector must be equal to zero. For *exactly k* we encode *at least k* and *at most k*. For example, asserting that *at least* 3 bits are set is done in the following way:

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & 0 & (bv_a) \\ \wedge & 1 & 1 & 0 & 0 & 1 & 1 & (sub(bv_a, 1)) \\ \hline & 1 & 1 & 0 & 0 & 0 & 0 & (reduceBit(bv_a)) \\ \wedge & 1 & 0 & 1 & 1 & 1 & 1 & (sub(reduceBit(bv_a)), 1) \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & (reduceBit(reduceBit(bv_a))) \end{array} \tag{5.10}$$

Since the final bitvector, which we have obtained by applying reduceBit twice, is different from the zero bitvector we conclude that *at least* 3 bits are set in bv_a.

It is important to note that when using the modeling for local search, bitvectors can be implemented using binary integers and standard binary operations over bits can be used. Additionally, most modern processors have special operations for determining the number of bits set in an integer. These operations are called *population counts* or

hamming weight instructions. We recommend using them if possible as they are more efficient than repeatedly applying the defined reduce operation when implementing local search algorithms with bitvectors.

Soft Cardinality Constraints

A similar technique to the one previously described is used for soft cardinality constraints. For *at least* k, it is asserted before each application of *reduceBit* and after the last application of *reduceBit* that the current bitvector is different from zero and is penalized by some weight if it is not the case. For example, asserting that *at least* 2 bits are set is done in the following way for the soft version:

$$\begin{array}{cccccccc} & 0 & 1 & 0 & 0 & 0 & 0 & (bv_a \neq 0 \rightarrow \textit{no penalty}) \\ \wedge & 0 & 0 & 1 & 1 & 1 & 1 & (sub(bv_a, 1)) \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & (reduceBit(bv_a) = 0 \rightarrow \textit{assign penalty}) \end{array} \tag{5.11}$$

Note that we checked for penalties in two cases (for the initial bitvector bv_a and $reduceBit(bv_a)$), but only one case was penalized in this particular case.

For *at most* k, a similar algorithm is used: *reduceBit* is applied k times as in the regular cardinality constraint version and then *bitReduce* is applied $n - k$ times to this bitvector (n is the size of bv_a) and before each application it is asserted that the current bitvector is zero and is penalized by some weight if it is not the case. Note that if we have some hard constraint limiting the maximum number of bits that may be set in a bitvector to some k_{max}, we do not perform the second part of the algorithm $n - k$ times, but rather just $k_{max} - k$ times. This is used frequently while modeling for SMT.

The penalty weights depend on the cost function chosen and this is discussed in the next section.

Cost Functions

The way the penalty weights are assigned depends on the constraint that is being modeled. Following XHSTT, we use three different penalty schemes: Linear, Quadratic, and Step. The Linear scheme penalizes linearly to its violation, the Quadratic scheme penalizes by squaring it, and the Step scheme assigns a penalty of one if there is a violation (regardless of how severe) and zero otherwise. These values are then multiplied by a weight w which is given in the constraint that is being modeled.

For the example used in Equation (5.11), the linear scheme assigns a penalty of w to each violation, the quadratic one would assign w to the first and $3 * w$ to the second, while step would assign w and 0.

5.1.3 Constraints

Each constraint has a set of points of application and each point generates a deviation. The cost of the constraint is obtained by applying a cost function on each deviations,

multiplying it by a weight, and then summing up all these values. There are three different cost functions, as discussed in Section 5.1.2.

When modeling XHSTT as SMT, we simplify the objective function by not tracking the infeasibility value, rather regarding it as zero or nonzero. By doing so we simplify the computation for the SMT solver, possibly offering faster execution times. However, when using the bitvector modelings for implementing local search algorithms, both hard and soft costs are tracked.

E, T and R are sets of events, times and resources, respectively. Each constraint is applied to some subset of those three, denoted by E_{spec}, T_{spec} and R_{spec}. These subsets are naturally 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.

We present encodings used in the experimental results, in which we assume that all resources are already assigned to events. We make this assumption as this eases the modeling and readability of the constraints. Later on we provide a description on how this limitation can be overcome.

Unless explicitly stated, soft constraints are implemented by using soft instead of hard cardinality constraints for the key equations which encode the limitations enforced by the constraint. In cases when this differs, we provide an explanation.

Assign Time Constraints

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

Each event's Y_e vector must have exactly d bits set, where d is the duration of the event:

$$\bigwedge_{e \in E_{spec}} (exactly\_d[Y_e]) \tag{5.12}$$

If the constraint is specified as soft, then instead of the equation above we would use the soft cardinality encoding for $atLeast\_d$ and a hard cardinality constraint $atMost\_d$ with Y_e. Points of applications are events and the deviation for each event is calculated as the number of times not assigned to the event.

Split Events Constraints

This constraint has two parts. The first part limits the number of starting times an event may have in the solution. The second part limits the duration of the event for a single subevent.

For example, if four times must be assigned to a Mathematics lecture, we may limit that the minimum and maximum duration of a subevent is equal to 2, thus ensuring

that the lecture will take place as two blocks of two hours, forbidding having the lecture performed as one block of four hours.

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

$$\bigwedge_{e \in E_{spec}} (atLeast\_A_{min}[S_e] \wedge atMost\_A_{max}[S_e]) \tag{5.13}$$

In addition, this constraint also imposes the minimum d_{min} and maximum d_{max} duration for each subevent:

$$\bigwedge_{\substack{e \in E_{spec} \\ d \in \{i | i < d_{min} \vee i > d_{max}\}}} (atMost\_0[K_{(e,d)}]) \tag{5.14}$$

Distribute Split Events Constraint

This constraint specifies the minimum d_{min} and maximum d_{max} number of starting times of a specified duration d. For example, if $duration(e) = 10$, we may impose that the lecture should be split so that at least two starting times must have duration three. The constraint is encoded as follows:

$$\bigwedge_{e \in E_{spec}} (atLeast\_d_{min}[K_{(e,d)}] \wedge atMost\_d_{max}[K_{(e,d)}]) \tag{5.15}$$

Prefer Times Constraints

This constraint specifies that certain events should begin at certain times. If an optional parameter d is given, then this constraint only applies to subevents with duration d. For example, a lesson of *duration 2* must be scheduled on Monday, excluding the last time on Monday.

Let P_e be the bitvector in which the *i-th* bit is set iff i is a preferred time. We then encode:

$$\bigwedge_{e \in E_{spec}} (atMost\_0[and(\star, inv(P_e))]) \tag{5.16}$$

where $\star$ is either S_e or $K_{(e,d)}$, depending on whether the optional parameter d is given.

If the constraint is required to be soft and the optional parameter d is not given, then the following formula is used instead (D_e is the set of duration event e can be subdivided into):

$$\bigwedge_{e \in E_{spec}} \bigwedge_{k \in D_e} (k * atMost\_0[and(K_{(e,k)}, inv(P_e))]) \tag{5.17}$$

If the optional parameter d is given, then instead of D_e we would use the singleton $\{d\}$. The k in front of $atMost\_0$ represents that when calculating the weights for violating the constraint, one must consider the deviation k times larger than normally (the constraint penalizes misplaced (sub)events of longer duration more).

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.

An event group eg is a set of events. Depending on these events, we propose two encodings for this constraint. The first encoding is simpler, but requires that the events in the specified event group cannot share any times. Formally, we require that:

$$\bigwedge_{eg \in EG_{spec}} \bigwedge_{\substack{(e_i, e_j) \in eg^2 \\ e_i \neq e_j}} (= (and(Y_{e_i}, Y_{e_j}), 0)) \tag{5.18}$$

The previous equation holds in all of the instances considered in our work because events in the event groups share a common resource and Avoid Clash Constraints prevents them from having shared times. Therefore, we use the the simpler encoding for modeling. We now proceed with this description and give the general case afterwards.

Let Z_{eg} be a bitvector which has its *i-th* bit set iff an event $e \in eg$ has a starting time at time i. This is obtained by applying *or* to all of the appropriate S_e vectors.

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 events from a given event group must have within times of that time group. Let TG_{spec} denote this set of sets of times and let $mask_{tg}$ be the bitvector which has its *i-th* bit set iff i is a time of time group tg. We define helper bitvectors $C_{(tg,eg)}$:

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (= (C_{(tg,eg)}, and(Z_{eg}, mask_{tg}))) \tag{5.19}$$

This constraint specifies the minimum d_i^{min} and maximum d_i^{max} amount of starting times within a given time group tg_i:

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atLeast\_d_i^{min}[C_{(tg_i,eg)}] \wedge atMost\_d_i^{max}[C_{(tg_i,eg)}]) \tag{5.20}$$

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead. Points of application are event groups (not events) and deviations are calculated as the number of set bits by which $C_{(tg_i,eg)}$ falls short of the minimum or exceeds the maximum.

As discussed previously, the provided encoding holds only if Equation (5.18) holds. Otherwise, the encoding given above will not be correct, because Z_{eg} does not account for more than one starting time at any time. Therefore, for each time t we would need to count how many starting times (from the events in the event group) take place at that time t. This can be done by using a helper bitvector $Q_{(tg,eg)}$ defined as:

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (= (Q_{(tg_i,eg)}, \bigvee_{e_j \in eg} (lshift(extendBV(and(S_e, mask_{tg_i}), |T| * j), |T| * j)))) \tag{5.21}$$

Here the indicies i and j represent the position of a time group or event within its time group ($i = 0..(|tg_i| - 1)$ or event group $j = 0..(|eg| - 1)$). The function $extendBV(bv_a, n)$ extends the bitvector bv_a to the size of n by adding the appropriate number of zeros to the end of bv_a. We use this function because otherwise $lshift$ would remove all information about the starting times due to the length of S_e (which is equal to $|T|$). The resulting bitvector $Q_{(tg,eg)}$ is of size $|eg| * |T|$ (number of events in eg multiplied by the number of times in tg). The inner *and* operation ensures that only bits related to tg_i are taken into consideration and the $lshift$ operations places the bits related to tg of the events from eg one after the other in $Q_{(tg,eg)}$. We can now encode the constraint:

$$\bigwedge_{\substack{tg_i \in TG_{spec} \\ eg \in EG_{spec}}} (atLeast\_d_i^{min}[Q_{(tg_i,eg)}] \wedge atMost\_d_i^{max}[Q_{(tg_i,eg)}]) \tag{5.22}$$

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 and Z_{eg} be a bitvector which has its *i-th* bit set iff an event $e \in eg$ is taking place at time i.

We define a helper bitvector L_{eg} whose *i-th* bit is set iff at time i at least one event is taking place but not all the events of the specified event group:

$$\bigwedge_{eg \in EG_{spec}} (= (L_{eg}, \bigvee_{e_i \in eg} and(Z_{eg}, inv(Y_{e_i})))) \tag{5.23}$$

The constraint is now encoded as:

$$\bigwedge_{eg \in EG_{spec}} (atMost\_0[L_{eg}]) \tag{5.24}$$

Order Events Constraints

This constraint specifies pairs of events and constrains that there must be a certain number of times in between the last time of the first event and the first time of the second event. Parameters B_{min} and B_{max} are given which define the minimum and maximum time separations between two events and are by default set to zero and the number of times, respectively. The constraint specifies a set of pairs of events to which it applies.

In order to encode this constraint, we define helper bitvectors with the aim of tracking the distance between the two events. The first type of helper bitvectors we define are MAX_e and MIN_e, which have its $i-th$ bit set iff event e is taking place at time i but not in any time after or before i, respectively:

$$\bigwedge_{e \in E_{spec}} (= (MAX_e, and(Y_e, inv(G_{(e,T)})) \wedge (= (MIN_e, and(Y_e, inv(H_{(e,T)}))) \tag{5.25}$$

Both MAX_e and MIN_e have exactly one bit set. In the above equation T is the set of all times, and $G_{(e,T)}$ and $H_{(e,T)}$ are as defined in Section 5.1.3 but with Y_e being used instead of X_r.

The next helper bitvector is $MAX_e^{'}$, which has the same bit set as MAX_e but also all bits to the right of it. Similar for $MIN_e^{'}$ except all bits from the left are set. For example, if $MAX_{e_i} = (0,0,0,1,0)$ and $MIN_{e_j} = (0,1,0,0,0)$ then $MAX_{e_i}^{'} = (0,0,0,1,1)$ and $MIN_{e_j}^{'} = (1,1,0,0,0)$. This is done for $MAX_e^{'}$ by taking the *or* of all bitvectors $rshift(MAX_e, i)$ with $i = 0..(|T|-1)$. For $MIN_e^{'}$, $lshift(MIN_e, i)$ is used instead.

We now define a helper bitvector for a pair of events $SEP_{(e_i,e_j)}$, which has its $i-th$ bit set iff time i is between the last time of e_i and the first time of e_j:

$$\bigwedge_{(e_i,e_j) \in E^2_{spec}} (= (inv(or(MIN_{e_j}^{'}, MAX_{e_i}^{'})), SEP_{(e_i,e_j)})) \tag{5.26}$$

Since $MAX_{e_i}^{'}$ has all bits set until the last time of e_i, and $MIN_{e_j}^{'}$ has all bits set after the first time of e_j, by taking the *or* of these two vectors we would obtain a new bitvector which has zeros only in position which are in between the last time of e_i and first time of e_j. Therefore, performing an inverse of this would result in the desired bitvector $S_{(e_i,e_j)}$. Note that the order in the pair is important (e_i, e_j): $SEP_{(e_i,e_j)}$ and $SEP_{(e_j,e_i)}$ are two different bitvectors (at least one of the two will be a zero bitvector).

The above statements for $SEP_{(e_i,e_j)}$ hold only if the last time of e_i is before the first time of e_j. Therefore, the constraint is encoded as follows, given the specified minimum d_{min} and maximum d_{max} times in between events:

$$\bigwedge_{(e_i,e_j)\in E^2_{spec}} (atLeast\_d_{min}[SEP_{(e_i,e_j)}]) \wedge (atMost\_d_{max}[SEP_{(e_i,e_j)}]) \tag{5.27}$$

$$\bigwedge_{(e_i,e_j)\in E^2_{spec}} (<(MAX_{e_i}, MIN_{e_j})) \tag{5.28}$$

If the constraint is specified as a soft constraint, additional modifications and equations are required. We do not discuss the encoding in detail and briefly sketch it instead. The main idea is to consider three cases: when the last time of e_i is before the first time of e_j, when the last time of e_i is exactly first time of e_j, and when the last time of e_i is after the first time of e_j. For each of these cases, we would encode constraints which penalize the objective function only if the given case is satisfied. In order to determine each case, equations similar to Equation (5.28) would be encoded, but with $<$, $=$, and $>$ operators. The penalty equations for the first case would correspond to the same as Equation (5.27) but with soft cardinality encodings, for the second case a fixed penalty would suffice, while for the third case an equation similar to Equation (5.27) with $SEP_{(e_j,e_i)}$ and soft cardinality encodings would be used.

Avoid Unavailable Times Constraints

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

Let UAT be the bitvector which has its *i-th* bit set if i is unavailable time. We encode the constraint as follows:

$$\bigwedge_{\substack{r\in R_{spec}\\ e\in E(r)}} (atMost\_0[and(Y_e, UAT)]) \tag{5.29}$$

Avoid Clashes Constraints

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

Let $E(r)$ be the set of events which require resource r. For each resource r, each time i and each combination of two Y_e vectors of events from $E(r)$ at most one bit at the *i-th* location may be set:

$$\bigwedge_{\substack{r\in R\\ e_1,e_2\in E(r)\\ e_1\neq e_2}} (=(and(Y_{e_1}, Y_{e_2}), 0)) \tag{5.30}$$

If the constraint is specified as a soft constraint, a different encoding should be used. Points of application are resources and deviations are calculated as follows: for each time in which the resource is used by two or more events, compute the number of events which require the resource minus one. Then, the sum of all these numbers is the deviation for a single resource.

We give equations which can be used if the cost function is linear, which we have used in our local search bitvector implementation. To do so, first we recursively define auxiliary variables $f_{(r,i)}$ (the index i goes from zero):

$$\bigwedge_{r \in R} (= (0, f_{(r,-1)})) \tag{5.31}$$

$$\bigwedge_{\substack{r \in R \\ e_i \in E(r)}} (= (or(Y_{e_i}, f_{(r,i-1)})), f_{(r,i)})) \tag{5.32}$$

The constraint cost for the linear case is then encoded as:

$$\bigwedge_{\substack{r \in R \\ e_i \in E(r)}} (atMost\_0[and(f_{(r,i-1)}, Y_{e_i})]) \tag{5.33}$$

Limit Idle Times Constraints

This constraint specifies the minimum and maximum number of times in which a resource can be idle during the times in specified time groups. For example, a typical constraint is to impose that teachers must not have any idle times.

A time t is idle with respect to time group tg (set of times) iff it is not busy at time t, but is busy at an earlier 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 $Wed6$. This constraint places limits on the number of idle times for each resource.

To ease the encoding of this constraint, we define a helper bitvector X_r for each resource, such that its *i-th* bit is set if resource r is busy at the *i-th* time:

$$\bigwedge_{r \in R} (= (X_r, \bigvee_{e \in E(r)} (Y_e))) \tag{5.34}$$

We define two other helper bitvectors: $G_{(r,tg)}$ and $H_{(r,tg)}$. For $G_{(r,tg)}$, the *i-th* bit is set if resource r is busy at some time within time group tg that takes place after i. For $H_{(r,tg)}$, it is similar except it considers times happening before i. For $G_{(r,tg)}$, these can be computed by taking *or* of bitvectors $rshift(and(X_r, mask_{tg}), k)$ where $k = 1..n$ and n is the number of times in time group tg. For $H_{(e,tg)}$ it is similar, except using $lshift$

instead of $rshift$. Before finalizing the encoding for this constraint, we define another auxiliary variable.

$$\bigwedge_{\substack{r \in R_{spec} \\ tg \in TG_{spec}}} (= (W_{(r,tg)}, and(inv(X_r), and(H_{(r,tg)}, G_{(r,tg)})))) \tag{5.35}$$

If for a resource r the *i-th* bit in $G_{r,tg}$ and $H_{r,tg}$ is set but not in X_r, then the *i-th* bit in $W_{(r,tg)}$ will be set indicating an idle time. We now encode the constraint.

$$\bigwedge_{r \in R_{spec}} (atMost\_idle_{max}[\bigvee_{tg \in TG_{spec}} (W_{(r,tg)})]) \tag{5.36}$$

A similar encoding to the one above is also used, but with $atLeast\_idle_{min}$.

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.

We define a helper bitvector B_r for each resource, in which the *i-th* bit is set iff the resource is busy at the *i-th* time group. Let us denote with tg_i the *i-th* time group, and with $B_r(i)$ and $X_r(i)$ the *i-th* bits of B_r and X_r[1], respectively. We can then encode this constraint as follows:

$$\bigwedge_{tg_i \in TG} (= (B_r(i), \bigvee_{t \in tg_i} X_r(t))) \tag{5.37}$$

This constraint specifies the minimum b_{tg}^{min} and maximum b_{tg}^{max} busy time groups:

$$\bigwedge_{r \in R_{spec}} (atLeast\_b_{tg}^{min}[B_r]) \wedge (atMost\_b_{tg}^{max}[B_r]) \tag{5.38}$$

Limit Busy Times Constraints

This constraint 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 in preventing situations in which teachers or students would need to come to school only to have a lesson or two.

A resource is busy at a time group tg iff it is busy in at least one of the times of the tg. We create a helper bitvector X_r which represents a bitvector which has its *i-th* bit set

[1] As defined by Equation (5.34).

if resource r is busy at time i. This can be done by taking the *or* of Y_e for all events which require resource r. With TG_{spec} we denote the set of sets of times given by the constraint and encode the constraint as follows:

$$\bigwedge_{\substack{r \in R_{spec} \\ tg \in TG_{spec}}} (or(atLeast\_b_{min}[and(X_r, mask_{tg})], (= (and(X_r, mask_{tg})), 0))) \tag{5.39}$$

The formula $exp = (= (and(X_r, mask_{tg})), 0)$ will return *true* if resource r is *not* busy within time group tg. Therefore, in this case the constraint given above will be satisfied. Otherwise, we force the *atLeast* constraint to be satisfied, limiting the minimum number of times r must be busy during that time group. With this, we capture the behavior we would like: if the resource is not busy during the day do not make any further constraints, but if it is busy make sure the resource works for at least b_{min} times. A similar encoding to the one above is also used, but with $atMost\_b_{max}$. Note that in this case *or* represents *logical or*, rather than *bitvector or*.

If this constraint is used as a soft constraint, the soft cardinality constraint is used instead, although special care must be given as this is a conditional cardinality constraint: if the calculated vector is different from zero then the cardinality constraints need to be fulfilled. Points of application are resources and for each resource its deviation is calculated as the sum of number by which the events group falls short of the minimum or exceeds the maximum for each time group.

Extending the Model

As mentioned in the beginning, we made the assumption that all resources have been assigned to events, as it is easier to model, implement, and present the formulation. This is a reasonable assumption, as most instances are of this form. Still, a significant part of the instances require assignments of resource to events. Our model can be extended with these requirements by introducing new bitvectors: for each event e and resource r, a bitvector is created in which the *i-th* bit is set iff resource r has been assigned to event e at time i. With these bitvectors, the other resource assigning constraints (we direct interested readers to [PKA+14]) can be encoded in a similar fashion as the ones already presented, along with certain modifications that need to be made to Avoid Clash Constraints. In the general case, this would lead to a significant increase in bitvectors and in turn might lead to longer solutions times, which is why particular cases rather than general ones should be considered (see next paragraph).

Special care needs to be given when doing so with concrete instances, as requirements for resource assignments can be diverse. For example, in instance *SpainInstance* given in the ITC repository, assignments consist of assigning one gym room out of two available. For instance *EnglandStPaul*, rooms need to be assigned and many symmetries appear because all rooms are identical. Hence, it might be a better idea to restrict the number of events at each time to the number of rooms, rather than assigning rooms directly to

events. A similar situation arises in $FinlandArtificialSchool$, where there are many rooms, but only three different types and a counting strategy like the one described for $EnglandStPaul$ would be more appropriate.

In addition, it may be of interest to simplify the $K_{(e,d)}$ and S_e encodings. The general formulation allows a variety of situations to be encoded, but in most instances times are partitioned into days, events do not span over more than one day, and an event has at most one starting time per day. With this in mind, we could simplify the encoding of $K_{(e,d)}$ and S_e from Section 5.1.1. One way to do so would be to forbid the appropriate $K_{(e,t)}$ variables so that events cannot span over multiple days and simply state that if an event has n consecutive times followed by an unset bit in a day that it has a starting time with duration n (the corner case being when the event ends at the last time of the day). This would lead to simpler encodings which would be potentially easier to solve than the general formulation.

When using the described model for implementing local search algorithms, one must decide whether to allow situations in which an event may clash with itself. For example, we may split a Mathematics lesson of four hours into two lessons of two hours. When scheduling this event, we schedule the first and second subevent to take place on the first or second time. However, since both subevents are of length two, the event will clash with itself. If such a situation is considered legal, then certain modifications to the present modeling need to be taken care of, as individual subevents need to be tracked and used in some constraints. For example, if an event is self clashing, when calculating its Spread Events Constraints one must check each of its subevents rather than using Y_e, since it may be the case that two subevents are scheduled to take place at the same time. In our local search implementation, we allowed self clashing events, since the KHE engine and state-of-the-art algorithms for XHSTT define this as a legal solution, although we note that forbidding self clashes significantly simplifies the implementation.

We note that our model cannot be directly used by constructive local search algorithms which would start from a solution with no assignments and construct a solution according to some heuristic. The reason is that when calculating the deviations for each constraint, it is assumed that all events are assigned the appropriate amount of times by Assign Times Constraint. Therefore, if one wishes to use our model with such an algorithm this needs to be taken into consideration and appropriate modifications should be performed when calculating deviations for constraints which are affected.

5.2 Computational Results

In this section we evaluated our bitvector model by using simple implementations of local search algorithms such as hill climbing and simulated annealing, as well as solving XHSTT with SMT. All tests were performed on (Intel Core i3-2120 CPU @ 3.30GHz with 4 GB RAM) and each instance was given a single core. We restricted the computational time per instance to 10 minutes for local search experiments and 24 hours for SMT

experiments. All produced solutions were verified using HSEval[2] and are available online[3]. We evaluated our approach we used the same instances as described in Section 4.3.1.

5.2.1 Bitvectors and Local Search

We have implemented basic variants of hill climbing and simulated annealing local search solvers for XHSTT using the presented bitvector approach to model XHSTT and calculate constraint violations. For comparison purposes, we have implemented the exact same algorithms using the engine KHE for calculating the constraint costs.

Brief Discussion on the Implementation

In KHE the solution consists of a number of subevents and their assigned times. It is important to note that subevents of the same event are allowed to clash with each other (constraints like Avoid Clash Constraints will penalize such solutions). We now discuss this particular situation in more detail, first by giving an example in KHE and then viewing the same situation with our model.

In KHE, for example, a math lesson of duration four hours can be split to two subevents with duration of two hours. If the first and second subevents are scheduled to take place at $Monday$ 9 am and $Monday$ 10 am (respectively), we will notice that there is an overlap at $Monday$ 10 am, because the second subevents starts while the first subevent is still taking place. Therefore, we have a clash of subevents. This is treated as any other clash and the appropriate constraints such as Avoid Clash Constraints apply.

However, in our general bitvector model we cannot have this situation as clashing subevents of the same event is not possible. Instead, for the previous example, the exact same solution using our model could be modeled such that one subevent of duration two starts at $Monday$ 9 am, another subevent of duration one starts at $Monday$ 11 am, and the event would have one hour of lessons unassigned. In this scenario Assign Time Constraints would penalize such an assignment rather than Avoid Clash Constraints as in KHE.

For the local search implementation we modified our model to take into account subevents. This is done by assigning a bitvector to each subevent. The number of subevents for each event is obtained after generating an initial solution. This modification introduces difficulties when checking some constraints, as in some cases one needs to check for an event whether it has multiple subevents starting at the same time, but this is done to make our implementation more similar to KHE.

We note that we believe the way our general model treats clashing subevents is more natural and appropriate, apart from it being simpler to calculate for our model when compared to the modification described above. For example, we find it unintuitive to allow a lesson to take place in the same time more than once, and that one can avoid

[2] http://sydney.edu.au/engineering/it/~jeff/hseval.cgi
[3] http://www.dbai.tuwien.ac.at/user/demir/XHSTT_SMT.tar.gz

violating Assign Time Constraints by creating a new subevent and assigning it a time in which another subevent of the same event is taking place, thus shifting the violation towards Avoid Clashes Constraint. A possible approach would be to encode subevents as separate events and modify the appropriate constraints to accommodate for this (e.g. Spread Events Constraint), but this would not eliminate our first concern. However, we agree that this is somewhat debatable and do not pursue further discussion on this in the following text.

Comparison of KHE and Bitvectors

KHE is the leading open source software library for the general high school timetabling. It offers users a lot of useful functionality when implementing XHSTT algorithms and has its own solvers as well.

The reason we chose simulated annealing and hill climbing is because they are closely related techniques to GOAL (the winner of the ITC 2011 [BFT+12a]), as well as the improvements made later on [FSC16b, FS14]. GOAL has been implemented using KHE, which is why we chose to compare our approach with KHE. We also use KHE to generate an initial solution.

Events are split into one or more subevents. Regarding the local search algorithms, two local search moves are considered: moving a randomly selected subevent to a new random time and swapping the assigned times of two randomly selected subevents. These moves are chosen because they have been used in [BFT+12a] and [FS14]. The algorithm by itself is a simplified version of the mentioned state-of-the-art algorithms. We deliberately keep the algorithm as simple as possible because the aim is to compare our modeling approach with KHE regarding the number of iterations.The algorithm implemented is described in Algorithm 5.1, which is a basic simulated annealing algorithm (one obtains a variant of hill climbing by omitting the second *or* part of the outer if statement).

In experiments, the following parameters were used: $T_{initial} = 0.1$, $T_{min} = 0.01$, $\alpha = 0.99$, $max\_no\_improvement = 10000$. The cost difference Δ was calculated as follows (taken from GOAL):

$$\begin{aligned}\Delta = (hardCost(s_{new}) - hardCost(s_{cur})) * 10000.0 + \\ + \frac{softCost(s_{new}) - softCost(s_{cur})}{hardCost(s_{best}) * 10000.0 + softCost(s_{best})}\end{aligned} \tag{5.40}$$

As a measure for comparison between KHE and our bitvector approach, we compare how many algorithm iterations could be performed in 10 minutes. In Table 5.1, we present both the objective value and number of iterations performed. We note that the running times for simulated annealing and hill climbing were very similar, therefore we only present one table.

Algorithm 5.1: Simulated Annealing

```
1  begin
2      s_best ⟵ s_initial
3      while enough time do
4          s_cur ⟵ s_best
5          T ⟵ T_initial
6          while T > T_min ∧ counter_no_improvement < max_no_improvement
           do
7              s_new ⟵ localMove(s_cur)
8              Δ = cost(s_new) − cost(s_cur)
9              if (Δ < 0) ∨ e^{−Δ/T_cur} > random(0, 1) then
10                 s_cur ⟵ s_new
11                 if cost(s_cur) < cost(s_best) then
12                     counter_no_improvement = 0
13                     s_best = s_cur
14                 end
15             else
16                 counter_no_improvement ⟵ counter_no_improvement + 1
17             end
18             T ⟵ T ∗ α
19         end
20     end
21 end
```

In each example our implementation managed to produce more iterations, with the results being mostly better. In some cases less iterations turned out better because of the stochasticity of the algorithms used. We excluded the instance NetherGEPRO because the generation of initial solution took more than the allowed computational time.

We used simplified variants of hill climbling and simulated annealing, because we wanted to show that the bitvector implementation can be used effectively in local search techniques and that it is possible to model the whole problem with the bitvector approach. As we experiment with very simple local search techniques the results are not competitive, but we can see that in each example our implementations produces more iterations.

We believe the improvements come from the data structures used, as they are very compact and simply consist of bitvectors. This makes certain constraints easy to calculate, but more importantly for simulated annealing it allows the solver to efficiently restart from another solution by copying the bitvector data structure which can be done very fast.

When calculating the cost function after performing a local move, KHE and our approach both recalculate costs for the affected resources and events, but the main difference is that KHE recalculates only part of the constraint, while we calculate the complete

Name	BV(obj)	BV(iter)	KHE(obj)	KHE(iter)
Brazil2	(1, 69)	233m	(1, 69)	36m
Brazil4	(22, 90)	212m	(22, 102)	15m
Brazil6	(5, 270)	226m	(4, 270)	11m
GreecePatras10	(6, 224)	39m	(10, 91)	3m
GreeceUni4	32	62m	32	10m
GreeceHSchool	0	34m	0	2.7m
Italy4	2047	85m	(2, 2927)	1.1m
FinlandHSchool	(0, 43)	98m	(0, 88)	12m
FinlandCollege	(9, 115)	93m	(14, 150)	1.6m
FinlandSSchool	(2, 147)	77m	(2, 154)	5m
KosovaInst	290	92m	(254, 17509)	0.15m
Brazil1	78	284m	78	53m
Brazil3	156	247m	171	31m
Brazil5	(8, 156)	236m	(10, 192)	13m
Brazil7	(1, 322)	208m	(10, 314)	5.4m
Italy1	36	181m	43	44m
FinlandSSchool2	50	65m	78	2.5m
FinlandESchool	(2, 4)	109m	(2, 6)	3m
GreecePreveza	(2, 334)	51m	(4, 169)	3.7m
GreeceUni3	17	85m	16	17.4m
GreeceUni5	10	90m	(1, 11)	18m
GreeceAigio	(2, 447)	26m	(0, 271)	1.1m

Table 5.1: Comparison of the bitvector approach and KHE for basic simulated annealing and hill climbing.

constraint cost. In our bitvector implementation, in some cases considering only a part of the constraint would not make a difference (e.g. Avoid Unavailable Times Constraint), but other constraints might benefit from it, although this has so far not been explored.

Although our implementation for simulated annealing and hill climbing as a whole currently shows better results than when using KHE, we cannot make a general claim that our modeling is better than the approach used by KHE. Indeed, it could be that KHE is more efficient in particular solution components, but this is hard to evaluate as it is difficult to view algorithm components isolated. Nevertheless, our results show that our modeling approach is a useful modeling approach for XHSTT and can be used as it is by local search techniques and SMT.

5.2.2 Bitvectors for SMT

We evaluated modeling HSTT with bitvectors for Satisfiability Modulo Theories (SMT). The developed bitvector modeling is suitable to be used for solving XHSTT with SMT

solvers which provide tools for reasoning over bitvectors. To test our approach we used the same instances as described in Section 4.3.1.

We experimented with the SMT solver Z3 (v4.4.2) [DMB08] with optimization support [BP] using the *wmax* optimization engine. We chose this solver because, to the best of our knowledge, it is the only active solver that supports optimization over bitvectors. When modeling we used the encoding for cardinality constraints as described in Section 5.1.2 rather than population count instructions (mentioned in 5.1.2). The reason for doing so was because there is no support for cardinality constraints in the solver.

We restricted the computational time to 24 hours with one core. The time to convert an instance from XHSTT to an SMT instance is negligible when compared to the SMT solution process.

The comparison of SMT solutions and best known results can be found in Table 5.2. For each instance we display only the soft constraint cost if the hard constraint cost part is zero. Otherwise, we use a dash to indicate that no feasible solution has been calculated. Our model differentiates only between feasible or not feasible (hard constraints equal to zero or not), that is, it does not give the hard costs. For $ItalyInstance4$, $(0, x)$ means that an initial solution was computed but no optimization could be performed. The instances in the upper part of the table (separated by the hold horizontal line) represent instances that were used in the final phase of ITC 2011, while the other instances were used in previous phases. The table only displays instances which we could model with our approach.

In all of the instances (except KosovaInstance and NetherGEPRO), the SMT solver managed to compute an initial solution within a few minutes and do some optimization. For three instances (Brazil1, GreeceHighSchool, and FinlandESchool) optimal solutions were found. However, overall when compared to the best existing results, the SMT method is not competitive, although one must consider that the best known results were obtained without any time or resource limitations.

Therefore, given the current state, it would be best to use our approach to generate an initial solution for a local search, as local search algorithms can struggle in some cases to find a feasible solution (e.g. see Table 2 in [FS14]). Finally, SMT solvers are continuously being improved and future developments of SMT optimization will directly improve our results.

5.3 Summary

We presented a new bitvector modeling of the general high school timetabling problem (XHSTT). We modeled all constraints, except for those that deal with resource assignments. With our approach, we could model 23 out of 39 used instances. We considered instances that were used in the International Timetabling Competition 2011 (ITC 2011) and ones which were carefully selected by ITC 2011 after the competition.

Name	SMT	Best
Brazil2	54	5
Brazil4	166	51
Brazil6	226	35
GreecePatras10	883	0
GreeceUni4	163	3
GreeceHSchool	**0**	0
Italy4	(0, x)	27
FinlandHSchool	371	1
FinlandCollege	2311	0
FinlandSSchool	3502	77
KosovaInst	-	0
Brazil1	**41**	41
Brazil3	72	24
Brazil5	177	19
Brazil7	452	53
Italy1	532	12
FinlandSSchool2	3343	0
FinlandESchool	**3**	3
GreecePreveza	1080	0
GreeceUni3	120	5
GreeceUni5	94	0
GreeceAigio	1790	0
NetherGEPRO	-	(1, 566)

Table 5.2: Comparison of SMT and best known results . Bold values indicate optimal solutions.

The main use of our bitvector model is as a data structure which efficiently represents XHSTT for local search algorithms such as hill climbing and simulated annealing. We provided an experimental evaluation of the bitvector approach by comparing it to the leading engine KHE on simple hill climbing and simulated annealing algorithms. The bitvector approach demonstrated excellent results on the mentioned algorithms.

Additionally, the model was used to encode XHSTT as a Satisfiability Modulo Theory (SMT) problem and use Z3, an SMT solver, to provide timetabling solutions. Although this complete approach was able to provide optimal solutions for three instances, we note that overall the presented maxSAT approach in Chapter 4 is more effective.

CHAPTER 6

MaxSAT-Based Large Neighborhood Search for High School Timetabling

In this chapter, we describe our new algorithm for XHSTT which combines local search with a novel maxSAT-based large neighborhood search. A local search algorithm is used to drive an initial solution into a local optimum and then more powerful large neighborhood search (LNS) techniques based on maxSAT are used to further improve the solution. We proved the effectiveness of our approach with experimental results on instances described in Section 4.3.1, by comparing our algorithm to other XHSTT state-of-the-art solvers. For the instances that could be modeled with maxSAT, our algorithm shows good performance and for several instances new best known upper bounds have been computed. We also present several variants of the algorithm, in order to illustrate the importance of each component.

The rest of the chapter is organized as follows. The main section is Section 6.1, where we describe our algorithm. Afterwards, Section 6.2 provides experimental results on benchmarks instances. Lastly, a summary is given in Section 6.3.

6.1 Algorithm Description

We introduce a new algorithm for the XHSTT problem, which combines local search and maxSAT-based large neighborhood search (LNS). LNS is a technique first introduced by Shaw [Sha] and has been used for many problems, including related timetabling problems [MO], but never in a combination with maxSAT. The LNS algorithm is the main contribution of this paper and consists of two main components: destroy and insertion operations. The destroy operator unschedules certain subevents. Insertion is

the opposite: it assigns times to the previously unscheduled subevents. We proceed by explaining in detail the destroy operator, the insertion technique with maxSAT, the initial solution generation by local search, and finally we give a complete overview of the algorithm.

6.1.1 Destroy operator

The destroy operator selects a neighborhood from one of the two neighborhood vectors and destroys the solution with respect to the selected neighborhood. The two neighborhood vectors are based on resources and days. We first introduce these vectors and then explain how a neighborhood is selected from them.

Neighborhood vector based on resources. This vector consists of all possible combinations of two resources (e.g. rooms, teachers, etc). When a neighborhood from this vector is used, every subevent which uses at least one of these two resources is unscheduled (unassigned from each time unit), as well as every subevent which is linked to any of the unassigned subevents via Link Event Constraints. A similar idea of unscheduling resources has been presented in [Kin12].

Using events that share one of the selected resources and linked events are both important for the insertion step. For example, students in HSTT have compact schedules and attempting to assign a subevent which requires class C to a new time without previously unassigning other subevents which also require class C will most likely result in a clash. Since linked events should take place simultaneously, unassigning one subevent requires that its linked subevents are also unassigned in order to be able to schedule all of the subevents at a different time.

Neighborhood vector based on days. This vector consists of all possible combinations of days. For example, if we are considering a timetable with three days {Mon, Tu, Wed}, the corresponding day vector would be: { {Mon}, {Tu}, {Wed}, {Mon, Tu}, {Mon, Wed}, {Tu, We}, {Mon, Tu, Wed} }. When a neighborhood from this vector is used, subevents assigned to times pertaining to the days of the selected neighborhood are unscheduled.

This neighborhood vector is used to make better rearrangements within each individual day (e.g. for Limit Idle Times Constraint), as well as to be able to move, merge, or split subevent throughout different days (e.g. for Spread and Cluster Events Constraints).

Neighborhood selection. Only one of the two neighborhood vectors is considered active and neighborhoods are selected from the active one. The other neighborhood vector will become active after either a timeout occurs (equal to half of the total running time) or if all neighborhoods from the active vector have been visited exactly twice. A neighborhood can be visited a second time only after all other neighborhoods in the vector have been visited once, and no neighborhood will be visited more than twice within a single activation of its neighborhood vector.

After using a neighborhood to destroy the solution and applying the insertion operator based on a maxSAT solver, we record the objective value and whether the insertion

operator has successfully exhaustively explored the neighborhood. If the maxSAT solver did not show progress within a certain amount of time (did not find a better solution nor proved that one does not exist), we stop further exploration, label the attempt unsuccessful, and move on to the next neighborhood. Initially we set the objective value of a neighborhood to be a large number and label it as if an insertion operation has successfully terminated. Note that if the recorded objective value for a neighborhood is the same as the current upper bound then it means that the solution has not changed since the last time the neighborhood was visited.

The order of the neighborhoods in a neighborhood vector are reset randomly when a neighborhood vector becomes active or when all of its members have been visited once. Neighborhoods are then visited in order, with the exception of the day neighborhood vector where smaller neighborhoods are visited first before proceeding to larger ones (e.g. single-day neighborhoods are visited before two-day neighborhoods).

In some cases a neighborhood will be skipped rather than examined. This is done in order to avoid having the solver spend time with neighborhoods that are likely to be unsuccessful. A neighborhood is skipped if its recorded objective value (previously explained in this section) is equal to the current upper bound, and any of the following conditions hold:

- Its previous insertion attempt was successful. The neighborhood is surely of no use as it has already been thoroughly explored.
- It is being visited for the second time since the current neighborhood vector has been set active. In this case we heuristically choose to skip the neighborhood, because it has already been unsuccessfully explored, so the chances are that it will be unsuccessful again with the same amount of time. It is better to allocate time to other neighborhoods.
- The neighborhood is an element of the day neighborhood vector, and since the last time the neighborhood vector became active, another neighborhood which is a subset of the currently considered one has not successfully been explored. For example, the three-day neighborhood {Mon, Tu, Wed} will be skipped if the single-day neighborhood $\{Mon\}$ has been labeled unsuccessful.

Each neighborhood will be allocated a specified amount of time. For each subsequent run on a neighborhood that has not successfully terminated, more time is allocated. After successfully terminating, the next run on that neighborhood will be given the default amount of time.

We now further comment on our motivation for choosing this neighborhood selection strategy. Our strategy was devised through heuristic reasoning and experimentation, leaving the possibility that we might have missed other better strategies. Nevertheless,

experiments in Section 6.2 show that our approach is effective. The two main reasons are as follows:

First, the idea is to examine smaller neighborhoods before going on to larger ones. Accordingly, local search (see Section 6.1.3) takes place before the LNS algorithm, resource based neighborhoods precede the day neighborhood vector, and smaller day neighborhoods are examined before larger ones. The reason is that smaller neighborhoods are much easier to explore and often provide improvements quickly. Once they are no longer usable, we go to larger ones, which will now be searched faster because of the clauses learned and bounds obtained (see Section 6.1.2) in the process of solving smaller neighborhoods (as variant LNS.v2 confirms, Section 6.7).

Second, we want to thoroughly examine all neighborhoods in the hope of finding good ones. We wish to avoid situations where a useful neighborhood is overlooked because other neighborhoods are selected repeatedly. Therefore, we introduce fairness where a neighborhood will be examined a second time only after all other neighborhoods have been examined at least once. We chose to explore every neighborhood exactly twice before moving on to the other neighborhood vector. Considering the same neighborhood again after a series of other neighborhoods have been examined might be useful, but we opted to not examine it more than twice in order to escape from a search space which consumes too much time. The neighborhood skipping mechanism directs the search by filtering out neighborhood which are unlikely to contain better solutions.

6.1.2 Insert operation

The insertion operation repairs the solution by trying to find the best possible insertion for the unscheduled events using an exhaustive search based on a maxSAT formulation. The idea is to call a maxSAT solver to solve the maxSAT formula which represents a XHSTT-instance, while fixing the assignments of all subevents which were *not* unassigned in the destroying phase. We use the maxSAT formulation described in Chapter 4. Recall that the formulation relies heavily on the usage of Boolean variables $Y_{e,t}$, which are true iff a subevent of event e is taking place at time t (not just starting at t). In principle, any exhaustive search technique could be used for the insertion operator. In Section 6.2.5 we provide a comparison of maxSAT with Integer Programming and further elaborate on our decision to choose maxSAT.

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

We use the Linear maxSAT algorithm (Algorithm 6.1) [LBP10] which makes repeated calls to a SAT solver and after each call adds constraints which ask for a better solution

than the previous one (in Algorithm 6.1 K is the set of soft constraints). The optimal solution is obtained when the SAT solver returns $false$.

Another maxSAT algorithm is based on *unsatisfiable cores*. An unsatisfiable core is a set of clauses whose conjuction is unsatisfiable. Initially one may consider all soft clauses as hard and then attempt to solve the formula. If the solver reports the formula is unsatisfiable (not all soft clauses can be satisfied), an unsatisfiable core is computed and used to relax the SAT formula and the process is repeated iteratively.

Algorithm 6.1: Linear Algorithm for maxSAT

```
1  begin
2      P ⟵ maxSAT formula
3      c = ∞
4      bestAssignment = ∅
5      while isSatisfiable(P) do
6          bestAssignment = satisfiableAssignment(P)
7          c ⟵ cost(P, bestAssignment)
8          P = P ∪ (∑_{i∈K} softConstraint(i) < c)
9      end
10 end
```

We opted to use the linear algorithm as it was one of the algorithms that had good performance for XHSTT (see Section 4.3). In this algorithm, the original maxSAT formula is changed because bounds are added at each iteration, in addition to *learned* clauses which are added to direct the search (see [SLM09] for clause learning). It is not straightforward to remove the added clauses at later stages of the algorithm, because clauses are learned with respect to other clauses (including other learned clauses) and removing some clauses may therefore invalidate previously learned clauses. To the best of our knowledge, no maxSAT solver supports this kind of search. An alternative is to restart the solver after each call, losing possibly valuable learned clauses and bounds. This motivated us to investigate a different approach: instead of restarting between calls, we keep the modified formula intact. Thus, each call to the solver depends on all previous calls due to the bounds and learned clauses. When querying the solver with a new set of assumptions, it will attempt to report the best solution possible, but only if it is better than all of the previously computed solutions. To this end, we modified the linear algorithm in the open-source maxSAT solver Open-WBO [MML]. Keeping the solver state between runs proved to be important and this is discussed in more detail in Section 6.2.6. A different related approach related to ours is presented in [MJML] for lower bounding maxSAT algorithms.

6.1.3 Initial solution

We use two approaches for generating an initial solution. The first approach is to use the KHE14 algorithm [Kinb] to obtain an initial solution. We chose KHE14 because it

is a publicly available state-of-the-art solver designed to produce high quality solutions very quickly. If KHE14 does not succeed in generating a feasible solution, we generate an initial solution by ignoring all soft constraints and solving the corresponding XHSTT as a pure SAT instance, with the exception of Split Events Constraints which are treated as hard constraints even if they are given as soft ones. This solution is then improved with a local search procedure. Split Events Constraints are treated as hard constraints because the following local search algorithm does not split or merge subevents, making it very difficult to find a good solution if the constraint is not satisfied initially. The local search procedure is based on simulated annealing (SA) and it uses two neighborhoods: swap (exchange the times of two subevents) and block-swap (similar to a swap, with the exception that if a swap move causes two subevents to overlap, assign an appropriate time to the second one so that they appear one after the other). Similar neighborhoods were previously used in [FS14]. In our algorithm we only apply feasible moves to subevents which share resources. The importance of using subevents with shared resources has been discussed in Section 6.1.1. The following parameter values are used in the SA: the initial temperature $T_{initial}$ is set to 0.1 and the temperature is multiplied by $\alpha = 0.99$ every 10 iteration. If the last five improvements are made at the initial temperature, then the algorithm starts accepting only improving moves. The probabilities of selecting the swap or block-swap neighborhoods at each iteration are set to 2/3 and 1/3, respectively. When a neighborhood is selected, two subevents are randomly selected until they both share a resource (this is attempted 100000 times). The goal is to quickly improve the solution simple moves leaving more complicated moves to LNS. We note that the initial solution generation part of our algorithm takes only a small amount of time when compared to the maxSAT-based LNS part. However, it is still important and we discuss in Section 6.2.6 the impact of using a non optimized initial solution (c.f., $LNS.v1$ variant).

6.1.4 Algorithm summary

The pseudo-code in Algorithm 6.2 summarizes our problem-solving approach. An initial solution is generated and improved with SA, after which the LNS provides further improvements. LNS consists of two parts: the destroy operator which unschedules subevents based on a neighborhood chosen using either the resource or day vector (described in Section 6.1.1), and the insertion operator which reschedules the previously selected subevents and records the performance of the chosen neighborhood. The algorithm iterates until a timeout occurs (in our experiments, 1000 seconds).

6.2 Experimental Results

In this section we introduce problem instances, experimentally evaluate our algorithm including different variants, and compare with existing state-of-the-art solutions.

Algorithm 6.2: Large Neighborhood Search XHSTT Algorithm

```
1  begin
2      S ⟵ initialSolutionAndLocalSearch()
3      t_current ⟵ current time; t_vec ⟵ t_current
4      t_default ⟵ 30 secs; t_max_active ⟵ 500 secs
5      (V_r, V_d) = generateNeighborhoodVectors()
6      active(V_r) ⟵ true; active(V_d) ⟵ false
7      while there is time left do
8          if condition for vector switch satisfied then
9              switch active vector
10             t_vec ⟵ t_current
11             randomize orderings of active vector
12         end
           // Section 6.1.1
13         N ⟵ selectNeighborhood()
14         S_destroyed ⟵ destroy(S, N)
15         S_old ⟵ S
16         S ⟵ insert(S_destroyed, allocatedTime(N))
17         if insertion was successful then
               // Used for selection
18             previouslySuccessful(N) ⟵ true
               // Used for selection
19             cost(N) ⟵ cost(S); allocatedTime(N) ⟵ t_default
20         else
               // Used for selection
21             previouslySuccessful(N) ⟵ false
22             allocatedTime(N) ⟵ allocatedTime(N) + 10
23             S ⟵ S_old
24         end
25     end
26 end
```

6.2.1 Computer Specification, Computational Time Limit, and Instances

All tests were performed on an Intel Core i3-2120 CPU @ 3.30GHz with 4 GB RAM and each instance was given a single core. To determine the computational time we used the ITC's benchmark tool which is designed to test how fast a machine is at performing operations relevant for timetabling. The tool suggested a computational time 1007 seconds, which is a similar computational time used in the second round of ITC 2011 (1000 seconds). We evaluated our approach on the same instances as described in Section 4.3.1.

6.2.2 Solvers

We compared our approach (abbreviated by LNS) with VNS [FS14], KHE14 [Kinb], Matheuristic [SS, Sør13b], and a pure maxSAT approach, all using the time limit of 1000 seconds. Additionally, we ran our solver for longer running times and compared its results to the best known upper bounds.

VNS [FS14] was developed by the winners of ITC 2011 after the competition. KHE14 [Kinb] is a competitive solver also used in our approach for the initial solution, as described in Section 6.1.3. Matheuristic [SS, Sør13b] is an Integer Programming-based LNS algorithm. These solvers were chosen because they are state-of-the-art XHSTT algorithms which can generate good solutions in the time limit set by the competition. The pure maxSAT approach runs the maxSAT solver used in our algorithm (open-WBO) on the same maxSAT model of a XHSTT-instance as LNS, but without using any of our LNS techniques.

6.2.3 Results

The algorithms were run for the same amount of time (1000 seconds) on the same machine, except Matheuristic because it is not available to the public. In this case, we compare with the results reported in [SS, Sør13b]. We believe the comparison is fair because the authors used the same benchmarking tool as we did to determine the computational time. When no result was reported with Matheuristic for a given instances, we put '-' in the table. Since KHE14 was designed to run for shorter durations, we ran the algorithm multiple times during the time limit and present the best solution found.

We denote the objective function cost as a pair (x, y), where x and y are the hard and soft constraint costs. If the hard cost is zero, we only present the soft cost. The algorithms include some forms of randomness during their execution and we present the average values of five runs. For BrazilInstance5, we included the best solution computed out of the five runs, since the solution represents the best solution known for this instance so far.

The comparison is given in Table 6.1. Instances noted above the bold horizontal line are part of the XHSTT-2014 benchmark set, while the ones below are not but have been used in the competition.

Name	LNS	VNS	KHE14	maxSAT	Math.
Brazil2	**5.4***	(1, 44.4)	14	57	6
Brazil4	61.4	(17.2, 94.8)	$-^{c}$	214	**58**
Brazil6	**50.6**	(4, 223.6)	124	352	57
GreecePatras10	**0***	**0***	**0***	2329^{m}	-
GreeceUni4	**5**	6.2	8	141	12
GreeceHSchool	**0***	**0***	**0***	**0***	-
Italy4	**35**	178	40	16979^{m}	48
SAfricaWood	1.2*	(2, 6.2)	(3, 0)	**0***	(2, 429)
SAfricaLewitt	$-^{m}$	**8**	$-^{c}$	1039^{m}	-
FinlandHSchool	**9.8**	36.6	29	812	-
FinlandCollege	54.6	(2.8, 25)	**20**	1309	-
FinlandSSchool	95.2	(0.4, 93)	**90**	504	-
KosovaInst	$-^{m}$	14	(8 , 6)	29946^{m}	(9, 23525)
EngStPaul	-	(92, 1739.4)	(26, 764)	$-^{m}$	-
Brazil1	**39**	52.2	54	**39**	-
Brazil3	**23***	107.8	116	75	-
Brazil5	**19.4 (17)**	(4, 138.4)	(1, 179)	224	-
Brazil7	**136.2**	(11.6, 234.6)	179	603	-
Italy1	**12**^{p}	21.2	31	**12**^{p}	-
FinlandSSchool2	**0.2***	**0.2***	2	3523	6
FinlandESchool	**3**^{p}	**3**	4	**3**^{p}	**3**
FinlandASchool	**0***	(5.4, 4.2)	(4, 6)	12	-
GreecePreveza	38.2	**2***	**2**	5617	-
GreeceUni3	7	**5**	7	7	6
GreeceUni5	**0***	**0***	**0***	**0***	**0***
GreeceAigio	368	(0.2, 6.2)	**6**	4582	180
NetherGEPRO	$-^{m}$	(3, 7518)	$-^{c}$	$-^{m}$	-

Table 6.1: Comparison of results with solvers with the time limit of 1000 seconds. The best results are in bold. Legend: * optimality was found within the five runs; ^{m} 'out of memory'; ^{p} proof of optimality; ^{c} program crash (a bug); and " $-$ " solution has not been produced within the time limit .

Our approach outperforms the VNS solver for 16 instances out of 27, while in five cases gives the same result. For KHE14, we obtain better results in 15 cases and remain equal in four cases. When compared to the pure maxSAT approach, our algorithm has a clear advantage given the computational limit. As for the Matheuristic algorithm, we perform better in five out of nine instances and are tied in one instance based on the published results. Overall, our solver is better or equal than any other solver in 16 cases. We note that we evaluated our results with respect to the recent hyper-heuristic approach [KK16]

as well. For every instance in Table 6.1 our approach produced better results compared to the results given in [KK16].

We note that a new best known upper bound has been computed for BrazilInstance5 within the time limit. Moreover, in two cases our solver found and proved optimality within the time limit, while other solvers are not able to generate proofs of optimality. Our method is able to prove optimality for these instances because larger and larger neighborhoods are explored over time until the whole problem is solved (the complete solution is destroyed and optimally reassigned). Further improvements to the best known solutions with other time limits are discussed in Section 6.2.4.

Overall, the results obtained are very encouraging, as our approach outperformed the state-of-the-art solvers on many instances that were modeled with our maxSAT approach.

6.2.4 Longer runs and additional improvements to the best known solutions

With more computational time, it was possible to produce three new best known solutions (in addition to the ones from the previous section). For the KosovaInstance, we ran our algorithm using the previous best known solution as a starting point. In Table 6.2, we present the results in column $LNS(time)$ together with the time used in brackets, but only when it was possible to produce better solutions with longer running times. In column $'Best'$ we give the previously best known solution values. The other instances (including the ones that we have previously computed optimally) are marked with $'-'$. We note that after providing our results for the Brazilian instances to ITC 2011's repository, the instance modelers decided to slightly modify the instances. After reviewing our solution, they decided to make Prefer Times Constraints hard instead of soft constraints. Nevertheless, we provide these results as no other method could compute these solutions.

6.2.5 maxSAT vs Integer Programming

We now provide a comparison of maxSAT with Integer Programming and further elaborate on our decision to choose maxSAT for the insertion operator.

We compared Integer Programming[1] [KSS15] and pure maxSAT for XHSTT. Both solvers were run for 30 seconds (the initial time used in this paper for each neighborhood) and 600 seconds using a single thread and the results are given in Table 6.3. In most instances the pure maxSAT approach gave better results. The maxSAT results can easily be reproduced by running our maxSAT instances [2] with open-WBO (parameter: -algorithm=1). Based on the results, we believe that maxSAT is an appropriate exact method to be used as part of LNS.

In addition to the above, we note two other very important points for our maxSAT choice: this is the first time, to the best of our knowledge, that maxSAT has been used in a

[1] We thank the authors for providing their solver

[2] http://www.dbai.tuwien.ac.at/user/demir/WCNF_SNA.rar

Name	LNS(1000s)	LNS(time)	Best	LNS.v1	LNS.v2	LNS.v3(R)	LNS.v3(D)
Brazil2	5	-	5	5.4	5	12.6	5
Brazil4	61.4	53 (4000)	51	60.6	69	91	88.2
Brazil6	50.6	**34** (27000)	35	99	75	56.8	74.6
GreecePatras10	0	-	0	57.8	6	0.5	0
GreeceUni4	5	-	5	38.8	6	13	5
GreeceHSchool	0	-	0	0	0	0	0
Italy4	35	35 (3500)	34	$-^m$	52	48	55
SAfricaWood	1.2	-	0	1.2	0.4	27.8	0
SAfricaLewitt	-	-	0	-	-	-	-
FinlandHSchool	9.8	8 (3200)	1	37.2	15.8	14.6	23.2
FinlandCollege	54.6	23 (4000)	0	43.4	84.8	50	643.8
FinlandSSchool	95.2	82 (9000)	77	92.2	104	117	126
KosovaInst	-	**0** (300)	3	-	-	-	-
Brazil1	39	38 (10000)	38	38.6	38	41.8	38.8
Brazil3	23	-	23	23.6	23	61	23.2
Brazil5	19.4	**17** (20000)	20	23.4	27.6	43.6	27.6
Brazil7	136.2	**57** (14000)	67	282	176	114.4	217.6
GreecePreveza	38.2	0 (1200)	0	39.6	69	168.8	5.5
GreeceUni3	7	-	5	8	6	10	7.2
GreeceUni5	0	-	0	0	0	1.2	0
GreeceAigio	368	97 (5300)	0	$-^m$	442	461.2	200
Italy1	12	-	12	12.2	12.2	14	12
FinlandSSchool2	0.2	-	0	76.8	1	5	0
FinlandESchool	3	-	3	3	3	3	3
FinlandASchool	0	-	0	0	0	30	0

Table 6.2: Comparisons of results for longer running times, variants of our algorithm, and previously best known upper bounds. The best known upper bounds computed are in bold.

LNS algorithm. Second, Gurobi (the Integer Programming solver used in [KSS15]) is a commercial highly engineered piece of software, while open-WBO (the maxSAT solver we modified) is open source and not so heavily engineered, but still provides competitive results.

Name	maxSAT(30s)	IP(30s)	maxSAT (600s)	IP (600s)
Brazil2	**98**	(16, 293)	**55**	87
Brazil4	**261**	(55, 266)	**199**	241
Brazil6	**505**	(162, 580)	**341**	609
GreecePatras	**3219**	-	3219	**25**
GreeceUni4	**186**	193	141	**55**
Italy4	**18671**	(2387, 26756)	**18671**	21228
SAWoodlands	**4028**	-	**811**	-
SALewitt	**1189**	-	1189	**802**
FinlandHSchool	**1231**	(331, 981)	501	**351**
FinlandCollege	**1056**	-	**1056**	(546, 1185)
FinlandSSchool	**765**	(30, 1053)	575	**166**
Brazil1	**44**	59	41	41
Brazil3	**171**	(77, 407)	**65**	93
Brazil5	**406**	(118, 487)	**225**	585
Brazil7	**819**	(240, 887)	**771**	(157, 908)
Italy1	2039	**141**	27	**17**
FinlandSSchool2	**2464**	-	**2464**	(279, 3483)
FinlandESchool	**3**	-	**3**	4
FinlandASchool	**4004**	(416, 125)	**43**	(202, 111)
GreecePreveza	**5617**	-	**5617**	(18, 3145)
GreeceUni3	161	183	122	**37**
GreeceUni5	**100**	52	**32**	37
GreeceAigio	**4582**	-	**4582**	(6609, 195)

Table 6.3: Comparison of Integer Programming [KSS15] and pure maxSAT for XHSTT. The hypen in the table indicates that the solver was not able to produce a solution within the specified time limit.

6.2.6 Variants of the algorithm

We experimented with three variants of our algorithm and compared them with our standard algorithm ('LNS' column), still using 1000 seconds of computational time. The results of all three variants are given in Table 6.2. Each variant is aimed at testing a certain aspect of our algorithm.

Variant LNS.v1 is similar to our standard algorithm, except that it uses an unoptimized initial solution as a starting point, generated by ignoring all soft constrains and solving the problem as a SAT problem. Based on the results, we conclude that using optimized initial solutions is better overall, since local search quickly eliminates simple improvements, leaving more time for the maxSAT solver to deal with the more challenging ones.

The second variant LNS.v2 consists of restarting the maxSAT solver between different calls. In the original version, the maxSAT solver is not restarted from scratch, thus all learned clauses and previous bounds are kept (see Section 6.1.2). The lower quality of results indicate that restarting the maxSAT solver after every insertion operation is detrimental. We note that the total time required for restarting the solver was not very significant (typically under 50 seconds).

The third variant LNS.v3(R) and LNS.v3(D) consists of using only one of the two proposed neighborhood sets (resource or day vector), in order to analyze whether the combination of neighborhoods is beneficial. Overall these variants seem to be worse than the standard version. We believe that the resource neighborhood vector can provide quick improvements and let the more complicated ones be performed by the day neighborhood vector. Thus, both neighborhoods complement each other.

6.3 Summary

We presented a new large neighborhood search algorithm which exploits maxSAT to solve XHSTT. We proposed a destroy operator with two neighborhood vectors and a novel insertion approach, for which we modified the open-source maxSAT solver Open-WBO [MML] to support our exhaustive insertion strategy. The overall algorithm combines local search and large neighborhood search to solve XHSTT-instances which we modeled by maxSAT.

We experimentally compared our algorithm to other approaches in the literature on 27 out of 39 instances. Our approach outperformed the state-of-the-art solvers on many instances which we modeled by maxSAT. Using our algorithm, we managed to compute four new best known upper bounds. In addition, in contrast to pure metaheuristic approaches, our algorithm was able to prove optimality for two instances with nonzero solutions. The presented approach is a novel contribution to the state-of-the-art for XHSTT. Furthermore, to the best of our knowledge, the first time maxSAT is used within a large neighborhood search scheme.

In addition, to demonstrate the importance of each component and gain further insight into the inner workings of our algorithm, we experimented with four different variants. The experiments reveal several important lessons. Optimizing the initial solution by inexpensive local search techniques has proven to be valuable, as the powerful maxSAT techniques are saved for more challenging problems. Keeping the solver state in between successive maxSAT calls improves the overall performance, as the kept learned clauses

aid the maxSAT search algorithm. Lastly, combining both neighborhood vectors yields better results than using any of them individually.

CHAPTER 7

Conclusion

In the previous chapters, we described the general high school timetabling problem, introduced a formal definition, as well two different modeling approaches, SAT- and bitvector-based. By modeling XHSTT with maxSAT and bitvectors, we were able to take advantage of existing solvers to generate timetabling solutions. We have done extensive experiments with the maxSAT formulation in order to determine the best modeling choices (for cardinality constraints and important special cases) and maxSAT solvers. The results proved that our maxSAT approach is state-of-the-art for XHSTT, outperforming the state-of-the-art complete approach based on integer programming in many instances. Our bitvector modeling performed well for local search algorithms. Lastly, our maxSAT-based large neighborhood search algorithm has demonstrated to be state-of-the-art as well, excelling in producing quality solutions in limited computational time. Each of the described methods are vastly different from each other and represent distinct ways to tackle XHSTT.

Our work shows that SAT-based approaches are promising for very complex problems such as the general high school timetabling problem. This implies that investigating its application for other timetabling problems is a promising research direction. Moreover, the hybridization of maxSAT and large neighborhood search is able to provide significantly better solutions than a pure maxSAT approach within short computational times. To the best of our knowledge, this was the first time that a large neighborhood search algorithm was used with maxSAT, hinting that a similar approach might be useful for other problems, but also for maxSAT itself. In addition, the modelings can be useful as concise and compact representations of the problem at hand, serving as efficient data structures for local search algorithms.

The formulation of XHSTT into a propositional logic formula is a challenging process, since precisely stating the problem in the low-level language of propositional logic can be difficult. Many symmetries may be present in the problem, which can severely hinder the solution process if they are not given special attention. As our work has shown, it is very

important to consider different modeling options and solution techniques. In our case, even after extensive experimentation, there was no clear winner among the modeling and solver choices. There were a number of good options, with each modeling or solving method having its own advantages depending on the instance. This should not be seen in a bad light, as having multiple complementary choices can lead to better and more robust solution approaches.

From our empirical results with variants of our maxSAT-based large neighborhood search algorithm, we concluded that starting from a good initial solution is important, as well as using multiple different neighborhoods. In addition, it was very beneficial for the algorithm to learn about the search space at run-time. In our example, we did so by keeping the learned clauses during the search in between solver calls and performing book-keeping to avoid using nonpromising neighborhoods.

Overall, this thesis introduced several novel approaches for XHSTT and gave contributions to the state-of-the-art for XHSTT. There has been tremendous progress since the ITC 2011, but we do not consider XHSTT to be *solved*. It still remains a challenging problem to find and prove optimal solutions in reasonable time in many instances. There are a number of open issues left, both in general for XHSTT and our algorithms. We highlight some of the issues in the next section.

7.1 Future Work

An important topic for future work would be to consider a portfolio-based algorithm selection approach. Such approaches analyze a given instance and select the most appropriate algorithm for it based on the instances' features. Two issues are to be explored. The first is to decide which cardinality constraint encoding to use given a set of variables, the cardinality value k, and constraint in question. This is suitable for a portfolio-based algorithm approach, as we can view different modeling strategies as different algorithms to choose from. We have partially addressed this problem in our experiments. However, instead of using one cardinality constraint encoding for a wide range of constraints, it might be worthwhile to use several different ones, depending on the characteristics of the cardinality constraint. The second issue is to determine which maxSAT solver to use given a XHSTT instance. From our experiments (Chapter 4), it can be seen that better results can be obtained if different maxSAT solvers are considered. Taking this methodology further, we believe it would be advantageous to unite all of the XHSTT solvers, in order to exploit their diversity. We believe studying XHSTT instance features, which is necessary for a portfolio approach to be successful, will additionally result in a deeper understanding of the challenges XHSTT algorithms face.

Our large neighborhood search approach (Chapter 6) proved that combining domain-specific knowledge and maxSAT can lead to a very efficient algorithm. Therefore, we believe a good research direction would be to investigate whether XHSTT knowledge can be integrated directly within a maxSAT solver. This may concern information about the constraints (e.g. domain-specific propagation) or special procedures to handle cardinality

constraints natively. As pointed out in [AN14], the *pigeon hole problem* might arise during the search in timetabling problems, which involves proving that n pigeons cannot fit into $n\text{-}1$ holes, assuming no hole can be occupied by more than one pigeon. In the context of XHSTT, an equivalent situation would be to state that n events with total duration d, all of which share a common resource, cannot be placed in $d\text{-}1$ times. For SAT this is a problem because there is no polynomial-size proof of its unsatisfiability [Hak95], even though it is obvious when posed in natural language. Thus, developing specialized SAT techniques for XHSTT might be very beneficial.

In addition to the above, another important issue is the modeling of resource assignments. We have provided a model for a specific case, but not for the general case. If this is done in a straightforward manner, apart from possibly drastically increasing the number of variables in our model, symmetries may easily be introduced, negatively impacting the solution process. Therefore, innovative modeling approaches are to be studied to address these concerns.

List of Tables

List of Algorithms

Bibliography

[ADG] Carlos Ansótegui, Frédéric Didier, and Joel Gabàs. Exploiting the structure of unsatisfiable cores in maxSAT. In *Proceedings of IJCAI-15*, pages 283–289.

[AN14] Roberto Javier Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and maxSAT. *Annals of Operations Research*, 218(1):71–91, 2014.

[ANOR] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Proceedings of SAT-09*, pages 167–180.

[AÖK15] Leena N. Ahmed, Ender Özcan, and Ahmed Kheiri. Solving high school timetabling problems worldwide using selection hyper-heuristics. *Expert Systems with Applications*, 42(13):5463–5471, 2015.

[BBa] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Proceedings of CP-03*, pages 108–122.

[BBb] Edmund K Burke and Yuri Bykov. A late acceptance strategy in hill-climbing for exam timetabling problems. In *Proceedings of PATAT-16*.

[BFT+12a] Samuel S. Brito, George H. G. Fonseca, Túlio A. M. Toffolo, Haroldo G. Santos, and Marcone J. F. Souza. A SA-ILS approach for the high school timetabling problem. *Electronic Notes in Discrete Mathematics*, 39:169–176, 2012.

[BFT+12b] Samuel S. Brito, George H. G. Fonseca, Túlio A. M. Toffolo, Haroldo G. Santos, and Marcone J. F. Souza. A SA-VNS approach for the high school timetabling problem. *Electronic Notes in Discrete Mathematics*, 39:169–176, 2012.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BP] Nikolaj Bjørner and Anh-Dung Phan. νZ-Maximal Satisfaction with Z3. In *Proceedings of SCSS-14*.

[DdAB14] Árton P. Dorneles, Olinto César Bassi de Araujo, and Luciana S. Buriol. A fix-and-optimize heuristic for the high school timetabling problem. *Computers & Operations Research*, 52:29–38, 2014.

[dFST+16] George Henrique Godim da Fonseca, Haroldo Gambini Santos, Túlio Ângelo Machado Toffolo, Samuel Souza Brito, and Marcone Jamilson Freitas Souza. GOAL solver: a hybrid local search based solver for high school timetabling. *Annals of Operations Research*, 239(1):77–97, 2016.

[DM] Emir Demirović and Nysret Musliu. Modeling high school timetabling as partial weighted maxSAT. *LaSh 2014: The 4th Workshop on Logic and Search (a SAT / ICLP workshop at FLoC 2014)*.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[EIS75] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of time table and multi-commodity flow problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 184–193, 1975.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[FS14] George H. G. Fonseca and Haroldo G. Santos. Variable neighborhood search based algorithms for high school timetabling. *Computers & Operations Research*, 52:203–208, 2014.

[FSC16a] George H. G. Fonseca, Haroldo G. Santos, and Eduardo G. Carrano. Integrating matheuristics and metaheuristics for timetabling. *Computers & Operations Research*, 74:108–117, 2016.

[FSC16b] George H. G. Fonseca, Haroldo G. Santos, and Eduardo G. Carrano. Late acceptance hill-climbing for high school timetabling. *Journal of Scheduling*, 19(4):453–465, 2016.

[FSCS] George HG Fonseca, Haroldo G Santos, Eduardo G Carrano, and Thomas JR Stidsen. Modelling and solving university course timetabling problems through XHSTT. In *Proceedings of PATAT-16*, pages 127–138.

[Hak95] Armin Haken. Counting bottlenecks to show monotone P <=> NP. In *36th Annual Symposium on Foundations of Computer Science*, pages 36–40, 1995.

[ITC] International timetabling competition 2011. http://www.utwente.nl/ctit/hstt/itc2011/welcome/. Accessed: 9-12-2016.

[kina] The KHE high school timetabling engine. http://sydney.edu.au/engineering/it/ Accessed: 06-12-16.

[Kinb] Jeffrey Kingston. KHE14: An algorithm for high school timetabling. In *Proceedings of PATAT-14*, pages 498–501.

[Kinc] Jeffrey H. Kingston. Hierarchical timetable construction. In *Proceedings of PATAT-06, Revised Selected Papers*, pages 294–307.

[Kin12] Jeffrey H Kingston. Timetable construction: the algorithms and complexity perspective. *Annals of Operations Research*, pages 1–11, 2012.

[Kir84] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984.

[KK16] Ahmed Kheiri and Ed Keedwell. A hidden markov model approach to the problem of heuristic selection in hyper-heuristics with a case study in high school timetabling problems. *Evolutionary Computation*, 2016.

[KKMS15] Lucas Kahlert, Franziska Krüger, Norbert Manthey, and Aaron Stephan. Riss solver framework v5. 05. *SAT-Race*, 2015.

[KOP] Ahmed Kheiri, Ender Ozcan, and Andrew J Parkes. HySST: hyper-heuristic search strategies and timetabling. In *Proceedings of PATAT-12*, pages 497–499.

[KSS15] Simon Kristiansen, Matias Sørensen, and Thomas R. Stidsen. Integer programming for the generalized high school timetabling problem. *Journal of Scheduling*, 18(4):377–392, 2015.

[LBP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[MH97] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[MJML] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for maxSAT. In *Proceedings of CP-14*, pages 531–548.

[MML] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: a modular maxSAT solver. In *Proceedings of SAT-14*, pages 438–445.

[MO] Carol Meyers and James B Orlin. Very large-scale neighborhood search techniques in timetabling problems. In *Proceedings of PATAT-06, Revised Selected Papers*, pages 24–39. Springer.

[PAD+12] Gerhard Post, Samad Ahmadi, Sophia Daskalaki, JeffreyH. Kingston, Jari Kyngas, Cimmo Nurmi, and David Ranson. An XML format for benchmarks in high school timetabling. *Annals of Operations Research*, 194(1):385–397, 2012.

[PDGK+13] Gerhard Post, Luca Di Gaspero, JeffreyH. Kingston, Barry McCollum, and Andrea Schaerf. The third international timetabling competition. *Annals of Operations Research*, pages 1–7, 2013.

[Pil14] Nelishia Pillay. A survey of school timetabling research. *Annals of Operations Research*, 218(1):261–293, 2014.

[PKA+14] Gerhard Post, Jeffrey H. Kingston, Samad Ahmadi, Sophia Daskalaki, Christos Gogos, Jari Kyngäs, Cimmo Nurmi, Nysret Musliu, Nelishia Pillay, Haroldo Santos, and Andrea Schaerf. XHSTT: an XML archive for high school timetabling problems in different countries. *Annals of Operations Research*, 218(1):295–301, 2014.

[Sch99] Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[SD14] Matias Sørensen and Florian H. W. Dahms. A two-stage decomposition of high school timetabling applied to cases in denmark. *Computers & Operations Research*, 43:36–49, 2014.

[Sha] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of CP-98*, pages 417–431. Springer.

[Sin] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of CP-05*, pages 827–831. Springer.

[SKS] Matias Sørensen, Simon Kristiansen, and Thomas R Stidsen. International timetabling competition 2011: An adaptive large neighborhood search algorithm. In *Proceedings of PATAT-12*.

[SLM09] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [BHvMW09], pages 131–153.

[Sør13a] Matias Sørensen. Decomposing the generalized high school timetabling problem. In *Timetabling at High Schools, PhD thesis*, pages 119–136. Department of Management Engineering, Technical University of Denmark, 2013.

[Sør13b] Matias Sørensen. A matheuristic for high school timetabling. In *Timetabling at High Schools, PhD thesis*, pages 137–153. Department of Management Engineering, Technical University of Denmark, 2013.

[SS] Matias Sørensen and Thomas R Stidsen. Hybridizing integer programming and metaheuristics for solving high school timetabling. In *Proceedings of PATAT-14*, pages 557–560.