FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

A Timeslot-Based Heuristic Approach to Construct High-School Timetables

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Michael Pimmer
Matrikelnummer 0253076

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.-Prof. Dr. María Belén Melián Batista, Universidad de La Laguna, Spain

Wien, 30.11.2010

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Abstract

This master thesis describes an algorithm for creating high-school timetables. In the history of high-school timetabling, interchangeable and common instances were missing. Most scientists restricted their work to basic problem definitions, or to instances gathered from schools nearby. In 2007, the School Benchmarking Project was launched to correct this issue. A common XML file format and an evaluation function was defined, and scientists of various countries contributed real-world instances. However, the format, evaluation function and instances are still under development. We are going to test our algorithm using these instances.

Our approach is to pick a non-full timeslot. We will then grade the favorability of all meetings that can be held in this timeslot: All hard constraints and the goal of completely filling the timetable have to be considered, as well as the soft-constraints, to obtain valid solutions with low penalties. This information - the meetings and their grades, as well as which meetings can be held simultaneously - is represented as a weighted graph. We perform a heurisitc maximum weight clique search on this graph. If there are meetings with open roles, one out of a certain set of resources shall be assigned, e.g. any english teacher. This introduces an additional constraint to the maximum-weight clique search. Before assigning the found clique - a hopefully favorable set of meetings - to the given timeslot, all open roles are closed with a maximum-cardinalty maximum weight matching.

On top of this basic procedure, we are going to implement higher-level solving strategies. We try to improve the quality of the timteable by partly refilling it or by reassigning resources and meetings that cause penalty. There are various parameters that influence the grading of constraints to construct the weighted graph for the clique search. As the suitability of the parameters depends on the given instance, we are going to use a hill climbing procedure to search for appropriate values for these parameters.

This work describes the given approach in detail. The advantages and disadvantages of this timeslot-based algorithm will be discussed by evaluating it using the before-mentioned instances. We are also going to present and discuss the results, which are the first results published based on the School Benchmarking Project, as well as specifics of the format and instances.

Kurzfassung

Die vorliegende Masterarbeit beschreibt einen Algorithmus zur Erstellung von Schul-Stundenplänen. Austauschbare, internationale Problem-Instanzen waren in der Geschichte des “high-school timetabling” oder “Berechnung von Schul-Stundenplänen” genannten Gebietes kaum vorhanden. Viele Wissenschaftler beschränkten ihre Arbeit auf simplifizierte Problemdefinitionen oder Instanzen aus Schulen der Umgebung. 2007 wurde das “School Benchmarking Project” ins Leben gerufen, um diesen Mißstand zu beseitigen. Ein einheitliches XML-Format sowie eine klare Evaluierungsfunktion wurden definiert, und dank der Zusammenarbeit von Wissenschaftern verschiedenster Ländern sind heute eben diese internationalen, austauschbaren Instanzen vorhanden. Diese Instanzen, die wir für die Evaluierung unseres Algorithmus verwenden, sind teilweise noch ungetestet und in Entwicklung.

Im von uns verfolgten Ansatz werden wiederholt einzelne Stunden mit Unterrichtseinheiten (Meetings) gefüllt. Zuerst werden alle in der jeweiligen Stunde vorhandenen Meetings anhand ihrer Dringlichkeit und der vorhandenen Randbedingungen, wie z.B. der Vermeidung von Freistunden, gewichtet. Dabei werden sowohl verpflichtenden Randbedingungen (hard-constraints), und die Notwendigkeit einen kompletten Stundenplan zu erstellen, als auch Wünsche (soft-constraints) berücksichtigt. Die vorhandenen und gewichteten Meetings werden in einem gewichteten Graph dargestellt, bei dem Meetings verbunden sind wenn diese gleichzeitig abgehalten werden können. Auf diesem Graph führen wir eine heuristische Suche nach einer Clique mit dem maximalen Gewicht durch. Diese stellt eine Menge von Meetings dar, deren Zuweisung zu der gegebenen Stunde dringlich und wünschenswert ist. Bei der Cliquen-Suche müssen gegebenenfalls auch offene Rollen berücksichtigt werden, die eine weitere Randbedingung darstellen: Offene Rollen bedeuten, dass eine beliebige Ressource aus einer Menge an Ressourcen - zB ein beliebiger Englischlehrer - einem Meeting zugeordnet werden muss. Beim Zuweisen der Meetings zu einer Unterrichtsstunde werden alle offenen Rollen mit einem maximum-cardinalty maximum weight matching - derjenigen größten Paarung die das höchste Gewicht aufweist - geschlossen.
Auf diese Basisprozedur setzen generellere Algorithmen auf. Der Stundenplan wird teilweise neu gefüllt, indem beispielsweise problematische Ressourcen oder Meetings neu zugeteilt werden. Gute Werte für die Parameter der Bewertungsfunktion von Meetings sind von der jeweiligen Instanz abhängig. Um geeignete Parameter zu finden wird ein “hill climbing“-Algorithmus angewendet.

In dieser Arbeit wird der Unterrichtsstunden-basierte Algorithmus im Detail vorgestellt. Die Vor- und Nachteile dieses Ansatzes werden anhand der Evaluierung an den zuvor erwähnten Instanzen präsentiert. Neben den Resultaten, welche die ersten veröffentlichten Ergebnisse dieser Instanzen darstellen, werden auch länder- und instanzabhängige Spezifika sowie das XML-Format im Allgmeinen diskutiert.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The seemingly ordinary task of creating a high-school timetable causes disputes and dissatisfaction in almost every school around the world. Either the teachers and students are unpleased with their timetable, or the person responsible for the creation - often an honorable natural science teacher - is close to desperation in view of the multitude of wishes and requirements he should consider. Altough nowadays this work is partly supported by computer programs, many schools still create their timetable manually. Dependent on the size of the school, the creation of a timetable can take everything from a troublesome meeting just before or even after the new school year starts up to a team working full-time for weeks, often during the summer holidays.

Hence, it does not surprise that the problem of *High-School Timetabling*, also called *Class-Teacher Timetabling Problem* (CTTP) has been frequently revisited within the last decades - not only by the before mentioned honorable natural science teachers, but also by the scientific world. Despite the multitude of published articles, there was little cooperation between scientists of different countries. The reason is that the requirements of institutions and countries vary drastically. This often makes tailormade algorithms that work perfectly for specific institutions completely useless for most other schools.

The basic task of high-school timetabling is to assign class-teacher meetings to rooms and timeslots of a weekly schedule (the terminology is explained in the glossary, section 1.1). Aside from this basic task, the specific requirements and problems vary broadly: Some schools do not pre-define teachers to meetings. Instead, the teachers have to be assigned to the meetings in a way that one teacher should always teach the same subject to a school-class, considering a specific workload (total number of working hours) each teacher has to obtain. Other schools have meetings of several hours which should be split to sub-meetings of certain allowed durations. In Italy, one school requires that certain timeslots contain at least one out of a group of meetings. A finnish schools requires 39 of the 40 timeslots a room is available to be filled - with meetings of varying duration, and of course no meeting is allowed to overlap two days.

Because of this diversity, much scientific work was either limited to a very basic problem definition or to specific institutions close to where the author lives, ignoring demands of other countries. Some efforts were made to introduce common instances (an instance is a problem-definition for a specific schol and year) - for example the timetable specification language TTL [20], or finnish[1] and brazil instances [30]. None of theses projects was accepted broadly in the scientific world, and mostly, only instances of one country were provided.

Great ideas were introduced, but much practical work up to now is not compareable. Scientists had to limit themselves to country-specific instances, because the instances of different countries were available in different formats, and also the evaluation-functions for solutions differed. Some authors created their instances synthetically [25][12], or left out parts of the problem [15][23][35]. Therefore, most practical work is only of limited general relevance, and often not comparable. Instead, the most interesting and useful papers are rather of theoretical nature.
In 2007, the School Benchmarking Project was launched to correct this issue. A general file format was introduced, and scientists of various countries contributed instances of nearby schools and earlier work. We use these instances (described in detail in chapter 2) for our work, altough the file format and some instances are still under development. During our work, we stumbled upon a range of sloppy or errorous definitions, i.e. one instance is even unsolvable. Two month after we started the implementation, fundamental changes - which we do not fully endorse - were introduced with version 9 of the file format. The problems we encountered, as well as comments and suggestions to the format, can be found in section 2.6.
Considering the complexity of the problem, the variety of constraints, originating countries and previous formats, such errors are perfectly understandable in the initial phase of this project. The benefit of helping spread the instances by contributing results and some suggestions clearly outweights the drawbacks caused by changes and errors in the format and instances.

The approaches of computationally creating high-school timetables are as manifold as the requirements and contraints. Virtually every problem solving technique of modern computer science has been applied to timetabling, partly to real-world problems, partly to cut-down basic instances. The methods that are of influence to our approach are described in section 1.4.
Most practival relevant software-tools use any kind of a constructive heuristic algorithm in combination with backtracking or local search techniques. They usually pick the most urgent meeting and assign this meeting to the timetable, afterwards picking the next most urgent of the remaining meetings and so on. A meeting is urgent when the room it has to be in is almost fully booked, or the teacher holding the meeting has only few free timeslots left regarding his pending meetings.

The algorithm we present in this article (see chapter 3) can also be considered a constructive heuristic algorithm. The difference to the before mentioned technique is the following: We first chose a non-full timeslot. Then, all pending meetings will be evaluated (graded) according to their urgency. Afterwards, we construct a so-called constraint graph, which represents which

[1] `http://www.samk.fi/sttp`

meetings can be held together in this timeslot. Within this graph we search for the best-graded (most urgent) set of meetings. We assign this set to the current timeslot, and go on with the next timeslot. As far as we know, this approach has not yet been implemented.

Altough our algorithm is a general one which can be applied to all instances of the School Benchmarking Project, we do not obtain valid solutions for all instances (which up to now no algorithm does), and the quality of the obtained solutions varies. We present and discuss our results in chapter 4, and analyze the advantages and disadvantages of our approach.

This first chapter starts with a basic glossary in section 1.1. Then, we formally define the problem of high-school timetabling in section 1.2, followed by describing the complexity of this problem in section 1.3. In the last part, section 1.4, work that is related to our algorithm will be presented.

1.1 Glossary

The terminology we introduce here will be defined more exactly in the formal definition (chapter 1.2).

Resources are normally school-classes, teachers or rooms. Also other resource-types are possible. Resources can be assigned to *meetings*.

Meetings normally represent reunions of school-classes and teachers that take place in rooms. As these objects are abstracted to the term *resource*, a meeting will just require a set of resources. Assigning meetings to the timetable is the main scheduling task. A set of meetings can be united to a *course*.

Courses usually are a combination of a school-class and a subject. A course consists of a set of meetings, which not necessarily require the same resources: In example, different meetings of a course can take place in different rooms.

Instance As *instance* we understand a problem-definition of a specific school and year. It normally describes the existing teachers, rooms and classes, and the courses/meetings that have to be held. Moreover, it contains the the constraints that should be considered.

Constraints are requirements or wishes that a solution has to (hard-constraints) or should (soft-constraints) fulfill respectively. I.e., a teacher does not have to teach more than 6 hours a day, or wishes to not teach on fridays.

Timeslots are the smallest time-object that meetings can be assigned to. Usually timeslots represent one hour and belong to a certain day.

Solution A *solution* is a finished timetable for an instance. A solution is called **valid solution** if it has all necessary meetings assigned in a way that not violates any hard-constraint.

Penalty or Cost A solution is validated with an *evaluation-function*. All constraints are checked for validity: If any hard-constraint is violated, the solution is *infeasible / invalid*. Soft-constraint violations result in a penalty: The penalty of a solution is the sum of all those penalties.

1.2 Formal definition

We limit the definition to the basic problem as it appears in the instances described in chapter 2.1. The notations we use base on formulations and notations used in [35] and [29].
Because of the flexibility of the format, the typical case we describe here may not be appropriate to represent all future instances. For example, the format gives the possibility to use one resource multiple times within one timeslot - which is currently not used by any instance. The detailed definition of the constraints can be found in chapter 2.1.

In the following notations we make use of matrices, ordered sets and functions:

Matrices will be represented by upper case italics marked with accents, i.e bars: $\bar{M}$. Unavailability-matrices (i.e. of meetings) are always marked by tildes: $\tilde{M}$; Matrices that represent (part of) a solution are always marked with a dot: $\dot{S}$.

Sets are upper case italics without accents, i.e. X, and elements of sets lowercase italics x. The set of all Booleans is defined by $\mathbb{B} := \{true, false\}$. For natural numbers we use **N** to include zero, and $\mathbb{N}$ to not include zero. $\mathcal{P}(X)$ denotes the **powerset** of a set X.

Functions are of the kind F_S, where F is the name of its range, and S of its domain.

Timeslots and Time groups

- T: finite set of **T**imeslots
- D: finite set of time groups (which are often **D**ays)
- $D_T : D \to T$, where $D_T(d)$ denotes the set of all timeslots that belong to time group d
- $T_D : T \to D$, where $T_D(t)$ denotes the set of all time groups that timeslot t belongs to

Resources and Resource groups

- R: finite set of **R**esources
- $\tilde{R}_{|R|\times|T|}$: Resource-unavailability: $\tilde{R}_{rt} = \begin{cases} 1 & \text{if resource } r \text{ available in timeslot } t \\ 0 & \text{otherwise} \end{cases}$
- G: finite set of resource **G**roups
- $G_R : G \to R$, where $G_R(g)$ denotes the set of resources that belong to resource group g

Meetings

- M: finite set of **M**eetings
- $\bar{A}_{|M|}$: **A**mount-matrix of timeslots each meeting has to be assigned to: $\bar{A}_m$ denotes the total number of timeslots that meeting m has to be assigned to
- $M_L : M \rightarrow L$ where $L \in \mathcal{P}(\mathbb{N})$: Allowed lengths (durations) of meetings
  $M_L(m)$ denotes the set of allowed lengths of meeting m
  The length or duration of a meeting is the number of subsequent timeslots a meeting has to occupy when it is assigned. This entails that the timeslots a meeting is assigned to do not necessarily have to be subsequent.
- $\bar{M}_{|M| \times \max(A)}$: Required amount of meeting-lengths
  $\bar{M}_{ml} = \begin{cases} x > 0 & \text{if meeting } m \text{ has to have exactly } x \text{ assignments of length } l \\ 0 & \text{if meeting } m \text{ does not constrain number of assignments of length } l \end{cases}$
- $\tilde{M}_{|M| \times |T| \times \max(A)}$: Meeting-unavailability for lengths (durations), where
  $\tilde{M}_{mtl} = \begin{cases} 1 & \text{if start holding meeting } m \text{ with length } l \text{ is allowed in timeslot } t \\ 0 & \text{otherwise} \end{cases}$
- $M_R : M \rightarrow R$, where $M_R(m)$ denotes the set of resources that meeting m requires
- $\mathsf{OpenRole} : M \times G \times \mathbb{N} \rightarrow \mathbb{B}$: Open roles (resource groups) of meetings, where
  $\mathsf{OpenRole}(m, g, i) = \begin{cases} \mathit{true} & \text{if meeting } m \text{ requires a resource of group } g \text{ with ID } i \\ \mathit{false} & \text{otherwise} \end{cases}$
  In each timeslot this meeting is assigned to, a resource out of the resource group g has to be assigned to this meeting, see section 1.2 for details. Open resource groups are additionally identified by an ID, to enable assigning a resource to a specific open resource group.
  To avoid having duplicate IDs i within a meeting, the following condition has to hold:
  $\forall m \in M : \forall (i, g_1) \mid \mathsf{OpenRole}_{m,g_1,i} = \mathsf{true}$: *// For every ID of an open role of a meeting*
  $\nexists g_2 \in (G \setminus g_1) \mid \mathsf{OpenRole}_{m,g_2,i} = \mathsf{true}$ *// No other open role of this meeting has the same ID*

  The maxmimum value for IDs can be limited to $|R| + 1$: If a meeting requires more resources with its open roles than there are available, no valid solution exists.

Solution

A solution can be represented as follows:

- $\dot{S}_{|T| \times |M|}$: **S**tart-slots of meetings, where $\dot{S}_{tm} = l$ denotes that in timeslot t meeting m starts with length (duration) l.

- $\dot{T}_{|R|\times|T|\times|M|}$: Resource-assignments to meetings in timeslots
  $$\dot{T}_{rtm} = i = \begin{cases} 0 & \text{if resource } r \text{ is assigned to meeting } m \text{ in timeslot } t\text{: } r \in M_R(m) \\ > 0 & \text{if resource } r \text{ fills open resource group } g \text{ with ID } i \\ -1 & \text{if resource } r \text{ is not assigned to meeting } m \text{ in timeslot } t \end{cases}$$

A solution is usually called **valid** if it respects all conditions described above, and furthermore fulfills the following criteria:

- All meetings are assigned to as many timeslots as required:
  $\forall m \in M : \sum_{t=0}^{t=|T|} \dot{S}_{tm} = \bar{A}_m$
- No resource is assigned to more than one meeting in any timeslot:
  $\forall (r,t) \in R \times T : \sum_{m=0}^{|M|} (\dot{T}_{rtm} > -1) <= 1$
- All required resources and open resource groups of meetings are respected:
  $\forall (t_{i,\ldots,i+l-1}, m) \mid \dot{S}_{im} = l > 0 :$ *// In every timeslot a meeting is assigned to:*
  $\quad \forall r \in M_R(m) : \dot{T}_{rtm} = 0$ *// all its resource-requirements are respected*
  $\quad \forall (g,i) \mid \mathsf{OpenRole}(m,g,i) = \mathsf{true} :$ *// for every open resource group of the meeting:*
  $\quad\quad \exists r \in (G_R(g) \setminus M_R(m)) \mid \dot{T}_{rtm} = i$ *// a resource of this resource group is assigned*
- All resource-unavailabilities are respected:
  $\forall (r,t) \mid \tilde{R}_{rt} = 0 : \sum_{m=0}^{|M|} \dot{T}_{rtm} = 0$
- All applied meeting-lengths are allowed:
  $\forall (m,l) \mid (\dot{S}_{tm} > 0) : l \in M_L(m)$
- All required meeting-lengths are fulfilled:
  $\forall (m,l,x) \mid \bar{M}_{ml} = x, x > 0 : \sum_{t=0}^{|T|} (\dot{S}_{tm} = l) = x$
- All meeting-unavailabilities for durations are respected:
  $\forall (m,t,l) \mid \tilde{M}_{mtl} = 0 : S_{tm} \neq l$

1.3 Complexity

The basic Class-Teacher Timetabling Problem (CTTP) as described by Gotlieb [14] defines that each meeting has exactly one teacher and one school-class, and shall be assigned to a certain amount of timeslots. This problem is known to be NP-hard if there are any teacher/room/student-unavailabilities [11], and can be solved in polynomial time when no unavailabilities exist. This was shown in 1971 by Dominique de Werra [9], using flow algorithms.

Relaxing the restrictive definition of meetings and adding some common constrains leads to a range of NP-complete subproblems, as shown by Kingston [20] in 1996 for five cases. Willemen [35] extended this work in 2002, and proofed NP-completeness in seven independent cases. Page 51 of his thesis gives a summary of the seven cases.

We will briefly describe the NP-complete problems that were most challenging during the creation of our algorithm, and mention the instances where the respective problem is most relevant.

Assigning Meetings to Timeslots As mentioned before, this basic timetabling task is NP-complete as soon as unavailabilities are introduced. It is relevant (altough not necessarily NP-complete) in every timetabling instance.

Assigning tight Resources to Timeslots This problem is equivalent to the bin packing problem, which was proved to be NP-complete by Garey and Johnson [16] in 1979. Resources are assigned to meetings of varying duration (bin packing: items of different size). They have to be assigned to the days (bin-packing: bins), so that all total resource-workload is considered (all items are packed into any bin). We further describe bin packing of resources in section 3.1. This concerns resources of all types, and is contained in every instance that has differing event-durations, but especially is a bottleneck and challenge in the finnish instances.

Assigning Teachers to Meetings This problem also is equivalent to bin packing. Given meetings that do not have teachers preassigned, suitable teachers should be assigned in a way that their total number of meeting-durations does not exceed their desired/required workload. This problem is one of the main challenges in the australian instances.

Spreading Meetings over Days NP-completeness is introduced when meetings of a course should be split over different days while considering resource unavailabilities. Possible sources are the clusterBusyTimes-, limitBusyTimes and spreadEvents-constraint, so this is relevant for all given instances.

Another source of NP-complete subproblems is assigning students to subject groups with a given capacity. This again is bin-packing equivalent, and solvable in polynomial time if no capacity constraint is given, as shown by Willemen [35].

1.4 Related work

There exist various solving methods: Graph-algorithms as decomposition, graph-coloring or flow-algorithms, Contraint-Satisfaction Programming, neural networks, Mixed and Integer Linear Programming, Logics and SAT solving, as well as constructive and Meta-Heuristics. We will present some papers that are similar to our approach or of other influence or interest. For more extensive reviews of existing methods, the reader is refered to well-known surveys [6][31] and to the international conferences and books [7][5] of *Practice And Theory of Automated Timetabling*

(PATAT)[2] as well as the EURO working group on automated timetabling *EURO-WATT*[3].

Our approach presented in chapter 3 can be considered a depth-first search heuristic using graph-algorithms.
Schmidt and Ströhlein [32] described a similar approach in 1979, and point out the similarity to graph-coloring:
They consider a timetabling-problem of P, M and H as the sets of participants, meetings and hours (timeslots). Participants of meetings can be teachers T and schoolclasses C. Then, the following instance is created: There are 3 hours, 7 participants (3 teachers and 4 schoolclasses), and 9 meetings given. Teacher 4 is unavailable in hour 1, teacher 1 in hour 2, and teacher 2 in hour 3. Figure 1.1 represents the problem as a set of graphs, one for each timeslot: The nodes are the classes and the teachers. teachers (T) and schoolclasses (C) are connected if they can have a meeting in this timeslot and are both available. The bold marked edges represent a valid solution.

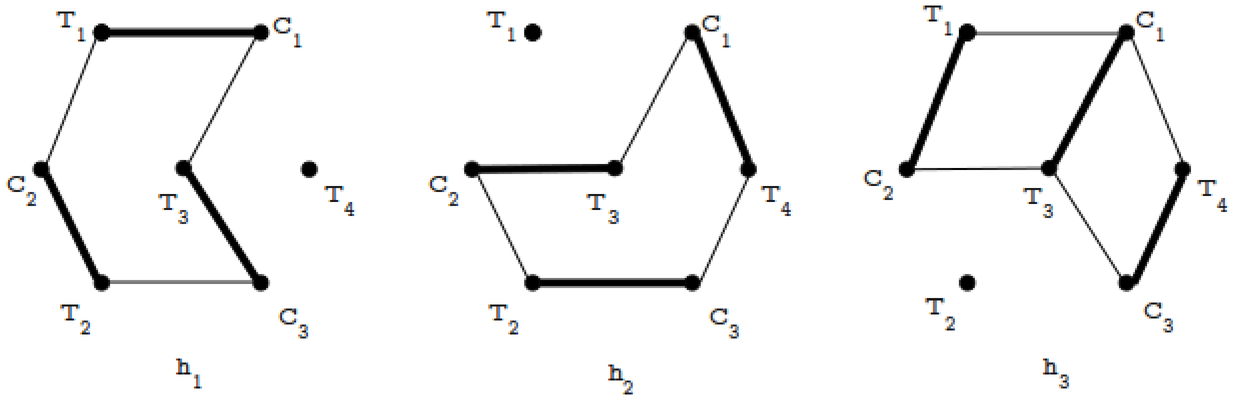


Figure 1.1: Representation of the CTTP as a set of graphs; example taken from Schmidt and Ströhlein [32]

The solution for a timeslot is a matching. The solution of the whole problem is an appropriate matching for each timeslot that sums up to cover all necessary meetings.
The similarity to edge-coloring is shown in figure 1.2: The left part presents the same problem as described above. Edges m_1 to m_7 represent the meetings. The unavailabilities have to be introduced by forbidding some colors for certain edges. The right part presents the same solution as in figure 1.1

To directly integrate unavailabilities into the graph, the problem can be represented by a **vertex coloring problem**. In this case, all hours and meetings are introduced as nodes. First, all pairs of different hours get connected. Second, all meetings are connected with the hours where they are *not* available. Third, all meetings with resource-conflicts get connected. Figure 1.3 is the equivalent vertex-coloring problem (left) and the solution (right).

Converting CTTP-Constraints into a graph-coloring problem is challenging and may not be possible for all instances. To represent rooms or linked meetings as edge-coloring, hypergraphs

[2]http://www.asap.cs.nott.ac.uk/patat/patat-index.shtml
[3]http://www.asap.cs.nott.ac.uk/watt/

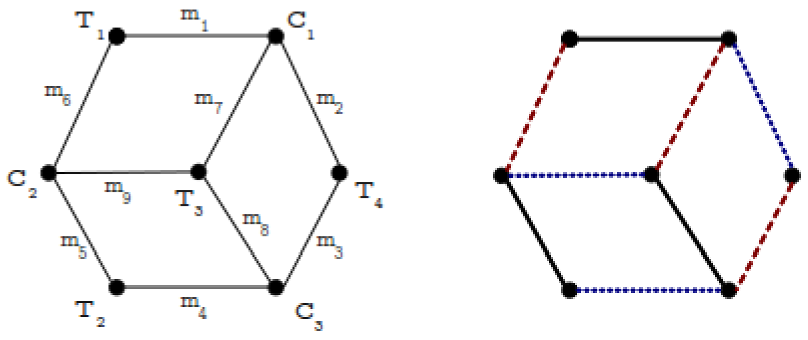


Figure 1.2: Problem-instance as edge-coloring (left) and solution (right) *The color (style) of the edges represents to which hour they are assigned. Hour 1: black (solid), hour 2: blue (dotted), hour 3: red (dashed).*

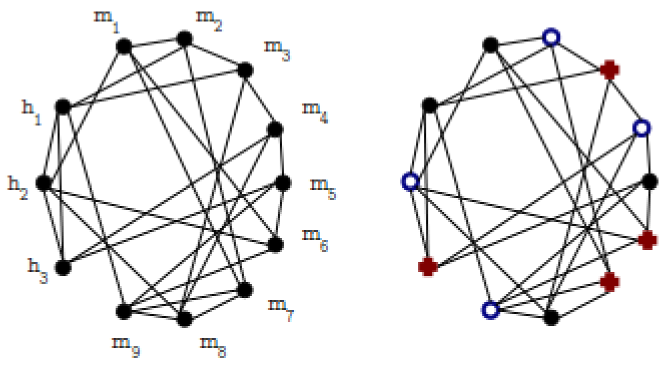


Figure 1.3: Problem-instance as vertex-coloring (left) and solution (right) *The color (style) of the nodes represents to which hour they are assigned. Hour 1: black (bullet), hour 2: blue (circles), hour 3: red (crosses)*

are necessary. Problems arise with meetings of longer duration, especially if it is not determined how a course should be split exactly (see chapter 2.1 for the particular constraint). In the case of vertex-coloring, not only the weight and possible colors of edges would change during the solving-procedure, but also the existence of vertices.

There are still efforts to solve CTTPs via graph-coloring, as described in [3]. Results are competitive, but not as good as results achieved some years ago [34] with a GRASP, refining filled timetables with a tabu search. Tabu search algorithms in general seem to be very suitable for solving CTTPs, as they help escaping from local minima and so to diversify the search.

Because of the absence of common benchmark-instances it is hard to tell which solving method is the most suitable for the field of class-teacher timetabling. Perhaps the most suitable solving method depends on the characteristics of the instance.
For sure one of the most successful methods nowadays are construction heuristics in combination with backtracking. These heuristics also use depth-first search. They do not take timeslots but meetings as basis, as described in the dissertation of Michael Marte [24]: A meeting is

chosen and assigned to the timetable, so that no hard-constraint is violated. If for any pending meeting there is no timeslot available anymore, the conflict-resolution or backtracking starts: One or more meetings are removed from the timetable, so that the before not schedulable meeting can be assigned to a timeslot. The meetings that are deleted are chosen by the density and importance of their constraints. Also when adding meetings, those with more and denser constraints are prefered.
The general approach described can be implemented in different ways, as directly implemented construction heuristic using backtracking, or - as in the mentioned dissertation - as Constraint Satisfaction Programming.
One of the most successful commercial software for school-timetabling is *GP-Untis*: *"14,000 users of all schooltypes in over 80 countries are using gp-Untis."* [15] This software uses a construction-heuristic as described above, optimizing the filled timetable with local search based on neighborhood-swaps.
A free alternative, *FET* - Free Educational Timetabling (see chapter 2), also uses the same principle, based on the already mentioned dissertation [24].

The most relevant works on complexity issues, already mentioned in section 1.3, are [20] and [35]. We highly recommend reading them to anyone interested in the field of CTTP. They analyze practical problems and deduct NP-completeness for 5 respectively 7 sub-problems of the CTTP. The mentioned sub-problems are not just of theoretical nature, but mostly present practical, real challenges and bottlenecks when creating high-school timetables.

Finally, we refer to [28], in which the XML file format of the instances we are going to use is introduced. An updated, extended paper was released in 2010 [27]. As the project is evolving, it is advisable to also check the prject website[4] for changed instances and further information. As noted, the website `http://opt-kd.cse.dmu.ac.uk/www/` contains a brief overview of the topic, and may some day contain a comparison of existing results.

[4] `http://www.utwente.nl/ctit/hstt/`

CHAPTER 2

Instances

Two different sources for instances were of question:

Free Educational Timetabling (FET) The first is **F**ree **E**ducational **T**imetabling[1]. On its website, various different instances from distinct countries are available. The format is easily understandable and stable in the sense that it is already in use for some years. Unfortunately, up to now it is not considered at all by the scientific world.

School Benchmarking Project (SBP) The second source was the **S**chool **B**enchmarking **P**roject (SBP) [28]. Those instances are the opposite: Still in development and undergoing rather fundamental changes. However, the instances are contributed by scientists from different countries, and are much more likely to play a fundamental role in the future of high-school timetabling.

We chose to use the instances of the School Benchmarking Project. This hopefully helps spreading the usage of the instances, and so getting compareable results of different approaches. Since December 2009, a converter for FET-instances to the XML format of the School Benchmarking Project is available at `http://opt-kd.cse.dmu.ac.uk/www/`.
In this chapter we describe the instances and their constraints. The terminology used up to section 2.3 arises from the official documentation of the SBP, which is available on the project Website[2]. When the instances are parsed, the data is transformed into internal datastructure and objects. To describe the internal data, a different terminology is introduced, starting with Section 2.4. In this thesis we always refer to the terminology, constraints, instances, documentation and evaluation-function of the School Benchmarking Project version 10, released in November 2009.
Up to now there does not exist a complete formal definition of the instances. We therefore only

[1]Free Educational Timetabling (FET), an open-source software to solving high-school timetabling problems `http://www.lalescu.ro/liviu/fet/`
[2]`http://www.utwente.nl/ctit/hstt/`

provide the formal definition of the basic problem, given in chapter 1.2. The constraints are described informally.

2.1 SBP: Official Terminology

The terminology presented here is given by the documentation of the School Benchmarking Project. For a more detailed documentation we refer to the project-website.

We use different terms for similar or equivalent objects to distinguish between different definitions, presented in table 2.1:
- *General Definition* are the terms we use in the glossary in chapter 1.1 and the formal definition in chapter 1.2
- *School-Benchmarking Project* means the official terminology used by this project
- *Internal Datastructure* denotes the names of objects we use in our internal datastructure
Terms that do not change in name and meaning (i.e. timeslots or resources) are not mentioned.

General Definition	School Benchmarking Project	Internal Datastructure
meeting	event	lesson
course	- [1]	session
open role	open role	open resource group
day	time group	time group

Table 2.1: Names of similar/equivalent objects within the distinct terminologies
[1] *The SBP does not explicitly define courses*

Resources

Resources are generalized objects which can be assigned to events. Each resource has a certain **ResourceType**. Recommended resourcetypes are *Room*, *SchoolClass*, *Student* and *Teacher*. Using these recommended types is not obligatory; other types can be created, i.e. the Italy-instance used a type called 'School'. This issue will be discussed in chapter 2.6.
The usage and assignment of resources can be constrained by the resource-constraints described in chapter 2.1.

Events

Events are the equivalent of meetings described in the glossary and the formal definition: They usually describe schoolclass-teacher-meetings that take place in rooms. Assigning them to the timetable is the main scheduling task. Events have a duration, which is the total number of timeslots they will occupy. If the duration is higher than 1, events can be split into subevents which do not have to be assigned consecutively. In this case, an event would rather represent a course.

Events have resources (as teacher, room etc.) assigned. If any resource out of a specific set of resources should be assigned (i.e. any English-teacher, or any Gym-room), this is called an "open role" (see formal definition, chapter 1.2). Normally, all open roles have to be filled to obtain a valid solution.
Sets of events can be grouped to form an event group.

Event groups and Courses

Event groups exist only to organize (group) events. They do not have any implicit meaning, and mainly exist for applying a constraint to a set (group) of events.
Courses have exactly the same meaning and appliance. They are just a synonym for event groups.

Constraints

Different to many other definitions, all constraints mentioned can be either hard (required) or soft (not required). We will briefly introduce all existing constraints here. *"All constraint start with a verb, and describe how the constraint is not violated."* [28].
For a detailed description or details of the evaluation we refer to chapter 3.1, or to the official documentation.

Scheduling-Constraints:

AssignResourceConstraint For all events constrained, assign a resource to the open role (also defined in the formal definition in chapter 1.2, see meetings $\rightarrow$ $\mathsf{OpenRole}$)

AssignTimeConstraint Assign each mentioned event to a timeslot

SplitEventsConstraint Forbid or penalize the splitting of events

Resource-Constraints:

AvoidClashesConstraint Do not assign resources more than once to any timeslot

AvoidUnavailableTimesConstraint Consider unavailabile timeslots of resources

LimitIdleTimesConstraint The idle times of a resource can be constrained to lie between a minimum and a maximum

ClusterBusyTimesConstraint Limit the number of time groups a resource should be used (assigned at least once). Usually, the time groups are days: This constraint can i.e. be used to prevent teachers from having (few) classes on many different days.

LimitBusyTimesConstraint Limit the usage (number of timeslots assigned) of a resource within one time group (usually days)

LimitWorkloadConstraint Events can have a *workload* defined - by absence of this definition the workload is the duration: This helps distinguishing between events that require more preparation or corrections (language-classes) or less preparation (i.e. gym-classes). It sometimes is not predefined which teacher gives which lessons (meetings), but the total workload of the teacher is limited: This is defined by this constraint.

Event-Constraints:

DistributeSplitEventsConstraint If an event is allowed to be split, this constraint can adjust how this should be done: Minimum- or maximum-amounts for each (sub-)duration can be defined.

PreferResourcesConstraint Define preferences for resources that open roles should be filled with

PreferTimesConstraint Assigning events to timeslots not mentioned here will lead to penalty or infeasability. The constraint is applied to the start-slot of events: If slot 6 is forbidden, it is legal to assign an event of duration 2 to slot 5.

AvoidSplitAssignmentsConstraint For a set of events, the open role should be filled with the same resource

SpreadEventsConstraint The assignment of events within one or more time groups should lie between the given minimum and maximum (checked/evaluated for each time group mentioned)

LinkEventsConstraint Events should be held at the same time

Cost-Functions (Penalty)

Violating soft-constraints results in a cost (penalty). Each soft-constraint returns either one single deviation or a list of deviations, which represent the times the constraint has been violated. For example, the LimitBusyTimesConstraint returns a list of deviations, with one entry for each time group it is applied to. This deviation(list) is evaluated by a **CostFunction**, which converts the (list of) deviations to a single deviation-value. The existing costcunctions are: SumStep, StepSum, Sum, SumSquares, SquareSum (italic explanations taken from SBP [28]), demonstrated with the deviation-list $[2, 0, 1]$:

SumSteps *The value is the number of strictly positive deviations*
$[2, 0, 1] \rightarrow (1 + 0 + 1) = 2$

StepSum *The value is 1 if one of the deviations is strictly positive, and 0 otherwise*
$[2, 0, 1] \rightarrow 1$

Sum *The value is the sum of the (positive) deviations*
$[2, 0, 1] \rightarrow (2 + 0 + 1) = 3$

SumSquares *The value is the sum of the squared (positive) deviations*
$[2, 0, 1] \rightarrow 2^2 + 0^2 + 1^2 = 5$

SquareSum *The value is the square of the summed (positive) deviations*
$[2, 0, 1] \rightarrow (2 + 0 + 1)^2 = 9$

This single deviation-value is then combined with the weight: $cost = deviation \cdot weight$

2.2 Parsing the Instances

When parsing the XML-format of the instances, several changes have to be performed internally. The given information is merged from the files - the larger one with about 100.000 lines - into the desired datastructure to gain reliable, quickly accessible information. The newly introduced terminology from now on refers to this internal datastructure.

2.3 Resource Groups

Open resource groups are the equivalent of open roles described in section 2.1. A resource group is the set of resources an open resource group (open role) can be filled with. The resources openRoles can be filled with are not only constrained by the given resourcetype. Also, the PreferResourcesConstraint (described in chapter 2.1) can further cut down the possible resources. Resource groups combine the information given by the resourcetype and the PreferResourceConstraint.
Resource groups are used to fill open roles with a correct resource. Also, they are necessary during the clique-search: They avoid finding a clique that requires more resources of a resource group than there are available.
Resource groups are created initially when parsing an instance. For each distinct set of resources an open role can be filled with, a new resource group is created. The "open roles" of the instance are converted to **open resource group** in the internal datastructure.
For each event, it is stored which open resource group it has, and which resource group it "**fills**": Assigning an event and its resources to a timeslot reduces the number of available resources of some resource group. This are the resource group an event fills. Each resource of an event will reduce all resource group it belongs to by one. Also, each open resource group will reduce exactly this open resource group by one, because a resource of this resource group has to be assigned. It can happen that the assigned resource belongs to more than one resource group. This problem is described in section 3.4.

2.4 Sessions

Using events as main objects to schedule has the disadvantage of being slow. Some events should be further split, some obligatory held together, others are already fixed to a specific timeslot. Instead of repeatedly searching all events when looking for available events in a given

timeslot, this information is precalculated: Sessions "group" similar events. Often, a session will represent a course. Grouping events of a course helps grading them more accurately, especially concerning the calculation of the session-urgency described in section 3.1.
An event is added to an existing session if it fulfills the following criteria (always comparing with the events that already belong to the session):

- Has same resource- and open resource group-requirement
- Not preassigned to a timeslot
- None of the event groups it belongs to can make only this event unavailable (keeping the other events of the session available): this is the case for hard-constrained SpreadEvents-constraints and for hard AvoidSplit-constraints.
- The forbidden start-timeslots of each duration the event can possibly have (see PreferTimesConstraint in section 2.1) equal the forbidden start-timeslots of the session.

If any of this criteria is violated, a new session will be created. Otherwise, the event(s) is/are added to the session: It/They will form a new **iteration** of this session. A session consists of one or more iterations, which themselves contain one or more events. Table 2.2 shows an example of a session containing the Mathematics-events of a school-class.

Session 1	
Iteration 1	Event: 1a_mathematics_1
Iteration 2	Event: 1a_mathematics_2
Iteration 3	Event: 1a_mathematics_3

Table 2.2: A session and its iterations

In practice, some events are defined to be obligatory held at the same time: Such events would belong to the same iteration of one session.
Different iterations are allowed to have different durations. This supports representing the reality in an accurate way, because it allows that the meetings of a course have differing durations.
Allowing differing durations is unavoidable if we do not want to create more than one session out of a single event: With SBP version 10, the possibility of splitting events was introduced (described in section 2.1). It can be part of scheduling how those events should be split exactly. An event of duration 5 can i.e. be split to sub-events of duration 2-1-1-1 or 1-1-1-1-1. This forces that within one iteration of a session, different *possible durations* exist.

Having sessions defined this way, one can precalculate which sessions (of a certain duration) and therefore which events are available for a specific timeslot. Another advantage is that the **session-peers** can be precalculated: This are sessions that can be held at the same time without any resource-conflicts. To keep the session-peers up to date, they have to be recalculated for a session in case of assigning or deleting resources from a role that is constrained by a hard AvoidSplit-constraint.

When trying to fill a timeslot, first the available sessions are picked. Then, **lessons** are created out of these sessions.

2.5 Lessons

Lessons are sessions with a fixed duration. For each possible duration of a session, one lesson will be created. If a session consists of more than one iteration, only the best-graded iteration will be instantiated.
The lessons described here are similar to the meetings mentioned in the glossary, but can contain additional information: First, they can still have open resource groups. Second, a lesson described here can contain more than one schoolclass-teacher-meeting, if those meetings have to be held at the same time.
If a hard AvoidSplit-constraint exists and a resource has already been assigned, the lesson inherits this information from the session: The open resource group of the lesson (and event) gets filled. Otherwise the resource group stays open.
A lesson is always created for a specific timeslot. It gets graded according to how urgent and suitable its events and resources are for the given timeslot. If a lesson is chosen to be assigned to a timeslot, we fill all of its open resource group during this assignment-process, which is described in chapter 3.4.

Deepness

The deepness is an approach to determine the size and importance of a lesson. This information is required during the clique-search, see section 3.3. If a lesson contains many teachers and schoolclasses, it will automatically get a higher grade. Considering the deepness helps to not favour such lessons during the clique-search.
The deepness is calculated by summing up the number of rooms and teachers a lesson requires. This is only possible if the instance considers the recommendation of using the resource-types "Teacher" and "Room". Unfortunately, the australian instances already (slightly) disregard the recommended resource-types by using the plural: "Teachers" and "Rooms".

If the resource-types do not contain reliable information, we have to estimate the deepness: Just counting the required resources is torpedoed by some instances including students as resources. Experiments showed that in most cases the total number of soft-constraints the resources of a lesson are constrained with would be a good (but still rough) substitute.

2.6 Discussion

The School Benchmarking Project aims at generally defining high-school timetabling problems. This is of extreme importance for the field of class-teacher timetabling problems, as it is exactly what has been missing for decades: Compareable, interchanged instances that support problem-definitions of various countries, and allow comparing the solutions of different approaches. In this respect, the School Benchmarking Project is one of the most important contributions to this

field.
To be able to support many different problem-definitions and constraints, the formulation of the instances was kept abstract. This generality directly reduces the readability and easy understandability of the instances. It takes some time to get familiar with the signification of the terms and constraints, which are sometimes not easily understandable. The benefit is that the format itself possibly supports a broad variety of applications.

In this chapter, we will describe some problems encountered, and give suggestions of how to improve the definitions. Having not as much insight as the authors of the format, we still hope that some of our suggestions will be of help.
All suggestions were given to the authors of the School Benchmarking Project during the creation of this thesis.

Instances under Construction

The instances are (partly) still under heavy development. At the moment (July 12., 2010), some instances still contain errors or misleading definitions:

- In all finnish instances available, the last two timeslots of each day are part of the time group "Monday". As all teachers have a LimitIdleTimes-constraint, solving this instance without manually correcting the time group-assignments does not make much sense.

- The finnish instances of the SBP differ from their original definition given at `http://samk.fi/sttp`. No valid solution exists for the SBP-instance "FinArtificialSchool", because for all rooms of the group "gr_Rooms_Z", a workload of 140 should be assigned to 20 timeslots without a clash. This error perhaps arose by misunderstanding assigning resource groups (gr_Rooms_Z) to events: Not one resource of this group gets assigned, but all resources.
The Instance "FinSecondary" has differing resource-availabilities, further described in section 2.6.

- The italian instance allows (unpenalized) clashes for a resource called "School". This resource is then constrained by the ClusterBusyTimesConstraint: Each timeslot is defined as a time group which contains exactly this single timeslot. The usage of the resource "School" is fixed to exactly 1 per time group by listing all 24 time groups (which are timeslots in fact) and fixing the minimum- and maximum-usage to exactly 24. Of course this is perfectly legal according to the Constraints. However, the SpreadEventsConstraint can be applied with the same effect, and without the necessity of split-resources:

  - Define each timeslot as its own time group (as done before)
  - Group all events requiring the "School"-resource into one event group
  - Constrain the event group with an SpreadEventsConstraint that has a minimum-assignment of 1 in each timeslot(group)

This proposal was already incorporated by the authors on 16.8.2010, the current instance does not require clash-resources any more.

- The AvoidClashes-constraint is not yet implemented by all instances
- The brasilian instances of version 10 (except instance 1 and 7) were released in autumn 2010, shortly before handing in this work. We include them in our algorithmic tests, altough thorough tests were left out due to lack of time. They derive from the brazil instances used in literature earlier, as [30].

Because of the abstract and general definition, considering all possible peculiarities - as appearing in the Italy-instance - is a lot of programming effort when parsing the instances. The datastructure we use does not support resource-clashes, and currently we do not know of any case where doing so would be necessary. The re-formulation of the italy-instance showed another problem: The algorithm presented in chapter 3 has problems when dealing with a spreadEvents-constraint of a minimum-assignment in only one timeslot, as appears in the new formulation. This is not a disadvantage of the instances but of our approach, and will be discussed in chapter 4.6.

Vague Information

When parsing the instances, much programming effort is necessary to extract reliable information out of the abstractly defined instances. We will present two problems here, and propose changes to improve the definitions.

Courses

As course we understand the combination of a schoolclass and a subject. Semantically, there do not exist courses. The term exists, but its meaning is equivalent to event groups. Currently, there are two ways to define a course:

- Define a set of events which belong to the same event group
- Define one event that has to be split

It is necessary that a course can consist of different events, because there are courses whose events require distinct resources, i.e. have to be held in different rooms.
Allowing splits of events was introduced in version 10. The necessity arises because some schools define the total duration of a course, i.e. 6 hours, but not how it has to be split exactly. There is the possibility to constrain the sub-events to a minimal and maximal duration, or require a certain amount of one duration. This splitting could not be represented with the events defined in version 9.

We think that it would be more natural to split a course into events instead of splitting some events into sub-events. Because of this - and the absence of the usage of a course - we propose the following change:

- Events should never be allowed to split. They again should be defined as in version 9.
- To define the current splitEvents, the term *course* should be introduced
- All constraints for events and event groups should be able to be applied to *courses*
- Splitting of *courses* can be constrained as currently the splitting of events (using the DistributeSplitEventsConstraint and SplitEventsConstraint)

The existing event-constraints that would have to be adapted to be applied to *courses* are: DistributeSplitEventsConstraint, PreferResourcesConstraint, PreferTimesConstraint, AvoidSplitAssignmentsConstraint, SpreadEventsConstraint and LinkEventsConstraint.
According to our proposal, a course can then be defined as follows:

- If its events always requires the same resources, it can be defined as a *course* - regardless whether it is known how to split it or not
- If its events require different resources, it has to be defined as now: a set of events that is part of one event group

Still one could also define a course whose events require the same resources as events of one event group. This would be no change to the current definition, which also allows both definitions.
We are aware that this would be a fundamental change. It would also require changing the definition of solutions: Not only events, but also courses could then be assigned to a timeslot. We hope that this change would not be more profound than the change from version 9 to 10 where the possibility of splitting events was introduced, but improve the availability of reliable information. It would not change the possibility which courses can be defined, but rather make the definitions easier understandable, and so perhaps lower the barreer of working with the instances.

Resource-Types

By abstracting teachers, schoolclasses and rooms to the term “Resource”, information is omitted: One cannot be sure anymore which resource he is dealing with. In case of the clique-search described in chapter 3.3, we want to know how many teachers and rooms are used by a specific event. This is necessary to estimate its size and importance.
In the documentation there exists the recommendation to use the resource-types *Room*, *Teacher* and *SchoolClass*. As mentioned, the australian instances disregard the recommendation by using the plurals “Rooms” and “Teachers” - a minor deviation that still can have a big influence on algorithms that overlook it.
Checking the usage of the recommended resource-types, i.e. by checking new instances with an xml-schema, would be an easy and effective step to prevent losing information.
Still, there exists the possibility of defining resources of another type if necessary, so this change would not affect the generality of the file-format.

Constraints

The instances and the constraints are built in a way that everything is allowed as long as it is not forbidden/constrained. The target is consistence: *"all constraints start with a verb, and describe how the constraint is not violated."* [28].

This approach leads to some peculiarities: By default, resources can be used by more than one events within a single timeslot. All events can be split into sub-events if they have a duration higher than 1. Open roles only have to be filled when they are constrained by the AssignResourceConstraint, and events do not have to be assigned to the timetable at all unless they are constrained by the AssignTimeConstraint. This approach forces all Instances (and authors of instances) to consider constraints which are possibly not relevant for them.
In example, it is clear that in the task of timetabling, all defined (sub-)events of an instance have to be assigned to the timetable. We consider such constraints as *implicit* or *immanent* to the problem. The authors of the School Benchmarking Project may discuss the following proposals, which would violate the mentioned consistence, but possibly help making the instances easier to construct and read:

- Leave out **AssignTimeConstraint**: Define that solutions are only valid if all events have a time assigned
- Leave out **AssignResourceConstraint**: Define that solutions are only valid if all open roles are filled
- Leave out **AvoidClashConstraint** (unless a case where its usage is necessary is known)
- Invert necessity of **SplitEventsConstraint**: Forbid splitting of events unless it is wanted explicitly (or implement the proposal of chapter 2.6)
- Reduce necessity of **AvoidUnavailableTimesConstraint**, **time groups** and **event groups**: Define that solutions are only valid if every compact (sub)event (an event that must not be split any more) has all its duration within just one day-timegroup (see below)

The philosophy of allowing everything unless it is constrained increases the need for event groups, time groups and constraints. As everything is allowed, one could assign a 2-hour gym lesson to the last slot of monday and the first slot of tuesday. To avoid this, many instances define **event groups** containing events of a certain duration. Then, the **AvoidUnavailableTimes**-constraints is applied to new **time groups** which just include the last slots of (day) time groups. For the given event groups, some durations are forbidden in Slots where their assignment would entail overlapping to the next (day) time group (i.e. duration 2 in the last slot of each day).
Currently, such overlapping assignments are theoretically allowed (not hard-constrained) by the instance Brazil7.
An alternative that does not violate the current philosophy is introducing an AvoidOverlap-constraint. Events this constraint is applied to are forbidden/penalized to be assigned in a way that their duration spans over more than one day-timegroup.

The current file-format allows weights of hard-constraints. Implementing the proposals described above would make weighting hard-constraints impossible for the mentioned constraints, which we consider to be of minor relevance.

The naming of the **PreferTimesConstraint** and the **AvoidUnavailableTimesConstraint** could be adapted: The first only can be applied to events, and the latter one only to resources. Names like “EventAvoidTimeConstraint” and “ResourceAvoidTimeConstraint” would be more self-descriptive.

The current definition of the **LimitWorkloadConstraint** seems unnatural:
“If the event is split each sub-event will have the original workload.” [28]
Assuming we have an event of total duration 2, which can possibly split to two events of duration 1. Further assume that this event has a workload of 4: If the event is not split, this would result in a workload of 4. If it is split, the workload would be 8.
Normally the LimitWorkloadConstraint is applied to teachers. The fact that holding 2x1 lesson is twice as much work as holding one lesson of duration 2 seems to not represent the reality. The appliance and semantics of this constraint is already planned to be changed in future versions.

The **LimitWorkloadConstraint** currently aims at being applied for teachers. It could also be applied to students or schoolclasses, to not have too many tough subjects within one day. This is not possible at the moment, because the constraint cannot be limited to time groups: it is always calculated for the whole timetable.
For the currently available instances this was not necessary, however, it may be so for future instances.

Other proposals

Documentation As the SBP is an evolving, growing project, the documentation partly was not accurate or contained obsolete parts. More detailed descriptions of how to apply constrains would have been helpful, as well as having more examples of calculating the deviation. In example, the SpreadEventsConstraint is described to return a list of deviations. It is not described whether this deviation-list is grouped by the event groups or the time groups of the constraint. The current definition of the constraints is available at http://www.it.usyd.edu.au/~jeff/hseval.cgi?op=spec.

Information about instances It would be helpful to have more information as the size and constraints of each instance in advance, altough some country-specific issues are described on the project website. As a first step, an overview of the constraints and other basic information of all instances can be found in appendix A.1.

A website showing the best results of different approaches available would be helpful, like there exists for the International Timetabling Competition of University Course Timetabling[3].

[3]http://tabu.diegm.uniud.it/ctt

Such a website was started in autumn 2009: http://opt-kd.cse.dmu.ac.uk/www/
Its development is stopped at the moment, and it unfortunately does not contain any results by now. Still, the "Algorithms" and "Academic History" give an interesting overview of the topic.

As currently no official result-comparison exists, Mr. Nurmi refered me to http://www.samk.fi/sttp during my search for existing results. After month of not even coming close to the results mentioned there, we examined the instances, starting with "S3 Secondary School". The best result mentioned on the website is a penalty of 2 without having any hard-constraint violations. The instance-description[4] gives an overview of the unavailability of resources. In the derived instance of the SBP a soft idle times constraint is applied, with a desired maximum of 0 idle times. This is also mentioned in the instance-description available on the web site:
Each class should have as few idle (leap) periods as possible: Yes
Each teacher should have as few idle (leap) periods as possible: Yes.

The resource (school class) "S8D" has a maximum of 18 consecutive slots over all days, and a workload of 22. The optimal assignment for this resource entails violating the idle times constraint by 4. The cost function of the SBP-instance is 'SumSquares', so the penalty would be 16. This also applies to resources S8E (workload 20, having 18 consecutive slots) and resource S8C. Thus, given the described conditions, no valid result with a penalty of 2 can exist. Mr. Nurmi confirmed the validity of the results, but unfortunately was not able to examine them in detail before handing in this thesis. Possible reasons for the divergence are a differing evaluation function, an incorrect documentation of the availability or an allowed minimum of idleTimes-violations.
Also, in the SBP-instance of the FinSecondarySchool, resource "S8D" has more unavailable slots (i.e. Slot 2-5) than the instance described on the website. It has exactly 22 slots available for its workload of 22, and so forces an idletimes-violation of 7 with a squared penalty of 49.
To avoid having problems when comparing results, a central website that publishes verified results would be of great help. Another possibility is to deliver the instances with the best known solution.

To emphasize the development of as well general but also specialized algorithms, further analyzing the instances, i.e. with graph-algorithms, would be of advantage. Revealing possible bottlenecks of instances in advance will especially be of importance when having a larger number of instances:

"Timetabling Problems are numerous: they differ from each other not only by the types of constraints which are to be taken into account, but also by the density (or the scarcity) of the constraints; two problems of the same "size", with the same types of constraints may be very different from each other if one has many tight constraints and the other has just a few. The solution methods may be quite different, so the problems should be considered as different." [10]

[4] http://www.bit.spt.fi/cimmo.nurmi/sttp/S3-Secondary-School.pdf

Limit Weights It would be nice to have a limit of how high the weight of a soft-constraint can be. Currently, no instance has a higher weight than 1000. Knowing that this will be true for all future instances would reduce the necessity of checking for exceeding datatype-limits.

Concluding, we think that the current definition of the instances is too abstract and vague. The format is in between being tailor-made for school-benchmarking and being abstract enough to describe all kind of timetabling problems, currently clearly tending to the latter.
Given school-benchmarking instances, this often unnecessarily complicates creating and parsing the instances. Splitting off the *immanent constraints* of section 2.6 from the instances would reduce the number of group-definitions and constraints within the instances, and hopefully facilitate creating and parsing the instances. The drawback would be giving up the policy of allowing everything unless it is constrained.
During the last month, some points mentioned here were already settled or improved, and others are under construction. Also, the information about the instances and the project itself is becoming more extensive and reliable. As the format is evolving and the number of instances increasing, we hope and believe that the School Benchmarking Project will play a key role in upcoming investigations and possibly also applications in the field of high-school timetabling.

CHAPTER 3

Solution Methods

In this chapter our algorithm will be presented in detail. Figure 3.1 shows the basic procedure:

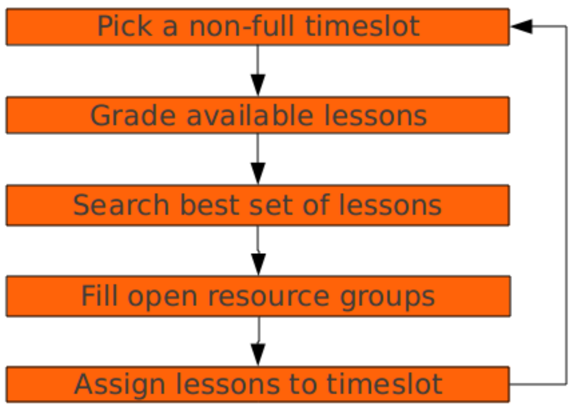

Pick a non-full timeslot
Grade available lessons
Search best set of lessons
Fill open resource groups
Assign lessons to timeslot

Figure 3.1: The basic procedure of our approach

This procedure is similar to the approach presented by Schmidt and Ströhlein [32] in chapter 1.4. We also construct a solution for the timetable out of solutions for each timeslot. We pick an empty timeslot and grade all lessons that are available, see section 3.1. Then, instead of creating a (hyper)graph with weighted edges as Schmidt and Ströhlein did, we chose to create a graph where the meetings (lessons) are weighted nodes, which is described in chapter 3.2. Meetings without resource-conflicts are interconnected; Figure 3.2 shows the graph we would construct out of the first timeslot of the problem they described in their work (which can also be found in chapter 1.4).

Having the constraint-graph constructed, we then search for the maximum-weight clique (chapter 3.3). Chapter 3.4 describes filling the open roles (resource groups) of the clique we found. The higher-level solving strategies can be found in chapter 3.5.
As noted in the introduction, we will first pick a not entirely full timeslot out of the timetable.

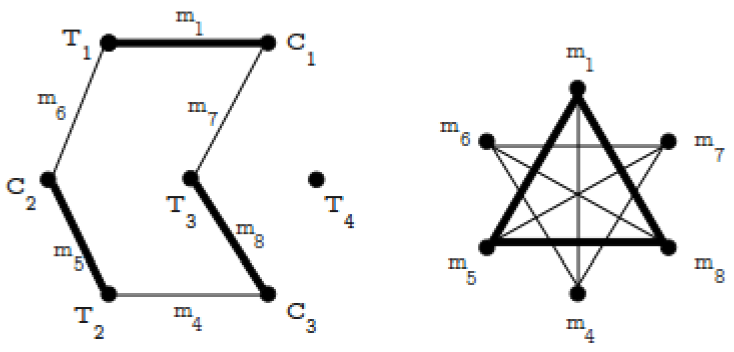


Figure 3.2: Left: The initial graph of hour one by Schmidt and Ströhlein
Right: The equivalent constraint-graph we would create in our approach.
Meetings m_i are edges in the initial graph, and nodes in our graph.

3.1 Grading

We basically calculate two different kinds of grades:

- **Penalties:** How much penalty would an assignment to the current timeslot entail
- **Urgencies:** How urgent is an assignment to the current timeslot

Grades are calculated for and applied to objects of timetabling, i.e. resources or lessons/sessons. Whereas the penalties are calculated according to the official evaluation-function (see chapter 2.1: *Constraints* and *Cost*), the main challenge is finding suitable urgencies. Two kinds of urgencies exist:

1. Constraint-urgencies aim at avoiding constraint-violations
2. Filling-urgencies aim at filling the timetable (assigning all events to timeslots)

For each urgency, a *ratio* is calculated that represents this urgency. For example, the session-urgency aims at assigning all sessions to the timetable, and will prefer sessions with dense assignments or few $possibleSlots$. Its ratio is a comparison of the pending assignments and the possible slots: $ratio = \frac{pendingAssignmens}{possibleSlots}$

All ratio-calculations were constructed and evaluated using a set of test-cases extracted from the instances. The target was to maintain the relation to the official evaluation: An urgency-ratio of 1 means that by assigning this i.e. resource, a deviation of 1 can be avoided. Ratios can also be negative if an assignment is unfavorable, or higher than 1 if they would avoid a deviation higher than 1.
To avoid disturbing the grades by currently unassignable sessions or resources, urgency-ratios can be limited to a range from -1 to 1.
The final urgency is always calculated as follows:

$$urgency = ratio^{exponent} \cdot weight \cdot externalWeight \tag{3.1}$$

The $exponent$ in (3.1) can be used to convert the ratios from a linear function to either a convex one ($exponent$ > 1) or a concave one ($exponent$ <1). This allows adjusting the sensitivity of this urgency, by either squeezing or straddling ratio-differences. The $weight$ denotes the weigth that this constraint is given within the instance, whereas the $externalWeight$ is a parameter we set to define the importance of this constraint compared to other urgencies. For hard-constraints we use the self-defined parameter $hardConstraint$ instead of $weight$. The value of $hardConstraint$ can be adjusted dependent on the difficulty of assigning all events to timeslots: The higher it is compared to other weights occuring in the instance, the more likely it gets to make assignments despite their penalty. This helps filling the timetable at the cost of increased penalty.
The $duration$ is the amount of consecutive slots of the lesson this urgency will be applied to, as defined in chapter 1.2. If a penalty or urgency changes with the duration, we calculate an urgency/penalty for each possible duration.

To get the final weight of a lesson, all (negative) penalties and urgencies are summed up, which themselve are derived from the resources, events and event groups that the lesson consists of. All penalties and urgencies arising from soft-constraints are multiplied with the parameter *softConstraintLevel*, as is described in chapter 4.1.
There are cases where a penalty exists, but neverthless an assignment is desired: The urgency will then compensate the penalty as well as express the favorability of an assignment. The first constraint (AvoidUnavailableTimesConstraint, section 3.1) includes a complete examples of the urgency- and penalty-calculation respecting the exponent and the weights.

In the following chapter we will first present constraint-urgencies (always having the name of the constraint, i.e. *ClusterBusyTimesConstraint*, as caption), and second the filling-urgencies. We omit code that aims at avoiding divions by zero, and code that distinguishes between the different cost-functions. The cost-functions are respected according to the official evaluation described in chapter 2.1. The notations are based on the formal definition of chapter 1.2.

AvoidUnavailableTimesConstraint

This constraint forbids or penalizes assigning resources to certain timeslots. We are using this constraint implicitly to represent a **resource-urgency**, which aims at assigning all the resource-workload to the timetable. The basic idea is to compare the possible timeslots with the pending resource-workload pW. The formula to calculate the ratio is simple:

$$ratio = \frac{pW}{slotsLeft} \tag{3.2}$$

pendingWorkload (pW) Table 3.1 shows the calculation of pW of a resource r. We sum up the duration of all pending meetings the resource is assigned to (considering open roles).

slotsLeft considers available slots and their penalty. For each slot where the resource and one of its sessions is available, we increase $slotsLeft$ by $1 - \frac{weight}{hardConstraint}$, where weight is the

Variable	Calculation/Explanation
$\mathit{workload} =$	$\sum_{m=0}^{\lvert M\rvert} \bar{A}_m \mid r \in M_R(m)$
$\mathit{openRoles} =$	$\sum_{m=0}^{\lvert M\rvert} \sum_{g=0}^{\lvert G\rvert} \sum_{i=0}^{\lvert R\rvert+1} (\bar{A}_m \cdot \frac{1}{\lvert G_R(g)\rvert}) \mid \mathsf{OpenRole}(m, g, i) = \mathit{true} \wedge r \in G_R(g)$
$\mathit{assignment} =$	$\sum_{t=0}^{\lvert T\rvert} \sum_{m=0}^{\lvert M\rvert} (\dot{T}_{rtm} >= 0)$
$pW =$	$\mathit{workload} + \mathit{openRoles} - \mathit{assignment}$

Table 3.1: Calculation of pendingWorkload (pW) of resource r

penalty of assigning the resource to this timeslot (0 if not constrained).

Example: Urgency Assume a resource-availability and penalty given in table 3.2. Further assume having the following variables:
$pW = 5$
$hardConstraint = 2000$
$externalWeight = 0.5$
$exponent = 2$

	Mo	Tu
1		
2		
3	-500	-500
4	-1000	-1000
5	■	■

Table 3.2: AvoidUnavailableTimesConstraint, calculation of slotsLeft: values are penalties; black slots mark unavailability

Except from pW, these variables are parameters we can adjust. We will now calculate the $slotsLeft$, and following the ratio and urgency:

$slotsLeft = 4 + 2 \cdot (1 - \frac{500}{2000}) + 2 \cdot (1 - \frac{1000}{2000}) = 6.5$
$ratio = \frac{5}{6.5} = 0.77$

The $weight$ or penalty of slots was considered when counting the $slotsLeft$. As noted above, we are then going to use $hardConstraint$ instead of the $weight$:
$urgency = ratio^{exponent} \cdot hardConstraint \cdot externalWeight$

This leads to a final urgency of $urgency = 0.77^2 \cdot 2000 \cdot 0.5 = 592.9$. This urgency is independent on the slot we are grading.

Combining urgency with penalty Table 3.3 continues the example using the penalties and variables introduced above. It shows the sum of the penalty and urgency we calculated: Slots 1 and 2 are not constrained, assignments to them are favored by the urgency we calculated above (592.9). Assignments to slot 3 are still favored, but the $weight$ (penalty) of 500 will be substracted: We so have a sum of 92.9. Slot 4 is weighted

	Mo	Tu
1	592.9	592.9
2	592.9	592.9
3	92.9	92.9
4	-407.1	-407.1
5	■	■

Table 3.3: Sum of urgency and penalty of the AvoidUnavailableTimesConstraint

with 1000, so the sum will be -407.1, and the assignment unfavorable. This is appropriate because there are enough other slots to assign the pending workload.
All lessons that require this resource will obtain the sum of urgency and penalty of the timeslot that we currently grade.

ClusterBusyTimesConstraint

The ClusterBusyTimes-constraint is applied to resources: It aims at assigning a resource to a certain amount of time groups (which are usually days). A resource r is assigned in time group d if $\exists t \exists m \mid t \in D_T(d), m \in M : \dot{T}_{rtm} > -1$

A $minimum$ and $maximum$ can be defined, and the number of time groups used should lie in between. The ratios we describe here do not consider the case that the constraint is only applied to a subset of all (day-)time groups, which up to now does not appear in any instance. We so can define the *current time group* as the intersection of the time groups the current timeslot t (the timeslot we are currently grading) belongs to: $T_D(t)$, and the time groups the ClusterBusyTimes-constraint is applied to.
Table 3.4 shows the variables we are using to calculate the ratio. pW and $workload$ are calculated as introduced in section 3.1.

Variable	Explanation
slotsLeft ...	Sum of all slots left in any time group with assignment
SLCG ...	**S**lots **L**eft in **C**urrent **T**imegroup
avgWorkload ...	Average workload we have to fill in in one time group when using the allowed maximum of time groups: $\frac{workload}{maximum}$
overflow ...	Number of possible timeslots of the current time group compared to the average workload we have to assign: $SLCG - avgWorkload$
dayLength ...	Nr of slots a normal day has
pressureExpo ...	Parameter for exclusively spreading pressure (default: 2.5)
weightOverflow ...	Weight of overflow (default: 1.5)
pendingGroups ...	$maximum - \sum Timegroups\ with\ assignment$

Table 3.4: ClusterBusyTimesConstraint: variables used for calculating the ratio

Maximum: No resource-assignment in current time group
If a resource is not yet assigned to a time group, we will try to find out whether an assignment to the current time group will help not reaching the $maximum$. This will be the case when there are many possibilities of assigning the resource within this time group. The $ratio$ of equation (3.3) consists of 2 basic components:

- $rawRatio$: Does the current time group offer enough possible assignments to avoid violating the maximum

- *pressure*: Difficulty of the constraint: Number of assignments compared to the typical day-length

$$
\begin{aligned}
rawRatio &= \frac{overflow \cdot weightOverflow}{dayLength} \\
pressure &= \frac{avgWorkload \cdot 2}{dayLength} \\
ratio &= rawRatio \cdot pressure^{pressureExpo}
\end{aligned} \tag{3.3}
$$

Maxmimum: >0 Resource-Assignments in current Time group

If there already exists a resource-assignment, we compare the pending workload with the number of slots that can be assigned without assigning the resource to more than $maximum$ time groups. The $pressure$ is calculated as in the case without assignment in the current time group.
The divisor of the ratio in equation (3.4) represents the number of possible slots when using $maximum$ time groups. We assume that in each $pendingGroup$ we will be able to assign the $avgWorkload$ - an estimation that could be made more exact by analyzing the number of timeslots in time groups without assignment.

$$ratio = \frac{pW}{\frac{pendingGroups \cdot avgWorkload}{pressure} + slotsLeft} \tag{3.4}$$

Minimum

The ratio to avoid minima is calculated by comparing the pendingWorkload pW with the pendingGroups (as defined in *"Maximum: No Resource-Assignment in current time group"*, section 3.1):

$$ratio = \frac{pW}{pendingGroups} \tag{3.5}$$

If the resource is assigned to the current time group, the ratio is *substracted* to avoid further assignments, and *added* to favour its assignment if the resource is not yet assigned.

LimitBusyTimesConstraint

The number of timeslots a resource is assigned to within a time group should lie between a $minimum$ and $maximum$ - or the resource is not assigned at all to this time group.
The $minimum$ of this constraint is the only case where we do not apply the penalty directly: Given i.e. $minimum = 4$, the first assignment in a time group would always be highly penalized. We instead calculate a subsitute for the penalty within the urgency.
Minimum and maximum have to be considered together to not interfere. The urgency and penalty is calculated differentiating between three distinct cases. The variables we use are defined in Table 3.5. Again, see section 3.1 for the exact definition of pW.

Variable	Explanation/Calculation
$assignment\ldots$	assigned slots in the current time group
$pW\ldots$	**p**ending resource-**W**orkload
$openMin\ldots$	Sum of resource-assignments (assigned slots) below $minimum$ in all time groups exept the current
$sL\ldots$	**s**lots**L**eft (maximum of further assignments) in current time group
$sLAG\ldots$	**s**lots**L**eft in **A**ssigned time**G**roups (all time groups with a resource-assignment >0)
$slotsLeftForMaximum\ldots$	nr of timeslots left in all time groups without violating maximum
$pWcm :=$	$pW - openMin$ // **p**ending**W**orkload **c**onsidering **m**inimum
$sLAGcm :=$	$sLAG - openMin$ // sLAG **c**onsidering **m**inimum
$sLrm :=$	$\min(maximum - assignment, sL)$ // **s**lots **L**eft in current time group **r**especting **m**aximum

Table 3.5: LimitBusyTimesConstraint: variables used for calculating the ratio

1. Resource not assigned to current time group

If the resource is not yet assigned to the time group (for the definition please refer to chapter 3.1), we look whether the $minimum$ of the time group can be fulfilled. $slack$ is the amount of possible resource-assignments in the current time group, below or above the minimum:

$$slack = \min(pWcm, sLrm) - minimum$$
$$ratio = \begin{cases} slack & \textbf{if } slack < 0 \\ -0.5 & \textbf{if } slack = 0 \textbf{ and } sLAG > pW \\ \frac{slack}{\min(sL, pWcm))} & \textbf{if } slack > 0 \end{cases} \tag{3.6}$$

2. Current Resource-Assignment below minimum

We will just try to make resource-assignments more urgent, dependent on how many possibilities for resource-assignments there still are. Line 3 in algorithm 3.1.1 is the general urgency of filling the open minimum of the current time group: The dividend, $minimum - assignment$, is the open minimum of the current time group, and the divisor, $min(sL, pWcm)$ represents the still pending possibilities of filling the open minimum.
Starting with line 4 we treat the case where the open minimum of the current time group is overfilled by assigning a resource with a too large duration: We penalize such assignments if they make filling the $openMin$ of the other time groups impossible.

Algorithm 3.1.1 Ratio-calculation of LimitBusyTimes when the current assignment is below the minimum

```
1: // input-arguments: minimum,assignment,openMin,sLrm,pWcm,duration
2: // output: ratio
3: ratio = $\frac{\text{minimum}-\text{assignment}}{\min(\text{sLrm},\text{pWcm})}$
4: overfill := duration − (minimum − assignment)
5: if overfill > 0 and openMin > 0 then
6:     currentViolation = openMin + (minimum − assignment) − pW
7:     newViolation = $\min$(openMin, (duration − pWcm))
8:     ratio = ratio + currentViolation − newViolation
9: end if
10: return [ratio]
```

3. Current Resource-Assignment fulfills minimum

If the $minimum$ of the current time group is already filled, we estimate whether further assigning the resource will increase or decrease the possibility of future violations. We again have to distinguish three different cases, and will demonstrate the ratio-calculation with algorithm 3.1.2. If the maximum is not explicitly considered (as i.e. done with the fulfilled $minimum$ when pW fits into assigned time groups), we add its penalty seperately according to the official evaluation. In any case we add an urgency for not violating the $maximum$:

$$ratio = \frac{pW}{\max(0.5, slotsLeftForMaximum)} \tag{3.7}$$

Algorithm 3.1.2 Ratio-calculation of LimitBusyTimes-minimum when the minimum of the current time group is fulfilled

```
// input-arguments: pWcm,sLAGcm,sL,sLrm,minimum,maximum,assignment,duration
// output: ratio
ratio := 0
currentMaxDev := $\max(0, \text{assignment} - \text{maximum})$ // current violation of maximum
newMaxDev := $\text{assignment} + \text{duration} - \text{maximum}$ // new violation of maximum
minDev := $\max(\text{minimum} - \text{pWcm}, 0)$ // deviation by violating minimum of new time group
if $\text{pWcm} < \text{duration}$ then
    // Pending openMin would be impossible to fill
    ratio = $\text{pWcm} - \text{duration}$
else if $\text{pWcm} \leq \text{sLAGcm}$ then
    // pW completely fits into time groups with assignments
    if $\text{newMaxDev} > 0$ then
        ratio = $\text{ratio} - (\text{newMaxDev} - \text{currentMaxDev})$
    else if $\text{pWcm} < \text{minimum}$ then
        ratio = $\frac{\text{minDev}}{\text{sLAGcm} \ / \ \text{pWcm}}$
    end if
else if $\text{pWcm} > \text{sLAGcm}$ then
    // pW too large to fit in assigned time groups
    if $\text{pWcm} > (\text{sLAGcm} + (\text{sL} - \text{sLrm}))$ then
        // opening new time group necessary also when violating maximum
        if $\text{newMaxDev} > 0$ then
            ratio = $\text{ratio} - (\text{newMaxDev} - \text{currentMaxDev})$
        end if
        if $(\text{pWcm} - \text{sLAGcm}) < \text{minimum}$ then
            pWcmas := $\text{pWcm} - \text{duration}$ // pWcm after assignment of current duration
            sLAGcmas := $\text{sLAGcm} - \text{duration}$
            ratio = $\text{ratio} - \frac{\text{minimum}-(\text{pWcmas}-\text{sLAGcmas})}{1+\text{sLAGcmas}}$
        end if
    else
        // opening new time group can be avoided by violating maximum
        maxDev := $\max((\text{duration} - (\text{maximum} - \text{assignment}), (\text{pWcm} - \text{sLAGcm}))$
        additionalDev := $\text{maxDev} - \max(0, \text{assignment} - \text{maximum})$
        ratio = $\text{ratio} - \text{additionalDev} + \text{minDev}$
    end if
end if
return [ratio]
```

LimitIdleTimesConstraint

An idle time exists if within a time group (day), a resource is assigned to previous and subsequent timeslots, but not to the current one. As there is no instance available that has a LimitIdleTimes-

constraint with a $minimum > 0$, we only calculate the urgency for the $maximum$.
The constraint sums up the deviation of all time groups it is applied to. Table 3.7 explains the variables we are going to use. The unconstrained slots uS are the number of timeslots which the resource can be assigned to without necessarily increasing the deviation. This can change by assigning the resource to the current timeslot.

Table 3.6 shows how the currently unconstrained slots, $uSOld$, and the unconstrained slots after an assignment, $uSNew$, are calculated: $uSOld$ are three slots on Monday (MO1, MO3, MO4). For Tuesday we do not yet have an assignment, so we take the size of the largest contiguous block of Slots. This are 4 slots (TU3 - TU6), so we have $uSOld = 7$.
If our current slot is TU1, we would have $uSNew = 4$. Slots TU3 - TU6 would not be assignable without a deviation any more, but TU1 would be: We do not really assign the resource when counting the slots.
Having MO6 as start would so lead to $uSNew = 8$. The urgency that is calculated this way is independent from the duration.

	Mo	Tu
1		
2	▨	■
3		
4		
5	■	
6		

Table 3.6: Calculation of unconstrained slots: grey slots indicate resource-assignment, black slots mark unavailability; white slots are assignable.

Variable	Explanation
$pW \ldots$	**p**ending **W**orkload of Resource
$uSOld \ldots$	**u**nconstrained **S**lots: Number of slots where unconstrained resource-assignment is possible (in all time groups)
$uSNew \ldots$	$uSOld$ considering the assignment in the current timeslot
$assignment \ldots$	Nr of used slots in the current time group
$currentDev \ldots$	current deviation: Nr of idle times in all time groups
$newDev \ldots$	new deviation: Nr of idle times in all time groups when assigning to the current timeslot
$\mathsf{cost}(x) \ldots$	Function that calculates the cost of deviation x respecting the costFunction of the constraint
$currentCost :=$	$\mathsf{cost}(currentDev)$
$increaseCost :=$	$\mathsf{cost}(currentDev + 1) - currentCost$

Table 3.7: LimitIdleTimesConstraint: variables used for calculating the ratio

If the deviation is below $maximum$, some of the following calculations would fail because they base on currentCost, which is 0 in this case. We define a parameter p where $0 < p < 1$ for adjusting the weight of increasing deviation below the maximum. After this initial calculation, the main ratio is calculated with algorithm 3.1.3.

Algorithm 3.1.3 Considering a deviation below maximum

```
1: // input-arguments: ratio,currentDev,newDev,maximum,p
2: // output: ratio
3: ratio := 0
4: devGain := newDev − currentDev
5: if devGain > 0 and newDev ≤ maximum then
6:     ratio = ratio − $p \cdot$ devGain
7: else if devGain > 0 and currentDev < maximum then
8:     ratio = ratio + $p \cdot$ devGain
9: end if
10: return [ratio]
```

Assigning the resource to the current timeslot, we distinguish three main cases regarding the deviation:

Decreasing Deviation

When the deviation decreases, the penalty will be negative - and so make the assignment favorable by itself. We do not a calculate additional urgencies.

Unchanged Deviation

An unchanged deviation means that we can assign a resource without increasing the deviation, which is desired normally.
Equation (3.8) shows the ratio-calculation in case that the resource is already assigned to the current time group:

$$ratio = \frac{pW}{uSOld} \cdot increaseCost \tag{3.8}$$

If there is no resource-assignment in the current time group, we again distinguish several cases which are shown in algorithm 3.1.4.

Algorithm 3.1.4 Ratio-calculation of LimitIdleTimes with unchanged deviation

```
// input-arguments: pW,uSOld,uSNew,currentDev,newDev,increaseCost
// output: ratio
ratio := 0
if usOld = uSNew then
    // Assignment is in the largest block: favorize
    prevCost := cost(currentDev − 1)
    ratio = $\frac{\text{pW}}{\text{uSOld}} \cdot \max(\text{currentCost} - \text{prevCost}, 1)$
else
    // Assignment is not in the largest block: penalize
    // Slack is uS compared to pW: nr of slots too much/less for pW
    // slackLoss is the percentage of slack we lose by assignment to current timeslot
    oldSlack := uSOld − pW
    newSlack := uSNew − pW
    if oldSlack ≤ 0 then
        slackLoss := $\frac{\text{oldSlack} - \text{newSlack}}{\text{uSNew}}$
    else
        slackLoss := $\min(1 - \frac{\text{newSlack}}{\text{oldSlack}}, 1)$
    end if
    ratio = −slackLoss · increaseCost
end if
return [ratio]
```

Increasing Deviation

An increasing deviation will always be penalized by the penalties. We use the urgency to modify and fine-tune this penalty. The ratio-calculation is shown in algorithm 3.1.5.

Algorithm 3.1.5 Ratio-calculation of LimitIdleTimes with increasing deviation

```
1: // input-arguments: pW,uSOld,uSNew,currentDev,newDev,currentCost,increaseCost
2: // output: ratio
3: ratio := 0
4: if $uSOld = uSNew$ and $(pW - 1) \geq (currentDev - newDev)$ then
5:    // the now opened gap can still be closed: reduce penalty by generating urgency
6:    $ratio = \frac{pW}{uSOld} \cdot increaseCost$
7: else if $pW > uSOld$ then
8:    // The currently created gap can not be closed
9:    // The pW is too high to fit into the currently available Slots (uS)
10:   if $pW \leq uSNew$ then
11:      // The slots gained by the current violation allow assigning all pW
12:      $ratio = increaseCost$
13:   else
14:      // The slots gained are still not enough; another violation would be unavoidable
15:      // Further increase the penalty by this future violation with a negative urgency
16:      unavoidableCost := $cost(newDev + 1)$
17:      $ratio = currentCost - unavoidableCost$
18:   end if
19: end if
20: return [ratio]
```

SpreadEventsConstraint

The purpose of the SpreadEvents-constraint is to spread events of an event group over different days. More formally, the number of events of an event group that are assigned to a certain time group should lie between a $minimum$ and $maximum$. Assigning an event of duration > 1 still counts as one assignment.

Again, the *current time group* is the intersection of the time groups the constraint is applied to, and time groups the timeslot we are currently grading belongs to. Table 3.8 explains the variables we are going to use.

Variable	Explanation
$cA \ldots$	**c**urrent **A**ssignment: nr of assignments in current time group
$pendingAssignments \ldots$	nr of pending event-assignments of the constrained event group
$possibleAssignments \ldots$	**p**ossible **a**ssignment in all time groups (also current) without violating the $maximum$, and considering the duration of the events of $possibleAssignments$
$pACG \ldots$	**p**ossible **a**ssignments in **c**urrent time**g**roup
$openMinCG \ldots$	open (unfulfilled) minimum of the **c**urrent time**g**roup
$openMinOG \ldots$	open minimum of other time**g**roups (all but the current one)

Table 3.8: SpreadEventsConstraint: Variables used for calculating the ratio

Maximum The maximum-urgency is only applied if the maximum is not yet reached ($cA \geq maximum$). Equation (3.9) shows the calculation. The left part $\frac{pendingAssignments}{possibleAssignments}$ is independent on the current time group and can be considered as a general urgency/pressure: It compares the number of pending assignments with the slots they can be assigned to without violating the maximum. The right part $\min(\frac{maximum-cA}{pACG}, 1)$ represents the urgency of assigning to the current time group: It relates the assignments missing to reach the maximum with the number of timeslots that are available for such assignments.

$$ratio = \frac{pendingAssignments}{possibleAssignments} \cdot \min(\frac{maximum - cA}{pACG}, 1) \tag{3.9}$$

If there are more possibleDurations left for any event that should be split, we assume the one that results in the largest number of subevents, and thus in the highest $possibleAssignments$. Having more possibleAssignments makes it more difficult to not violate a maximum, so we anticipate this case.

Minimum The only instance that contains a (hard) $minimum$ is the changed italian instance that incorporates the proposal of section 2.6.
Handling minima is difficult with our approach: Having an assignment below the minimum, we normally would favorize *all* lessons that possibly fill the minimum. This leads to assigning too many lessons, and so prevents the minima of other time groups from being filled. Only favoring *some* of those lessons conflicts with the idea of leaving the decision of which lessons to pick to the clique-search.
Fortunately the constraint of the Italy-instance is hard, which allows respecting it apart from the grading, whithin the clique-search (chapter 3.3): We count the open minima of all time groups. When filling any timeslot, we only allow assigning so many lessons that filling the other open minima is still possible. We do so by creating a new resource group: All lessons of the event group have to be modified to fill this new resource group. Then, we set a limit for this resource group (see chapter 3.2), according to the open minima: $limit = pendingEvents - openMinOG$. Soft-constraints (which up to now are not used by any instance) cannot be considered this way.
Additionally, we apply an urgency when the assignment is below minimum. The ratio-calculation is given by equation (3.10). $nrSessions$ is the amount of sessions which are available in any of these timeslots, $slotsLeft$ is the number of pending timeslots of the current time group.

$$ratio = \frac{openMinCG}{slotsLeft \cdot nrSessions} \tag{3.10}$$

Other Constraints

For the **LimitWorkloadConstraint**, only the penalty is calculated: First, our approach is in general less suitable for the australien instances, see section 4.6. This constraint only occurs in the australian instances. Second, the definition and evaluation of the LimitWorkloadConstraint is going to change, because its current definition causes some problems, see section 2.6.

All other constraints are either considered implicitly by unavailabilities and urgencies calculated above, or calculated later when filling the open resource groups (see section 3.4). We will briefly mention the not yet treated constraints and explain why no urgency is necessary. All but the AvoidSplitAssignmentsConstraint only occur as hard-constraints.

LinkEventsConstraint linked events get joined to one iteration of a session

AssignResourceConstraint considered by filling open resource groups

AssignTimeConstraint considered implicitly (main task)

SplitEventsConstraint we only instanciate lesson-durations that do not violate this constraint

DistributeSplitEventsConstraint as SplitEventsConstraint

AvoidSplitAssignmentsConstraint considered when grading and filling open resource groups

The urgencies described from now on do not focus on avoiding constraint-violations, but on filling the timetable.

Bin-Packing

This urgency is intended to facilitate gapless resource-assignments. Some instances define resources that have very few or no "slack-slots" compared to their workload, i.e. room R301 of the FinCollege-instance has a workload of 39, and 40 timeslots available. The events that require this room have a duration of 2 and 3. Such problems are known as **bin-packing**, and are one of the NP-complete problems described by Kingston [20]: We have to pack the items of size (duration) 2 and 3 into 5 bins (days) of size 8 (number of timeslots).
We implemented an algorithm inspired by the First Fit Decreasing of Johnson [17]: Items are assigned descending by their size; if an item does not fit in the current bin, it is assigned to a new bin. As we do not assign the events resource-wise, we cannot fully implement this procedure. Instead, we generally prefer large items (lessons). If an assignment would inhibit fully filling the current day, we penalize it, and favor sizes that still keep the possibility of a complete filling.

We distinguish hard and soft bin-packing as shown in table 3.9: The hard bin-packing only considers resource-unavailabilities, whereas the soft bin-packing also considers penalties of resources (AvoidUnavailableTimes). The timeslots to fill are picked ascending, so we look for the number of contiguously assignable timeslots starting with the current timeslot: grading timeslot 1 of table 3.9, we have 4 timeslots for the soft-constraint (using the weight of -500), and 5 for the hard-constraint (using $hardConstraint$ as

	Mo
1	
2	
3	
4	
5	-500
6	■

Table 3.9: Example for contiguous timeslots of the bin-packing urgency: negative values are penalties, black slots mark unavailability; white slots are assignable

weight).

The tricky part is to find out whether a duration of a specific session would inhibit a gapless filling of the contiguous timeslots. We assume that within one day, only one lesson of a session can be assigned - which is a simplification that may not always be correct. We look for all possible session-starts and durations within the contiguous slots. Then we calculate which session-durations still permit assigning the resource gapless into the contiguous timeslots. Continuing the hard-constraint example with two sessions and possible start-slots of table 3.10, two session-durations would be valid in slot 1: Duration 2 of session 1 (gapless filling by assigning session 2 with duration 3 in slot 3), and duration 3 of session 2 (gapless filling by assigning session 1 with duration 2 in slot 4). Assigning session 2 with duration 2 would inhibit a gapless filling.
We apply the urgency only if the following conditions hold:

- the current slot offers more than one possible session for the resource, or at least one session of the resource has more than one possible duration
- the session-durations that permit a gapless filling do not completely equal all session-durations that are available
- there is at least one session-duration that allows a gapless filling

The calculation of the ratio is simple - the tighter the resource-workload, the higher the ratio. The pendingWorkload pW is calculated as defined in 3.1; $slotsLeft$ is the number of timeslots where both the resource and any of its sessions are available.

$$ratio = \frac{pW}{slotsLeft} \tag{3.11}$$

	Session 1	Session 2
1	**2**	2, 3
2	2	2, 3
3	2	2, **3**
4	2	2
5		
6	■	■

Table 3.10: Example of possible start-slots of sessions

When calculating the final urgency, see equation (3.1), we recommend an $exponent > 1$ to not unnecessarily apply the bin-packing urgency to resources that are not urgent. Additionally, to not disturb the average resource-grade and interfering with other urgencies, we add half of the final urgency if the session-duration allows a gapless filling, and substract half of the final urgency if it does not.

Session-Urgency

As described in section 2.4, sessions can have lessons of differing duration. This can be courses that have meetings of differing length. We calculate an urgency *for each duration* of a session by comparing its pending assignments with the timeslots it can possibly be assigned to:

$$ratio = \frac{pendingAssignments}{possibleSlots} \tag{3.12}$$

We pick the durations of a session descending, and evaluate how often they can be assigned to the timeslot. When examining the next-lower duration we consider the slots that are theoretically occupied by higher durations.

Table 3.11 demonstrates how this is applied: Assume having a session consisting of 3 iterations with the durations 2, 2 and 1.
We first examine duration 2. The two only start-slots for duration 2 will be Tu-3 and We-1. We-2 is not a possible slot, because we consider that assigning to We-1 will occupy We-2. This leads to a ratio of 1 (2 $pendingAssignments$, 2 $possibleSlots$). Then we examine duration 1. There are 4 slots left: Mo-3, Tu-1, We-3 and Th-4. The ratio will be 0.25.

	Mo	Tu	We	Th	Fr
1		■	■		
2			■		
3	■	■	■		
4		■		■	

Table 3.11: Session-urgency with differing durations. Grey timeslots indicate that all resources of the session are available

After grading all lessons, we will search for the best-graded set of lessons that can be held within the timeslot. We do this using a so-called constraint graph.

3.2 Constraint Graph

We model which lessons of one timeslot can be held together using a constraint graph. In this graph, each node represents a lesson. Two nodes (lessons) are connected when they can be held simultaneously. They will not share any resource as room, teacher or school-class. Figure 3.1 shows an example of an constraint graph consisting of two rooms, two teachers and two classes.

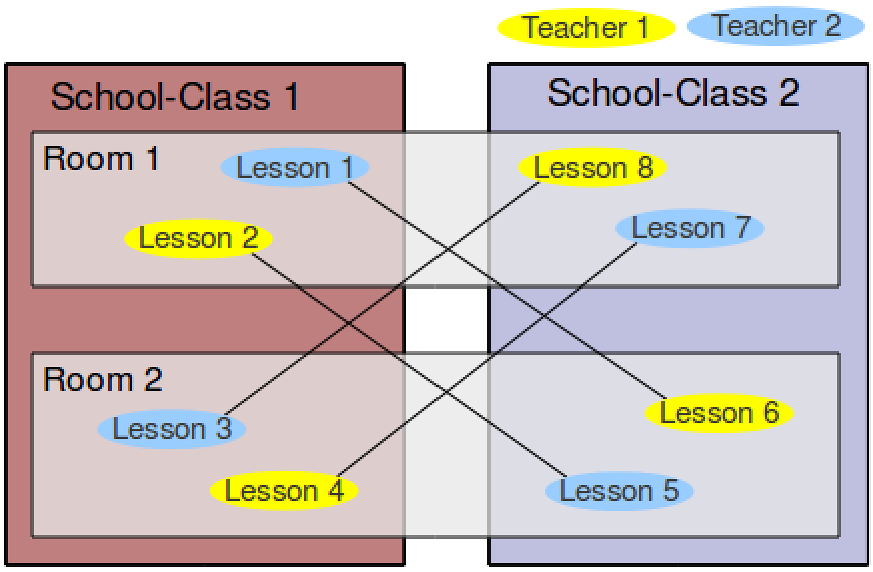


Figure 3.1: Example of a constraint graph*: All lessons that can be held simulteanously are connected*

A constraint graph is always constructed for a specific timeslot. The hard-constraints (such as teacher-availability) are represented by the existance and connectedness of the lessons. The soft-constraints are represented by the grades of the lessons.

Resource Group Limits

The initial idea was to instanciate each possible resource of an open resource group: If an event can be held in three different rooms, this would lead to three distinct lessons. Unfortunately, some instances contain events that have up 21 open roles, which makes instanciating each possible combination infeasible.

To avoid this problem, resource group limits were introduced: For each resource group we search for the number of existing resources in a specific timeslot. This existing resources then build the resource group limits. Having two gym-rooms available on the first timeslot on monday means that the resource group "gym-rooms" will have the limit 2. The nodes (=lessons) store how many resources of each resource group they will occupy. For any valid clique it is obligatory that the sum of these occupied resource groups does not exceed the resource group limits.
When having resource group limits, it cannot be guaranteed any more that it is possible to assign resources to all open roles of the maximum-weighted clique. This case is further described in section 3.4.

The final constraint graph contains the following information:

- Nodes (Lessons)
  - weight (grade)
  - occupies resource groups
  - deepness
- Edges: lessons are connected if they can be held simultaneously
- Resource group limit

Having defined the constraint graph, the next task is to search for the best-graded set of lessons that can be held simultaneously.

3.3 Search for Cliques: Maximum weight clique problem

Every clique in the constraint graph - for a specific timeslot - represents a set of lessons that can be held simultaneously. A clique is a set of nodes that are all interconnected, see figure 3.1. The interconnectedness means in our case that these lessons do not share any resources.

As we have graded (or weighted) lessons, we do not search for the maximum clique, but for the clique that has the maximum weight: This problem is known as *maximum weight clique*

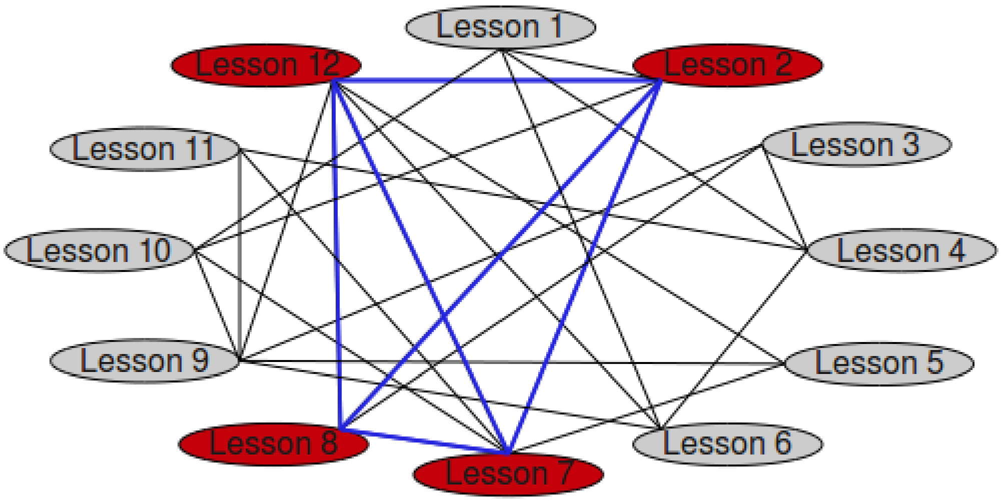


Figure 3.1: Example of a clique*: The red marked lessons are interconnected with each other and form the maximum clique of this graph. They represent lessons that can be held simultaneously.*

problem. It is a generalization of the maximum clique problem, which itself is known to be NP-complete [18]. In some instances, another constraint is added: The open roles of events, which are converted to open resource groups, result in a resource group limit which is described in section 3.2.

Upper bound of the number of solutions for one Clique-Search

Let us assume having a search-problem of 550 nodes: 55 school-classes with each 10 different subjects available in one timeslot.
In the first step there are $55 \cdot 10 = 550$ lessons available. Picking one lesson results in $54 \cdot 10$ possible solutions in the worst-case for the next step, and so on.
The upper bound of the number of possible solutions for one timeslot would then be $(55 \cdot 10) \cdot (54 \cdot 10) \cdot ... \cdot (1 \cdot 10)$, see equation (3.13) - assuming that the school-classes are never split.

$\alpha \dots$	number of lessons
$\beta \dots$	number of school-classes
$L = \frac{\alpha}{\beta}$	number of lessons per school-class

$$\beta! \cdot L^{\beta} \tag{3.13}$$

In our example this would result in an upper bound of $55! \cdot 10^{55} = 1.27 \cdot 10^{128}$ possible solutions for one single timeslot.

Existing Methods

Kumlander [21] describes an **exact algorithm** which makes use of a vertex coloring. Initially a vertex-coloring is created heuristically. Then, a depth-first search is performed by extending previously sorted small cliques/nodes. Whenever a subgraph cannot be extended to a clique that has a higher weight than the currently known maximum, it is pruned. The decision when this is the case is the key point of each branch-and-bound algorithm. Instead of using multiple color-classes per vertex as done in [1], Kumlander uses information gained by the initial vertex coloring, having only one color per vertex. He estimates the potential maximum weight clique by summing up the maximum weights per color of a subgraph. This sum is always equal or higher than the maximum weight clique of a subgraph. The sum is combined with the weight of the clique the subgraph is created of, and compared to the currently known maximum.
Kumlander combines his pruning with a backtracking introduced by Östergård [26]. On denser graphs, Kumlander finds the optimum significantly faster than Östergård, and even further improved the performance of his algorithm for dense graphs in 2006 [22].

Using genetic algorithms to **heuristically** solve maximum weight clique problems seems appropriate: The genetic information is a boolean vector of all vertices, and the fitness function is the weight of the clique it represents. Mutations are bit-flips of the vertex vector. The crucial part is crossing individuals, and repairing invalid cliques if necessary. Balas and Niehaus [2] cross two individuals by creating a subgraph induced by the union of the of their vertex sets. The maximum weight clique of this subgraph is searched by inverting it and applying a maximum flow algorithm, which is executable in polynomial time.

In 2006, Singh and Gupta [33] developed a hybrid evolutionary approach. Two parents are crossed using a fitness-proportional approach, where the vertex of a parent is added to the child with probability $\frac{fitness\ of\ parent}{fitness\ of both\ parents}$. As the child may not be a valid clique, it is repaired heuristically: single vertices are chosen randomly, and either the vertex itself or all vertex not adjacent to it are deleted from the child. The resulting valid clique C is then extended heuristically. One of the adjacent vertices is either added randomly or by chosing the best-graded adjacent vertex u of clique C, until no more adjacent vertices exist. The grade of a adjacent vertex is its weight $w(u)$ multiplied with the number of vertices that are both adjacent to u and to C, where C_{ad} denotes all vertices adjacent to C, and E are all edges of the graph: $grade(u) = w(u) \cdot |\{t : t \in C_{ad} \wedge (u,t) \in E\}|$.

Singh and Gupta report getting better results than the prior state-of-the-art approaches as the quadratic formulation of Busygin [8] and the pivoting approach of Bomze et al. [4], which are both based on the Motzkin-Straus theorem.

Having graphs with a density up to 0.968 percent and 950 nodes, exact algorithms are infeasible for our problem. Because of having an additional constraint (the resource group limits) and additional information (the deepness of nodes), we implemented a custom heuristic. The main targets were reliability and speed: As reliability we understand that the quality of the solution should be independent on the structure of the graph and the deviation of the gradings. As we iteratively delete and re-fill the timetable, speed is also crucial.

Terminology

To limit memory-usage, we define a **clique-limit**: It is the maximum number of cliques that we hold in memory at any given time. If the existing cliques exceed this limit, some bad graded cliques or cliques of small size (and deepness) are deleted.

A clique has **peers**: These are lessons that are themselves connected with all lessons of a clique. Peers can be used to extend a clique.

Extending a clique means that we create new, larger cliques out of an existing clique and each peer. Of course, the new clique must not violate the resource group limit.

One **step** means extending cliques until a certain amount of not yet existing cliques was evaluated: We only count new cliques that do not violate the resource group limit.

The **deepness** of a clique is the sum of the deepnesses of the lessons it contains. The definition of deepness can be found in section 2.5. This is the equivalent to the clique-size of the standard maximum weight clique problem.

Basic Procedure

We start with cliques of size 1, each consisting of one single lesson. Then we iteratively search for the most-promising existing clique, and further extend it. How promising a clique is depends on its grade (weight), deepness, and on the peers it has.
Obviously, it is of extreme importance to choose the right cliques to extend. We therefore grade the cliques again internally, dependent on their (raw) grade, their peers and deepness. With increasing runtime, we shift our search-focus to larger cliques, taking the clique-deepnesses into account.
With this procedure it is possible to repeatedly create an already existing clique by extending different subsets of it. To avoid this we store each created clique, and only create a new clique if it not already exists.
We often have to pick the best-graded clique, and also have to often insert new cliques. Therefore we decided to use a heap as the main datastructure for storing the cliques.
Two different approaches were tried out: First, using one single heap that contains all exising cliques, which is described by algorithm 3.3.1. Second, creating a heap for each deepness that will only contain cliques of this certain deepness. In this case, the lessons at line 10 and the new cliques at line 27 would be pushed into the heap of its particular deepness.

Algorithm 3.3.1 Basic procedure of clique-search

```
// input-arguments: lessons, peers, stepCliqueMultiplier, abortAfterEmptySteps
// output: ratio
currentMax, maxBeforeExtending, stepsWithoutMax := 0
stepCliqueNr = stepCliqueMultiplier · |lessons|
maxClique := ()
cliqueHeap := empty heap
// initially insert lessons into cliqueHeap
for all lesson ∈ lessons do
    calculate internGrade
    cliqueHeap.push(internGrade,lesson)
end for
// start expanding cliques
while stepsWithoutMax < abortAfterEmptySteps do
    maxBeforeExtending = currentMax
    // beginning of one step
    nrNewCliques := 0
    while nrNewCliques < stepCliqueNr do
        clique := cliqueHeap.pop()
        for all peers of clique do
            calculate newClique, cliquePeers, grade and internGrade
            nrNewCliques ← nrNewCliques + 1
            if grade > currentMax then
                maxClique = clique
                currentMax = grade
            end if
            if |cliquePeers| > 0 then
                cliqueHeap.push(internGrade,newClique) // insert new clique into heap
            end if
        end for
    end while // end of one step
    adaption of $c$ // deepness-compensator; adaption dependent on strategy
    if currentMax > maxBeforeExtending then
        stepsWithoutMax ← stepsWithoutMax + 1
    else
        stepsWithoutMax = 0
    end if
end while
return [maxClique]
```

Single Heap

Initially, only one heap was used: The internal grades of all cliques were re-calculated when the deepness-compensation c was changed to focus on other (normally higher) deepnesses. Also, not re-calculating the internal grades but just increasing c was tried out. This procedure does almost not have any overhead, and extends most cliques in a certain time. Good results were found quite fast, but this approach lacks a fine-grained control of the search-process. As this is of big advantage for larger graphs, we also implemented search-procedures using a heap for each deepness, called *multiple heaps*.

Multiple Heaps

Using multiple heaps - one for each deepness - has the advantage that the internal grades can be adjusted more flexible when changing c. Re-calculations of all grades are not necessary any more. For each existing deepness, only the intern grade of the best clique of this deepness is calculated. Then, cliques of this heap are picked and extended. The extension is aborted when the internal grade of the last chosen clique is worse than the best clique of any other deepness. Then, the internal grades of the best clique of each heap are re-calculated. New cliques are not considered immediately, but only after recalculating the internal grade. Repeatedly picking a large amount of cliques from one single deepness means that we will ignore the newly created cliques while doing so. When reaching the clique-limit at such a moment, there is the threat that those newly created cliques get deleted immediately. To prevent this, chosing cliques from only one heap is also aborted after a certain amount of extended cliques.

Intern Grades

The intern grade is the sort-criterium of cliques when picking the currently most promising clique. Various calculations of the intern grade were tested. The parameters to construct the intern grade of a clique are:

$c \ldots$	deepness-compensation
$rawGrade \ldots$	$\sum$ grade of lessons within the clique
$deepness \ldots$	$\sum$ deepness of lessons within the clique
$peerDeepness \ldots$	$\sum$ deepness of peers
$gradePerDeepness \ldots$	$\frac{rawGrade}{deepness}$

The deepness-compensation c is used to shift the focus to different deepnesses during the search-process. The different grade-functions and the search-strategy they were implemented for are presented in section 3.3. The results of the comparison can be found in section 3.3.

Search-Strategies

We implemented 4 different strategies for searching the maximum weight clique, which we will present in the following sections.
Unfortunately, there do not exist commonly accepted instances for the maximum weight clique

problem. Therefore, we created our own constraint-graphs to compare the four strategies: The graphs were extracted out of the main solving-procedure, using different instances in in differently filled timetables, to diversify the constraint-graphs in terms of density, number of cliques, and spread of the grades and deepnesses of lessons.
The results of the comparison of the strategies can be found in section 3.3.

Strategy 1: Single-Heap with recalculating Grades

The first strategy uses only one heap to store cliques of all sizes.
In each step, $nrOfLessons \cdot 80$ new cliques are evaluated. After each step, the initial value of the deepness-compensation c is increased by a value dependent on the average grade of the lessons. Then, the internal grades are re-calculated: This is necessary to re-establish a correct sorting according to the current value of c. The search-procedure is aborted after no new maximum was found for a certain number of steps. If the number of cliques exceeds the clique-limit, bad-graded cliques are deleted until the number of cliques falls below the clique-limit.
Equation (3.14) shows the calculation of the intern grade in strategy 1:

$$internGrade = \frac{rawGrade^c}{deepness} \tag{3.14}$$

Stragegy 2: Single-Heap without recalculating Grades

In the second strategy, again one heap was used for all cliques. The internal grades are only calculated once when adding a new clique to the heap. This means that the cliques within the heap were graded with different values of c. After each step, c was increased: Recently created cliques profit from higher c-values, and get chosen more likely. The intern grade-calculation is the same as in strategy 1 (equation (3.14)).
There exists almost no overhead in this extension-strategy. Many cliques are revised and extended, but the search lacks a finer control. Altough mostly not reaching known maxima, this strategy leads to reasonably good results already after a short runtime.

Strategy 3: Multiple-Heap extending each Deepness

The target of the third strategy is finding the optimum. It picks and extends cliques only out of the lowest deepness that currently exists. We stop extending cliques out of this deepness when all pending cliques won't be able to be better than the currently known maximum. Then we delete all pending cliques of the lowest deepness, and switch to the next (new) lowest deepness and extend cliques out of it.
The intern grade of this strategy is the theoretically reachable maximum weight of the clique. As we extend the lowest existing deepness, there is no need for the deepness-compensation c:

$$internGrade = rawGrade + gradePerDeepness \cdot peerDeepness \tag{3.15}$$

Having the grades sorted like this, and yet having found any maximum by using a faster heuristic, one can adjust the search-parameters to find the optimum. We only extend a clique if inequation

(3.16) is true. This ensures that we still can exceed the current maximum $currentMax$:

$$rawGrade + gradePerDeepness \cdot peerDeepness > currentMax \qquad (3.16)$$

As long as inequation (3.16) is true, the current clique can possibly lead to a clique that has a higher grade than the current maximum. Extending all of these cliques ensures to find the optimum: If any lower-graded clique of the same deepness possibly would be extendable to a higher maximum, it would already have been picked and extended. This is ensured because we choose the cliques to extend sorted by the intern grade, which represents the theoretically reachable weight of a clique.

One would assume that considering the peer-grades of a clique would have a positive effect, but the contrary is correct.
Figure 3.2 shows a sample graph which demonstrates the advantage of not taking into account the peer-grades. The optimum consists of the green colored nodes (the vertices of the optimal clique are left out). Currently, two sub-cliques which can find the optimum exist: A and B, each consisting of 3 lessons.
Lets assume a deepness of 1 for each lesson and a currently known maximum of 100. The

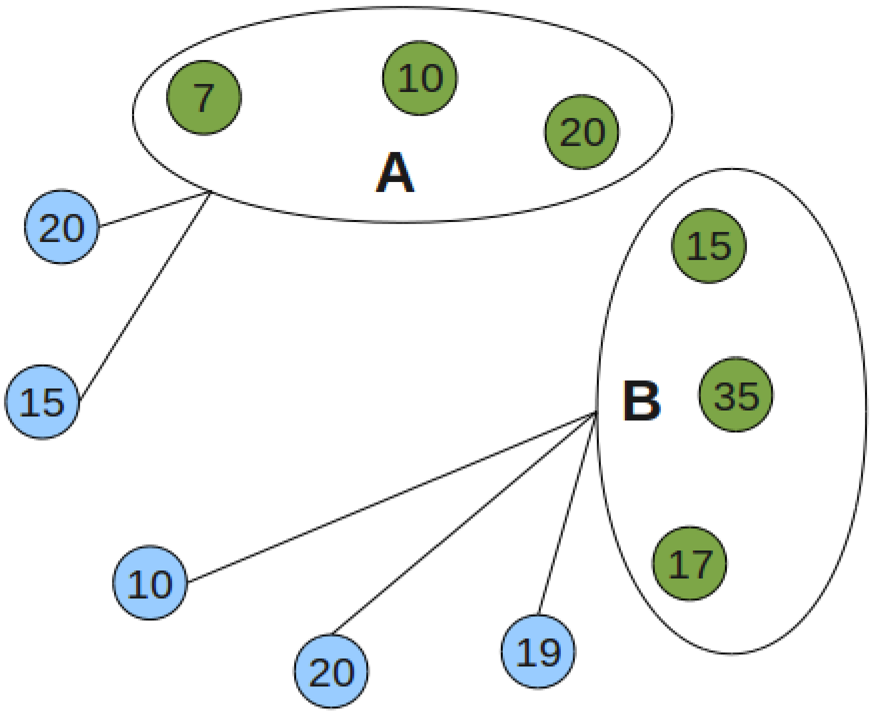


Figure 3.2: The number within the nodes represent their weight. Green nodes are part of the maximum weight clique, their edges have been omitted.

weight of the optimum is 104. We then get the grades in table 3.3. *intern grade* is the grade of equation (3.15), *sum of grades* is calculated by just summing up the grade of the clique and all of its peers. Remind that all green colored lessons of figure 3.2 represent the optimum, and are therefore interconnected. So, each lesson of B is a peer of clique A and vice versa.

Subset	grade	gradePerDeepness	nr peers	intern grade	sum of grades
A	37	12.33	5	98.67	139
B	67	22.33	6	201	153

In case of choosing the intern grade of equation (3.15) as sort-criterium, we only will pick and extend clique B. Clique A will not be chosen anymore, because its intern grade is below the known maximum. Nevertheless we can be sure that the optimum will be reached, because the higher-graded subsets of the optimum will get extended. This is also correct in the worst case, if all lessons have an equal (rather low) grade: For each clique, there exists at least one subset (or sub-clique) for which the intern grade (3.15) is higher or equal the clique-grade. This follows directly out of the fact that not all sub-cliques of a clique can have a lower grade per deepness than the original clique.

In case of using the sum of all grades, *every* subset will be extended because it can possibly reach the optimum. It is only necessary to extend (at a given deepness) at least one sub-clique of the optimum: At the next deepness this clique will be extended again until reaching the optimum.

The closer the yet-found maximum is to the optimum, the less cliques will be extended, and the faster the algorithm will be. As small cliques tend to have a high number of peers, they get high grades and are very likely to be extended. The stop-criterium of equation (3.16) only plays a role for larger cliques, so we still get a very large amount of cliques. Really implementing the current maximum as stop-criterium is only useful for examining small constraint graphs, because it requires a lot of memory.

In the constraint graphs used for comparing the different strategies, this was only possible for the two smallest instances (with the larger one having 81 vertices). To apply this strategy also to larger graphs, the stop-criterium of possibly reaching the current maximum can be exchanged with the total number of existing cliques (clique-limit). The lowest deepness will get deleted when this limit is reached. This helps staying below our memory-limit of 4 GiB, but it cannot be ensured anymore to find the optimum. Contrary to other strategies, we do not store the filled resource groups, peers and the raw grade of each clique to save memory. Instead, we repeatedly calculate this information before expanding a clique out of its nodes. This allows holding up to 12 mio. cliques in memory (dependent on the size of these cliques), at the cost of processor-time.

Strategy 4: Multiple-Heap using Window

The last strategy uses insights gained by implementing strategy 3. It was created with the aim to cut down the runtime to create a reliable, robust search, which finds solutions close to the optimum in a reasonable time. We try to focus the search on deepnesses slightly above the currently lowest deepness of all existing cliques, using a search-window.

Search-Window As a search-window we define limiting the deepnesses out of which we pick cliques to extend. Three parts of this window are defined, a lower, medium and higher part. during the search, we try to pick most cliques out of the medium part. Dependent on where we currently picked the most cliques from, we adapt c by 5 % after each step to favour the medium part of the window. This helps to spread the deepnesses we pick cliques from, and not only considering a specific deepness which seems promising at the moment.

The search-procedure still consists of *steps* described in section 3.3. The stepsize of the fourth strategy was reduced to 10, leading to a number of examined cliques per step of $10 \cdot Nr\ of\ Lesssons$.

A difficult and crucial task is to choose criteria when to delete the currently lowest deepness with all its cliques, and lift the search-window. Deleting deepnesses too early means possibly losing a maximum. Deleting them too late means wasting runtime. After testing out different approaches, the combination of four criteria turned out to be useful to initiate deleting the lowest deepness:

- *Steps without deepness-change*
  After n steps without a change of the lowest deepness that contains cliques, we delete all cliques of the lowest deepness. The finally used parameter was $n = 5$.

- *Steps without finding a new maximum*
  After m steps without finding a new maximum, we delete the lowest deepness, if the remaining number of cliques is above x percent of the current clique-limit. Finally used values are: $m = 10, x = 80$

- *Nr of existing cliques above clique-limit*
  As long as the number of existing cliques was above the clique-limit (defined in section 3.3), we delete cliques of the lowest deepness. As the focus of this approach was speed, we do not hold many cliques in memory - only about 100.000, and are not close to the memory-limit. When too many low deepnesses are deleted, there is the risk of deleting newly created cliques that not yet had the opportunity to get extended. Having enough memory left allowed us to limit the maximum number of deepnesses-deletions after one step to 2, to avoid this effect.

- *intern grade of lowest deepness below avg grade within the window*
  The intern grade rather favours smaller cliques because they have more peers. If the best clique of the lowest deepness nevertheless is below the average grade within the window, we delete all cliques of this deepness. This speeds up the search-procedure without having much impact on the quality.

Decreasing Clique-Limit With time, we decrease the clique-limit - the maximum total amount of cliques we hold in memory at a given time. It turned out that doing this slowly only slightly affected the quality of the final result, but sharply reduced the runtime.

The **intern grade** of strategy 4 is based on strategy 3. For a heuristic, the intern grade of equation (3.15) does not favour good clique-grades enough. We introduced a gradeMultiplier to compensate this and emphasize the heuristic approach. Also, c had to be introduced again:

$$internGrade = (grade \cdot gradeMultiplier + (peerDeepness \cdot gradePerDeepness))^{1+c \cdot deepness} \tag{3.17}$$

	Strategy 1	Strategy 2	Strategy 3	Strategy 4
Nr heaps	one	one	many	many
Aware of peers	no	no	yes	yes
Limit deepnesses for picking cliques	no	no	yes (one)	yes (Window)
Maintains clique-sorting	yes	no	yes	yes
Reduces clique-limit	no	no	no	yes
Adaption of c	hard-coded	hard-coded	not necessary	automatic

Table 3.12: Overview of the search-strategies

Unfortunately, the quality of results depends on the gradeMultiplier. Suitable values lie between 2 and 10. They are not bound to the instance, but rather to the specific constraint-graph of one timeslot. We did not find a direct correlation between the size, density or deviation of grades and a good gradeMultiplier. The lowest deviation was achieved with a value of 3.

We also tried out other intern grades; Equation (3.18) shows the best alternative grade-function we found. To retrieve results that are competitive with the intern grade of equation (3.17), more cliques have to be evaluated, and longer runtimes are necessary. Moreover, the variation of the result-quality is higher.

$$internGrade = \frac{grade^{(1+nrOfPeers \cdot peerExponent)\cdot(1+c \cdot deepness)}}{deepness} \tag{3.18}$$

$peerExponent$ was a value slightly above zero and c again is the deepness-compensation. The average results (weight of best clique) were about 5 % below those using the intern grade of equation (3.17) when adapting the parameters to having similar runtimes.

Table 3.12 presents the main differences of the strategies. *Aware of peers* indicates whether the internal grades considers the number or deepness of peers. *Limit deepnesses for picking cliques* is true when the cliques to extend can only be picked from specific deepnesses. *Maintains clique-sorting* refers to the correct sorting of cliques according to c. Note that for multiple heaps, the sorting is not correct for a short time because we consider newly created cliques only after finishing a *step*. The last row, *Adaption of* c, indiates how c is adapted. "hard-coded" means that after each step, c is increased by a fixed value (which is dependent on the number and grades of lessons). The intern grade of strategy 3 does not use c, because it incrementally extend each deepness. Strategy 4 automatically adapts c using the search-window, described in section 3.3

Results

As mentioned, we created our own test-samples of constraint graphs out of different instances and timetable-states. We rather chose empty timetables, because the constraint graphs tended to be larger and therefore harder to solve. The characteristics of the instances are given in table 3.13. *"Filled slots"* are the number of already filled slots of the timetable when creating and extracting the constraint graph. The column *"Std-dev"* is the standard-deviation of the deepness of the lessons within the constraint graph. A standard-deviation of 0 means that all existing lessons are of equal deepness. The *density* of a graph is calculated by comparing the existing edges with the total number of possible edges: $density = \frac{edges}{(|nodes| \cdot (|nodes|-1))/2}$. *"Size of best solution"* is the number of lessons of the best known maximum-weight clique. If the best solution is the optimum, it is marked red (italic). Except FinCollege and FinSecondary, all instances have a resource group limit.

Instance	Filled slots	Nodes	Edges	Density	Std-dev	Size of best solution
TES99	5	50	1134	0.93	2	*8*
BGHS98	3	135	8513	0.94	1.1	19
FinCollege	0	289	38438	0.92	0.6	29
FinSecondary	5	81	2575	0.79	0	*9*
KT2005-1	0	522	127689	0.94	4.9	29
KT2005-2	5	465	101212	0.94	3.7	31
StPaul	3	438	90960	0.95	6.1	54

Table 3.13: Characteristics of the search-instances used for comparing the different strategies

Table 3.14 shows the best results - the weight of the maximum-weighted clique - achieved by each strategy. Again, optima are marked red (italic). The *best solution* can be higher than any solution of strategies 1, 2 and 4, because it was achieved with other parameters that also lead to longer runtimes.
Additional information about the search-process is given in table 3.15. The column *"Nr cliques"* denotes the number of examined cliques of KT2005-1, the biggest instance, to find the given result. *"avg deviation"* is the average deviation of the best known result and the result that was found by the strategy. "1" means that the average result by a strategy is 1% below the best known solution. *"Runtime"* is the sum of the runtimes for achieving the results of all 7 constraint graphs.

	TES99	BGHS98	FinColl	FinSec	KT2005-1	KT2005-2	StPaul
Strategy 1	28665	30869	60239	5811	317390	41240	21593
Strategy 2	28665	30869	55583	5811	393271	50054	21650
Strategy 3	*28917*	-	-	*5811*	-	-	-
Strategy 4	28665	30869	62850	5811	402712	51011	22322
known Solution	*28917*	30869	63870	*5811*	403897	52305	22333

Table 3.14: Clique-search: Results achieved by the different strategies. Optima are marked red (italic)

We also conducted some tests on the deepness, as setting the deepness of all lessons to 1, or estimating the lesson-deepness, using their number of peers and the grade. While the latter

	Nr cliques	avg deviation	total runtime (s)
Strategy 1	585400	7.51	350
Strategy 2	500500	3.43	120
Strategy 3	-	-	-
Strategy 4	860000	0.78	260

Table 3.15: Additional Information about the search-process

lead to worse results in any case, for some instance better results were retrieved when setting all deepnesses to 1. However, the average grade and the variance over all results got worse. As the wrong estimation of deepnesses is bound to the instance and not to a specific timeslot, we run two searches for the first timeslot, one with a deepness of 1, and one using the normal deepnesses. For all later slots we use the setting that retrieved better results.

To find the best parameters for the finally used strategy 4, we created another set of 79 instances, which we tested using various parameter-combinations. The values of parameters mentioned above were found running these tests.
We calculated the optimum for the 41 smallest instances. For search-instances having a resource group limit, we used our search-strategy 3. For all others, we used the exact algorithm of Östergård [26], which is more memory-efficient. If we were unable to calculate the optimum in reasonable time because of the size of the graph, we used the best result ever achieved for this graph as the know maximum.
The above mentioned parameters were optimized in view of the sum of squared deviations of the 79 optima / known maxima. The best result was an average squared deviation from the optima and known maxima of 1.045, and the non-squared average deviation 0.49. This included the test of setting all lesson-deepnesses to 1. Always keeping the original lesson-deepness, the best average squared deviation was 1.67, with an average deviation of 0.7.

Discussion and Conclusion

Interesting to see is the number of cliques examined to find the maximum. This demonstrates the effectiveness of each approach of chosing promising cliques to extend. Not only a suitable intern grade is important for this, but also the priority of focusing certain deepnesses, and the choice of when to switch to a higher deepness.

Strategy 2 reached reasonable results in a very short time, whereas strategy 4, which was built to close the gap between performance and quality, turned out to be realiable in terms of finding a result close to the current maximum. Altough we still have the (minor) problem of not reliably chosing the best gradeMultiplier (see chapter 3.3), we are applying this strategy for the timetable-filling.

Strategy 3 is only interesting for examining instances to find the optimum of small instances, or get a rough idea where the optimum could be for medium instances.

A possible improvement is considering the filling of resource groups in the intern grade. We

implemented a first version: The filled groups of a clique was compared with the resource group limit and the total resource group fillings of all lessons. The intern grade was then adapted: Cliques that rather fill resource groups with loose limits were preferred. However, this first implementation did not lead to any improvements.
The results of strategy 4 are mostly close to the optima (or known maxima). The intern grade seems to be an appropriate sorting criterium. It could be further improved by analyzing the peers of a clique - at the cost of processing time. This would be even more interesting for the exact strategy 3, because it could cut down the number of cliques that are evaluated. Nevertheless, converting this strategy to a depth-first search is inevitable to allow analyzing larger graphs. The current bottleneck clearly is memory-usage, caused by the breadth-first approach. As we only used this strategy for some verifications and not in the main solving task, we omitted this implementation.
The achieved results and the suitability of parameters may be highly specialized to the instances we extracted. We make use of the deepness, which is not available normally - so the results may not be generalizeable to graphs with differing characteristics.
Because of those differences, our requirement for resource group limits and the absence of commonly accepted instances, we did not make further comparisons with other algorithms.

3.4 Filling open Resource Groups

When having found a clique, its lessons will be assigned to the timeslot. Before doing so, all open resource groups of the lessons have to be filled: to each open resource group a suitable resource has to be assigned. We will describe this procedure, as well as the case where filling all resource groups is impossible.

We have a set of open resource groups, and a set of resources which possibly fill these resource groups. They can be represented as a bipartite graph, as shown in figure 3.1: For each resource and for each open resource group, one node is created. A resource is connected to an open resource group if it can fill it. This is the case when the resource is available and belongs to the resource group. The connections are weighted. The weight consists of the resource-penalties and urgencies calculated in section 3.1. Additionally, urgency and penalty of the AvoidSplitAssignmentsConstraint is added. We also made tests only adding the avoidSplit-grades and the resource-urgency, see chapter 4.

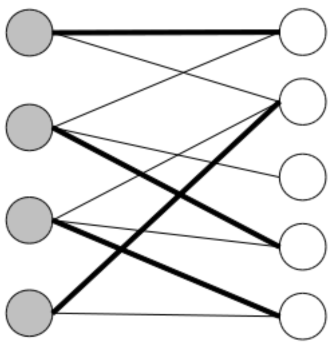


Figure 3.1: grey nodes are resource groups, hollow nodes are resources

AvoidSplitAssignmentsConstraint

This constraint aims at assigning the same resource to the OpenRole (which we convert to open resource groups) of events. The constraint is applied to an

OpenRole (identified by an ID) of all events of an event group. Assigning more than one distinct resource is penalized.
We cannot apply this constraint directly during the grading-procedure, because the open roles (resource groups) are filled *after* the clique-search. The grade/urgency resulting from this constraint will not influence the clique-search, but only the filling of open resource groups.

The penalty is applied to all resources that are not yet assigned to the open resource group of any event of the constrained event group. The urgency is calculated as follows:
We take all timeslots where any of the (still pending) sessions of the event group is available: $nrSessionSlots$. We then look in how many of these slots each resource of the resource group is available: $commonSlots$. These values form the ratio:

$$ratio = \frac{commonSlots}{nrSessionSlots} \tag{3.19}$$

The final urgency is calculated as given by equation (3.1). If the constraint is hard and one resource is already assigned, this resource will be required by the session. The open resource group is considered as closed in this case, but can be re-opened by clearing all assignments.

Maximum-cardinalty maximum weight matching

Having constructed the weighted bipartite graph, we will search the most suitable matching that covers as many open resource groups as possible: This is the **maximum-cardinality maximum weight matching** of the bipartite graph. We use an implementation by Joris van Rantwijk (available under the GNU Lesser General Public License[1]) of the matching-algorithm that was described 1986 in *“Efficient algorithms for finding maximum matching in graphs“* [13]. The documentation of the algorithm briefly notes the main principles as follows:

It is based on the "blossom" method for finding augmenting paths and the "primal-dual" method for finding a matching of maximum weight, both methods invented by Jack Edmonds.

Impossibility of filling all open Roles (Resource Groups)

Valid cliques found during the clique-search do not necessarily mean that all open roles can be filled. This is caused by resources that belong to more than one resource group. If the resource “Teacher Miller” belongs both to the resource group “English-Teachers” as well as “Biology-Teachers”, it increases the resource group limit of both resource groups - but can only be assigned once.
To minimize such conflicts, we calculate which resource groups are subsets of other resource groups. The information which resource groups a lesson fills (see section 2.3) is then adapted: Each open resource group will not only fill exactly this resource group. Also, all resource groups that have the open resource group as a subset will be filled.
With the availability of teachers and rooms, this subsets can change. To keep this information as

[1]http://www.gnu.org/copyleft/lesser.html

accurate as possible, the subsets are calculated newly for each timeslot.

Filling the open roles of all lessons will fail in the following case:
Assume we have two resource groups rg_1 and rg_2, and the following statements hold:

$$\begin{aligned}(rg_1 \not\supseteq rg_2) \wedge (rg_1 \not\subseteq rg_2) = \mathit{true} \\ rg_1 \cap rg_2 \neq \emptyset\end{aligned}$$

Resources of $rg_1 \cap rg_2$ will both increase the resource group limits of rg_1 and rg_2. Still, open roles of rg_1 and rg_2 will be considered to only fill their own resource group, because none of this resource groups is a subset of the other. Assume that the chosen lessons - found by the clique-search - fill rg_1 and rg_2 exactly. If an open resource group of rg_1 is filled with a resource out of $rg_1 \cap rg_2$, it is impossible to fill all open resource groups of the resource group rg_2 . This would hinder assigning all lessons of the chosen clique. Not assigning all lessons would possibly mean to not assign the best-weighted clique: There can (and almost always do) exist cliques that do not violate the resource group limit, and have a higher weight than the original clique substracting the weight of the not-assigned lesson.
To avoid this effect, the resource group limit of the conflicting resource group is reduced by one, and the clique-search with then trying to fill all open resource groups is repeated. This is done until no more conflicts exist.
Letting available resources fill only one resource group would be problematic, because this would make the resource group limits much tighter than they are in reality.

3.5 Higher-Level Solving Strategies

As higher-level solving strategies we understand algorithms that are one layer above the before mentioned procedure. “Filling one timeslot” is a basic action for this algorithm. Possible choices are which timeslots to fill first, or which sessions or resources to re-assign.
The grades of resources and lessons aims at considering their urgency. This is not always predicted correctly, and also the clique-search-procedure will not always reveil the best result possible. We can easily end up in a timetable-state where it is impossible to assign all pending lessons. Iteratively deleting some resources, events, lessons or timeslots and refilling/reassigning them helps getting out of such a timetable-state, and to diversify the results.

General filling Strategies

We distinguish three different strategies, which differ in how the timeslots that have to be filled are chosen, and whether we try to completely fill a timeslot or not.

Incremental Filling The incremental filling picks and fills the timeslots ordered by their occurance: Monday 1st Slot, Monday 2nd slot and so on. We try to fill each chosen slot completely.

Tetris Filling In the tetris-filling, the next slot to fill is chosen out of the day-timegroup that currently has the fewest lessons assigned. Again, each slot is filled completely.

Single Filling We here give up our basic algorithm, and instead simulate the procedure described in [24] and in chapter 1.4: We pick a meeting, and assign it to the most suitable timeslot within the timetable. To do so, we grade all timeslots and store which lesson had the highest grading in any timeslot. We then assign this lesson to the slot where it achieved its highest grade. This takes a lot of time because our algorithm is not aimed at doing so: for assigning one single lesson, we have to grade all lessons in all timeslots.

Refilling Strategies

We defined a set of different refilling-strategies. When having computed an initial timetable, we randomly choose one refilling-strategy by implementing a roulette-wheel selection. The probability of choosing this strategy again will be increased if it improved the current result.
The basic action before refilling the timetable is to remove sessions from timeslots. If we remove a resource from a timeslot, we will always delete the session that occupies the resource in this timeslot. When speaking about a *probability* of removing a resource, we mean that each session occupying this resource will be removed with a certain probability. This sometimes helps coping with tight resource-assignments.
We will now present the different refilling-strategies. The main difference is how to choose which sessions to delete, and how to refill the timetable: incremental or “tetris-like”. This can be changed for each refilling strategy. When “deleting random items”, we will delete random timeslots, events (the sessions of these events) and random resources, to get some slack for a higher diversity when refilling the timetable.

Reassign conflicting Resources We delete assignments of conflicting resources (see chapter 3.5) with a probability of 60. Also, some random items get deleted, before filling the timetable again.

Refill slots with few lessons The initial filling strategies often end up in assigning much more lessons to the first timeslots, letting possible slack in later timeslots unused. The following strategy helps distributing resource- and lesson-usage more equally over the timetable: The number of resources that are used in a timeslot is counted using the deepness of the assigned sessions. Then, the timeslots with the least resources - besides some random items - are cleared and refilled.

Assign unassigned events When having pending/unassigned events, we look which resources these events require. We delete all assignments of those resources with a default probability of *60*, and furthermore delete random items.
The difference to reassigning conflicting resources is that this strategy focusses on assigning events (lessons) by clearing all resources that an unassigned event requires, not only its conflicting resources.

Refill random items Here we remove random items and try to reassign them by again filling the timetable.

Reassign resources with high penalty The penalties of each resource are summed up. resources with a high penalty (and some random items) are deleted; then the timetable is filled again

Reassigning events with high penalty In this strategy, the penalties are grouped by events. Only event-constraints will be considered. Again, bad events and random items are deleted, and the timetable gets filled.

Iteratively refill time groups We start at a given day: All assignments to timeslots of this and the next cay are removed - the next day given friday is monday. Then, the given day is incrementally filled. We move on to the next day, until we again are at the day we initially deleted. Tests were conducted with choosing the next day consecutively or random.

Iteratively refill timeslots The procedure of iteratively refilling timeslots is presented in figure 3.1. It's like cleaning up the timetable with a snow-shovel, and re-filling it with a delay. The number of parallel days (shovel-breadth, 2 in our example) and the number of timeslots that lie between the currently deleted and currently filled timeslots (4 in the example) can be adjusted. The next day can be chosen consecutively and random.

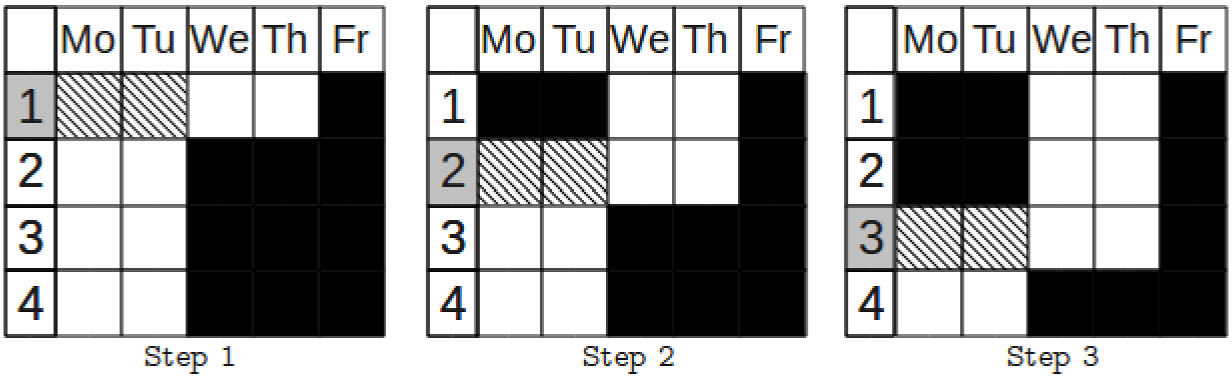


Figure 3.1: Iteratively refill timeslots: *Black slots are filled, white slots are empty, hatched slots are to be filled currently.*

Failed assignment Bonus

After filling the timetable with one or more of the above mentioned methods, there sometimes are iterations of sessions pending that we were unable to assign. To give lessons of these iterations a higher possibility to be assigned the next time, we will automatically add a certain grade during the next attempts of filling the timetable. The additional grade of such a lessons decreases when it is assigned, and further increases whenever its assignment fails again.

Local Fix

The local fix helps to resolve unavailabilities of resources with a tight assignment: During the grading, we calculate the “urgency” of resources. That is the number of pending assignments compared to the number of timeslots where an assignment is still possible. A resource is available in a timeslot if both the resource itself and any session/lesson that uses this resource are available. We call a resource “**tight**” if there are exactly as many pending assignments than timeslots available, and “**conflicting**” when there are more pending assignments than available timeslots. If - after filling a timeslot - any resource gets conflicting, we search for the reason and try to fix it. The unavailability has to be caused by a lesson that was assigned to the latest filled timeslot.

We first search for direct conflicts between any lesson assigned and the conflicting resource, which can arise when there exist lessons of a duration larger than 1. If no such lesson is found, we search for indirect resource-conflicts: We look why sessions that have the conflicting resource assigned got unavailable in a future timeslot. The unavailability can be caused by other resources that both the session and any lately assigned lesson require. Again, only lessons with a duration larger than 1 can lead to unavailabilities.

Figure 3.2 shows an example of an indirect conflict: There are the resources A, B, C and D. Assume that the resource C is tight. In the current day there are 3 timeslots (1,2 and 3) left. We just filled the first timeslot and assigned the green and the blue lesson. If the red lesson is the only

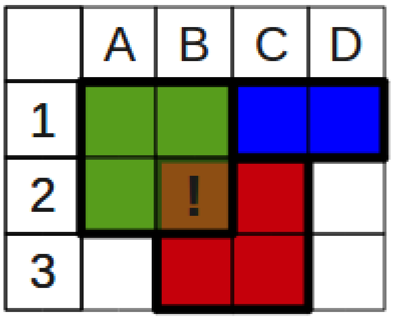


Figure 3.2: Indirect resource-conflict: *A, B, C and D are resources;* 1, 2 *and* 3 *the timeslots. Each color represents a lesson. The green and blue lesson are assigned to the current (first) timeslot; The red lesson got unavailable because of using resource B in timeslot 2. If resource C was tight already, it is conflicting now.*

one that can fill resource C in timeslot 2 and 3, we have an indirect conflict: Altough resource C is still available, it cannot be assigned because the session that requires it got unavailable. As resource C was tight already, it is conflicting now: In this state it will be impossible to assign all pending sessions to the timetable. This is caused by using resource B in timeslot 2.

We search for all resources that have direct or indirect resource-conflicts. Then, the lessons and resources that caused future resource-inavailabilities are searched and evaluated: “Bad” resources will be forbidden to be assigned. Resources that caused many inavailabilities are more likely to be bad (forbidden), and resources that are itself urgent will have a higher probability to not get forbidden. In our simple case, using resource B in timeslot 2 would be forbidden.

After forbidding using some resources in certain timeslots, we delete the lessons that would occupy such resources, and repeat the maximum-weight clique-search. If assigning the found clique causes more conflicts than the original clique, we re-assign the original clique.

Evaluate a Parameter-Set

Besides the parameters of the grading, we also parametrized higher-level decisions as the choice of the general filling strategy, or the usage of local fix. Section 4.1 explains all parameters we are going to use.
To evaluate a given parameter-set, we initially fill the timetable with the according filling strategy (see section 3.5). Then, a certain number of refilling-strategies (section 3.5) is applied. The parameters and solution of each refilling strategy are stored in a database for later analyses.

Hill Climbing

To find suitable parameter-sets for each instance, we put a hill-climbing procedure on top of the higher-level solving strategies, which is described by algorithm 3.5.1.

Algorithm 3.5.1 Hill Climbing

```
1:  // input-arguments: emptyRoundsLimit,nrBaseCamps
2:  // Create base camps
3:  $createdBaseCamps := 0$
4:  repeat
5:    $params :=$ random parameter-set
6:    evaluate $params$
7:    $createdBaseCamps = createdBaseCamps + 1$
8:  until $createdBaseCamps \geq nrBaseCamps$
9:  // Start hill climbing
10: for promising parameters-set $param$ do
11:   $emptyRounds := 0$ // emptyRounds are rounds without a new maximum
12:   while $emptyRounds$ < $emptyRoundsLimit$ do
13:     $newParams :=$ change one parameter of $params$
14:     evaluate $newParams$
15:     if result of $newParams$ better than $params$ then
16:       $params = newParams$
17:       $emptyRounds = 0$
18:     else
19:       $emptyRounds = emptyRounds + 1$
20:     end if
21:   end while
22: end for
```

When hill climbing, we allow changing all parameters that affect any hard- or soft-constraints the given instance has. We set the parameter *hardConstraint* to 10000, and allowed changing

the "*softConstraintLevel*". All parameters and the exact setup the tests were executed with are described in section 4.1.
Altough the hill-climber itself is not stochastic, a certain randomness is introduced by the use of the refilling-strategies: The strategies incorporate many random aspects as deleting a random set of resources or timeslots. Also the refilling-strategies are picked randomly, using a roulette-wheel implementation (see section 3.5). Thus, the results achieved by a specific parameter-set are usually close, but do not equal completely. To reduce deviations that could mislead the hill-climber, we use the average of the best three solutions obtained by the refilling-strategies when testing the parameter-set. This average is the reference whether a parameter-set is better than another one.
For the hill-climbing, we forbid the three refilling-strategies that achieved the worst results during the search for base camps. When applying the filling-strategy "single filling", we additionally forbid the refilling-strategies that would require completely filling single timeslots: This are the strategies "iteratively refill timeslots" and "iteratively refill time groups".
Two tabu-lists are incorporated: One of length two for picking the next refilling-strategy, and one of length three for picking the next parameter to change.
In section 4.3 we will discuss the results of the hill climbing. Resuming, the hill climbing is too less directed, which probably is caused by the large number of parameters. We implemented the **guided hill climbing** to reduce the complexity introduced by parameters.

Guided Hill Climbing

Contrary to the hill climbing, the *guided* hill climbing of algorithm 3.5.2 strictly distinguishes parameters that apply to hard-constraints (aim at completely filling the timetable), and soft-constraint parameters. We forbid parameter-changes that are of minor interest in a given stage of the solving procedure.

The guided hill climbing consists of two parts:

In the **first part**, we try to find settings for hard-constraint parameters that reliably fill the timetable. All soft-constraint parameters are set to default values and are forbidden to change, except *softConstraintLevel*: All penalties and urgencies arising from soft-constraint are multiplied with this parameter. Setting it to zero completely deactivates considering any soft constraint. We search the parameter-settings that, besides filling the timetable, are able to consider an as high *softConstraintLevel* as possible.

The **second part** then inverts the parameters that are allowed to change: Only soft-constraint parameters (without the *softConstraintLevel*) are changeable, and we try to minimize the penalty given the hard-constraint parameters found before.

Algorithm 3.5.2 shows the guided hill climbing: During the search for base camps, we increase the *softConstraintLevel* as soon as we found a valid solution. This stepwise increases the influence of all soft constraints, until no valid solution can be found any more.
For the hill-climbing itself, which equals the procedure described in section 3.5, we only allow changes of the soft-constraint parameters - the goal now is decreasing the penalty of a valid solution with different settings of the soft-constraints.

Algorithm 3.5.2 Guided Hill Climbing

```
// input-arguments: startLevel,levelStepSize,nrBaseCamps,emptyRoundsLimit
$level := startLevel$
$createdBaseCamps := 0$
$params :=$ random parameter-set
// Create base camps
only allow changes of hard-constraint parameters
repeat
  evaluate $params$
  $createdBaseCamps = createdBaseCamps + 1$
  if found a valid solution then
    $level = level + levelStepSize$
  else
    $params :=$ random parameter-set
  end if
until $createdBaseCamps \geq nrBaseCamps$
// Start hill climbing
only allow changes of soft-constraint parameters
for promising parameters-set $param$ do
  $emptyRounds := 0$ // emptyRounds are rounds without a new maximum
  while $emptyRounds$ < $emptyRoundsLimit$ do
    $newParams$ := change one parameter of $params$
    evaluate $newParams$
    if result of $newParams$ better than $params$ then
      $params = newParams$
      $emptyRounds = 0$
    else
      $emptyRounds = emptyRounds + 1$
    end if
  end while
end for
```

The guided hill climbing will be discussed and compared to the other approaches in section 4.3.

CHAPTER 4

Results and Conclusion

We will here present and discuss the achieved results. First, section 4.1 describes the test-setup and all parameters. The achieved results are presented in section 4.2. The results of the different higher-level methods can be found in section 4.3, followed by the discussion of each of these strategies in the sections 4.3, 4.3 and 4.3.
Testing and evaluating the higher-level approaches also allowed us a more detailed analysis of the refilling-methods (section 4.4) and parameters (section 4.5). The general suitability of our approach is discussed in section 4.6, followed by the conclusion in section 4.7. Finally, the appendix gives a brief overview of the characteristics and constraints of all instances we used.

4.1 Test-Setup

To receive compareable parameter-values, we transformed the weights of soft-constraints of each instance to values between 0 and 1000, so that the before highest weight then had a value of 1000. This is done by searching the maximum soft weight of an instance, and then dividing all weights by $\frac{maxSoftWeight}{1000}$. We set hardConstraint to 10000 for all instances: the relation between the filling the timetable and not violating soft-constraints is mainly expressed by the parameter *softConstraintLevel*, and the respective weights and exponents of the urgencies. Section 4.1 describes all existing parameters.
The algorithm was implemented using the Python programming language[1]. All tests were performed on a machine with an Intel Core 2 Duo 2.53GHz processor and 4GiB of memory, limiting the program to only run on one of the two cores.

Description of Parameters

We here present all parameters a parameter-set consists of. Note that, when creating a random parameter-set or changing a parameter, not all of these values are allowed to be changed. This

[1]http://www.python.org/

depends on the higher-level strategy, and is described in the section that introduces the respective strategy.

The *first kind* of parameters are those which are strongly related to the constraints. Each of these parameters has two components: a weight and an exponent, which directly influence the ratio, described in 3.1. These parameters influence the importance of certain constraints during the grading procedure.

The *second kind* of parameters are the *higher-level parameters*. They usually influence approaches that aim at filling the timetable independent from particular constraints.

Constraint-Related Parameters Each parameter of this section aims at avoiding violations of a certain constraint, and consist of two "sub-parameters": a weight and an exponent, which can be changed seperately.

Example: The parameter *BinFitHard* (see section 3.1) aims at gapless filling a resource to a block of timeslots. The block of timeslots can either have a hard-constrained border (resource-unavailability, end of day, no session available), or a soft-constrained border (soft AvoidUnavailableTimesConstraint). The bin-fitting assists the *AssignTimeConstraint* to assign all events when having tight resource-assignments: It favors lesson-assignments that maintain the possibility of filling a resource gapless into the current block of timeslots.
When having a soft-constrained border, we use the (constraint-)$weigth$ of the AvoidUnavailableTimesConstraint. For hard-constrained borders, we apply the parameter *HardConstraint* as (constraint-)$weight$. Finally, our parameter *BinFitHard* - consisting of an (external) weight and an exponent - is applied to the ratio, as already described in chapter 3.1:

$$urgency = ratio^{exponent} \cdot weight \cdot externalWeight \tag{4.1}$$

As well the penalty as this urgency is then multiplied with the parameter *softConstraintLevel* (described at the higher-level parameters). All constraint-related parameters are given in table 4.1.

Parameter	**Related Constraint**	**Described in**
resourceUrgency	AvoidUnavailableTimes	3.1
sessionUrgency	AvoidUnavailableTimes	3.1
binFitHard, binFitSoft	AvoidUnavailableTimes	3.1
spreadEventsMax, spreadEventsMin	maximum/minimum[1] of SpreadEventsConstraint	3.1
limitIdleTimes	LimitIdleTimes	3.1
limitBusyMax, limitBusyMin	minimum/maximum of LimitBusyTimes	3.1
clusterBusyMin, clusterBusyMax	minimum/maximum of ClusterBusyTimes	3.1
avoidSplit	AvoidSplitConstraint	3.4

Table 4.1: Constraint-related parameters. [1] *the minimum of the spreadEvents-constraint only exists in the altered Italy-instance*

Higher-Level Parameters These parameters represent higher-level decisions, and apply to different aspects of filling the timetable. The parameters can contain absolute values, boolean decision variables, or percentages of other absolute parameters. If not further specified, their value is absolute.

hardConstraint This parameter is introduced as weight whenever an urgency is calculated for avoiding hard-constraint violations, see section 3.1 and 4.1. Usually we set its value to 10000 and handle the relation between hard- and soft-constraints with the later described parameter *softConstraintLevel*.

deepnessBonus Give a lesson a certain default bonus (grade) for each deepness it has. Assigning large meetings is usually more difficult, so this parameter can assist if the grading-procedure itself does not sufficiently consider this.

durationBonus Lessons get this bonus for each duration above one. Some constraint-urgencies, i.e. the avoidUnavailableTimes-urgency of section 3.1, generally prefer higher or lower durations. The duration-bonus equalizes these preferations, but can also be used to favor meetings of higher durations to be assigned earlier, as having large meetings in the end of the timetable-creation is a common trap.

noUrgencyForTypes Some instances define certain school-classes on student-level and others as school-classes. This can disturb the grading procedure when calculating an urgency for each resource. Therefore, we give the possibility to skip calculating the resource-urgency (urgency of AvoidUnavailableTimesConstraint) for resources of a specific type:

- 0: Grade all resources
- 1: Do not give students a resource-urgency
- 2: Give neither students nor school-classes a resource-urgency

hardConstraintResolution The penalties we calculate consider solving hard-constraint conflicts by giving them positive values, whereas all usual penalties are negative. This can i.e. be assigning a meeting whose event-group is constrained by a hard splitEventsConstraint. The *hardConstraintResolution* defines how much percent of the *hardConstraint* solving a hard-constraint violation receives.

makeResourcesUnavailableBelowGrade This parameter adjusts making resources unavailable if their overall grade - their urgency minus the penalty - is below a certain barreer. The barreer (this parameter) is given in percent of the *hardConstraint*. It prevents bad graded resources from getting assigned by being part of a high-graded meeting, but principally aims at lowering the resource group limits when an assignment is generally undesired. For meetings (lessons) with open resource groups, it is not known in advance how desirable the assignment of the resources possibly filling the open resource group is - which we try to influence in this way.

failedAssignmentBonus Bonus given to lessons that we were unable to assign in our last attempt of filling the timetable, see section 3.5

newSessionOnHardSpreadEvents This boolean parameter is applied during the creation of sessions. If an event has a hard spreadEvents-constraint that is not part of the session by now, we can either create a new session, or add this event to the current session. Adding the event makes the session-urgency more accurate because all lessons that require the same resources are combined to one session. Not adding the event gives more accurate timeslot-availabilities of the session, because the session can not be partly unavailable, caused by the hard spreadEvents-constraint which is only applying to a part of it.

fillingMethod This is how we chose the next timeslot to fill:

- incremental filling
- tetris filling
- single filling

localFix Deactivate (0) or activate (1) the localFix

softConstraintLevel All penalties and urgencies that arise from soft constraints will be multiplied with this value. Considering any soft-constraint is deactivated with a value of 0, and 1 means that the highest soft weight will be exactly 1000 because of the weight-transformation described earlier in this section.

4.2 Best Results

Up to now, there are not many verified results available, except the few instances that already contain solutions as well. As far as we know, these are the first results published for the instances of the School Benchmarking Project. Altough some instances spring from previous scientific work, we discourage or at least warn from direct comparisons with those results. The evaluation function may have been different, or the transformation of the instances is inaccurate, as it is the case with the finish instances, see chapter 2.6.

The best results we were able to achieve are given in table 4.2. The solutions delivered with the instances, which are the only source of existing solutions, are given in column “Existing Solution”. By “Method” we understand one of the higher-level strategies, which we will discuss in section 4.3. The detailed parameters of each best solution can be found in the appendix, section A.2.

During the whole process of solving the instances, we learned to know about the suitability of parameters, and could i.e. speed up the search for suitable base camps. Thus, the runtimes necessary for achieving a certain result can not be compared directly, but instead are given in the discussion of the respective higher-level approach.
The reasons for not including and testing the australian instances are explained in section 4.6. We also skipped the artificial instances Abramson15 and FinArtificialSchool, because we focused our work on real-world problems, and no valid solution can exist for the current definition of FinArtificialSchool, as described in section 2.6.

Best Results	HC-Violations	Penalty	Existing Solution	Method	Runtime
Brazil1	0	1	104 penalty	hill-climbing	1.8s
Brazil4	4	1728	-	guided hill-climbing	46s
Brazil5	0	2375	-	guided hill-climbing	59s
Brazil6	0	2218	-	guided hill-climbing	40s
Brazil7	0	6581	-	guided hill-climbing	85s
FinHigh	0	248	-	hill-climbing	20s
FinSec	0	216	-	guided hill-climbing	40s
FinColl	5	424	-	guided hill-climbing	222s
Greece	0	0	-	guided hill-climbing	90s
GEPRO	0	19751	1HC, 566 penalty	guided hill-climbing	2330s
Italy1	0	302	-	hill-climbing	7s
KT2003	0	33565	1410 penalty	guided hill-climbing	1800s
KT2005	23	13530	1078 penalty	guided hill-climbing	1850s
StPaul	0	81996	32028 penalty	guided hill-climbing	1550s

Table 4.2: Best results we achieved

It is hard to draw a clear conclusion, as there are few existing solutions. We did not expect having so many troubles constructing full timetables (without hard-constraint violations) - the literature hardly ever describes such cases. Altough we were able to solve most of the instances without violating hard-constraints, our algorithm seems to deliver better results for smaller instances. Most difficult to solve were the instances FinCollege, StPaul and all instances from the Netherlands - for which exist valid results with a much lower penalty than we were able to achieve. We will further discuss this matter and country/instance-specific issues in section 4.6. Up to now we are unable to explain the bad result of the KT2005 instance, especially as there exists a valid solution with low penalty. As single filling did lead to even worse results, this problem is not caused by the clique-search, but perhaps by a bug in the design or or implementation of the grading process.

4.3 Comparison of High-Level Solving Strategies

We chose a small set of instances for which we compare the different solving strategies. The results are given in table 4.3, and discussed in detail in the subsequent sections. "*HC*" stands for hill climbing, "*GHC*" for guided hill climbing, and "*SF*" for single filling. The columns of "*Nr Parameter-Sets*" gives the number of parameter-sets that were evaluated for achieving the respective result. This number should not be overrated, because during the process we were able to cut down the ranges of some parameters which repeatedly yielded bad results, which also reduced the necessity of testing distinct parameter-sets. We limited the iterations of single filling because of the high runtimes.

The guided hill climbing (GHC) clearly is the most successful high-level strategy. The results achieved are either the best or close to the best known, with a reduced number of evaluated parameter-sets compared to the thill climbing. We expected the single filling to be more successful, but this may, as already mentioned, also be caused by the grading-function that was tailored

	Penalty			Nr Parameter-Sets		
Instance	HC	GHC	SF	HC	GHC	SF
FinSec	273	**216**	303	118	100	11
FinHigh	**248**	257	312	327	133	36
Italy1	**302**	312	394	244	201	101
FinColl	6HC[1]	**5HC**[1]	15HC[1]	78	48	13

Table 4.3: Comparison of High-Level Strategies [1] *HC: number of hard-constraint violations (no valid solution found)*

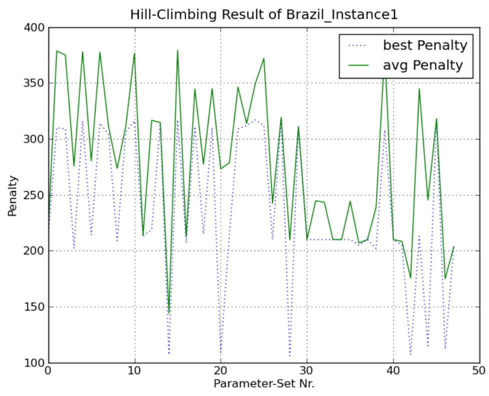
to our timetslot-based approach. The high number of hard-constraint violations of the FinCollege instance when applying single filling is caused by not being able to apply the refilling-methods "iteratively refill timeslots" and "iteratively refill timegroups", which is also going to be discussed in section 4.4.

Hill-Climbing

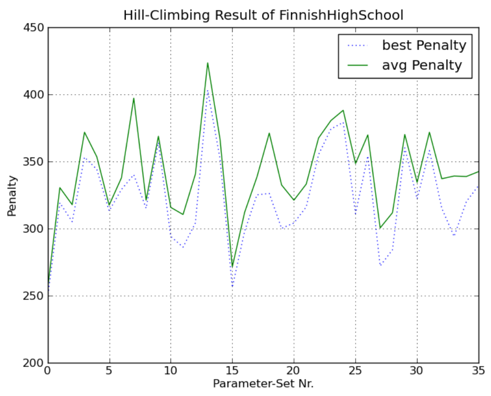
The hill climbing was implemented to find suitable parameter-sets for each instance. First, base camps (starts for the hill climbing procedure) were searched with testing random parameter-sets. For this evaluation, we limited ourselves on using 10 refilling-rounds, see section 3.5 for details. Then, the base camps are chosen in regard of the average penalty (or hard-constraint violations, if exist) of the best three results. During hill-climbing, each parameter-change is tested with 30 refilling-rounds. We forbid all refilling-methods with a negative performance, see section 4.4 for the evaluation of the refilling strategies. After 12 parameter-sets without finding a new maximum, we change two parameters per parameter-set and also increase the range by which parameters are modified. After not finding a new maximum for 20 rounds, the hill climbing is aborted.

We started testing the hill-climbing using small instances. Figure 4.1 shows two hill-climbing runs. "*avg penalty*" is the average of the three best solutions found while evaluating the given parameter-set. Remind that, as our intention is lowering the penalty, we are **trying to descend** in the given graphs. The left figure, a run of Brazil1, shows a successful run: The average penalty is decreasing with runtime. Unfortunately, such runs were an exception. We often had runs that stayed at the same level or even seemed trying to ascend, as shown on the right on instance FinHigh: In this case neither the average nor the best (lowest) penalty was improved.

Another aspect shown by the brazil instance in figure 4.1, is the solution-landscape. Given the instance Brazil1, the landscape is strongly influenced by the ClusterBusyTimesConstraint with a weight of 100. All solutions have a penalty of slightly above a multiple of 100, as can be seen in the graph: The best solutions (blue dotted) are all slightly above a penalty of 100, 200 or 300. This means that the solution-landscape has large plateaus with abrupt ascents or descents, which hindered the hill-climber from finding its way up within the limited number of steps we applied.



(a) good run of Brazil1



(b) bad run of FinHigh

Figure 4.1: Examples of hill-climbing runs. "avg Penalty" indicates the average of the best three results obtained while applying refilling-strategies. Remind that good solutions have low penalties, so the hill-climber will try to descend.

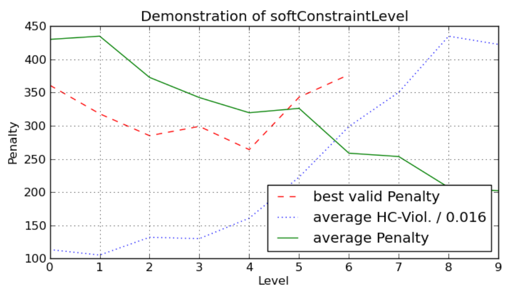
In general, the solution landscapes turned out to be rather bumpy and sensitive to the parameters. Varying one single parameter, the solution-quality often turned out to have several peaks within the solution landscape, so a "preferable direction" for changing a given value is hard to detect. A possible cause is not only the multidimensionality of the timetabling problem itself, but also the multitude of parameters we introduced. They often influence or depend on other parameters and may have added even more dimensions to the solution landscape.

The hill climbing we implemented is a high-level process. The 30 refilling-rounds are necessary to achieve reliable results, but lead to long runtimes of the overall algorithm. The runtimes for the whole process are acceptable for small and medium-sized instances, up to a total runtime of 24 hours. The border between medium-sized and large instances lies between a total event-duration of 500 and 1000, depending on event groups and on how to split the events. For larger instances, hill-climbing becomes unreasonable. In example, evaluating a parameter-set and only applying three refilling-methods takes between two and three hours with the instance GEPRO, dependent on the choice of the refilling-strategy.

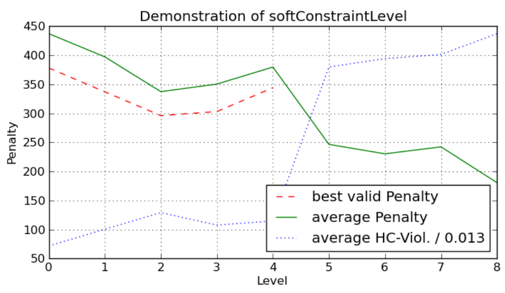
Because of the runtime we did not apply the "steepest step"-technique, as this would make even more runs of parameter-evaluations necessary. Instead, we climb the first step upward that we find, as described in algorithm 3.5.1. Because of this cut-down procedure and the bumpy solution-landscape, we were unable to escape from local minima. The hill-climbing applied with the parameters described above rather functioned as a local search for suitable parameters in a limited area of the parameter-space. Increasing the speed of the algorithm by implementing some key parts in a faster programming language would make further experiments possible.

Guided Hill-Climbing

First, we show that the adaptions of the soft-constraint level has the desired effect. Figure 4.2 demonstrates some effects of different levels with two random parameter-sets and the FinHigh-instance. The sets were evaluated for each level with applying ten refilling-rounds (see section 3.5). As expected, with increasing the level, the penalty decreases and the hard-constraint vio-



(a) Random parameter-set 1



(b) Random parameter-set 2

Figure 4.2: Demonstration of the effect of the softConstraint-level applied to the instance FinHigh

lations increase. With both parameter-sets we were able to achieve valid solutions, marked red dashed. On the left figure, the average penalty falls below the best penalty. This is caused by considering the penalty of invalid solutions when calculating the average penalty, whereas we only allow valid solutions for the best penalty. The effect that the penalty increases just before no more valid solutions can be obtained could be observed frequently (left figure: level five and six, right figure: level four). If the hard-constraints are adjusted in a way that the timetable can barely be filled, the penalty increases: During the first timeslots, too few meetings get assigned. Because of the reaction of the adaptive grading function, the urgency of tight meetings and resources increases. The high-level procedures still manage to assign all meetings to the timetable, but have to violate more constraints to do so, caused by the higher urgencies. The best hard-constraint settings are these which equally distribute this necessity of violating constraints over all timeslots.

The drawback of the guided hill climbing is that the initial soft-constraint parameters may be inadequate, and that this influences the softConstraint-level we are able to reach. Altough we were able to achieve results compareable to the normal hill climbing with examining less parameter-sets, the guided hill climbing still is too less directed. Despite the fact that we cut down the number of parameters to in between two thirds and a half of the original number, the hill climbing itself showed similar results as described in section 4.3. We therefore believe that the lowered number of evaluated parameter-sets is not caused by the more efficient and directed hill climbing, but by the strict distinction of hard and soft parameters.

Single Filling

We did not include single filling, described in 3.5, as an option for the random parameter-sets described above. The method strongly differs from our initial approach, and was therefore tested separately. It does not take timeslots as basis and fills them, but instead iteratively picks the most urgent lesson (meeting) and assigns it to the timetable.
Altough this basically is the method used by GP-Untis and FET, it can not serve as a direct comparison of the two approaches. The grading-procedure as well as the higher-level refilling strategies were developed in respect of the timeslot-based approach, and may contain pitfalls for the lesson-based approach. Furthermore, we did not implement any kind of backtracking or local search for the latter approach. The runtimes of our software are much higher for single filling - approximately by factor 10, because it is tailor-made for the timeslot based approach.

In theory, this approach should perform better for the italian instance. The problem described in section 4.6 does not arise when applying single filling, because we never completely fill a timeslot at once. However, the comparison of single filling with the other strategies, given in section 4.3, shows that single filling was the worst strategy for every instance tested, also for the italian instance. We also tested single filling on the KT2005 instance, to check whether the bad result could be caused by a flaw of the clique search. The results of single filling for KT2005, altough not tested thoroughly, were even worse than the normal filling methods. This may be rather caused by specifics of our grading procedure than by the general suitability of single filling for creating high-school timetables.

4.4 Suitability of refilling-methods

Of interest is the effect that single refilling-methods have. We therefore analyze the results collected during our search for base camps. The hill-climbing procedure is not analyzed, because we there forbid certain refilling-methods, which would influence the results.
For each parameter-set we tested, we order the methods descending by the respective best result they achieved. The suitability of a method within a parameter-set is $rank - \frac{|methods|}{2}$. Figure 4.1 presents the average method-results of the instances FinHighSchool and Brazil7. Positive values indicate that a method performed better than average, and negative that it performed worse.
It can be seen that the most suitable methods of Brazil7 are the worst ones for FinHigh and vice versa. This indicates that the suitability of a refilling method strongly depends on the instance.

The refilling-methods six, seven and ten of figure 4.1 derive from the refilling-strategies "Iteratively refill time groups" and "Iteratively refill timeslots" (see section 3.5). They differ in whether the next timeslot/group to delete is picked consecutive or randomly. These strategies maintain a certain number of empty timeslots within the timetable. All other strategies rather clear certain lessons, event groups or resources, or try to again fill timeslots with few assignments.
Evaluating the results showed that there are two groups of instances: One group that highly profits from applying the iterative refilling strategies (i.e. FinSecondary, Brazil7, FinCollege),

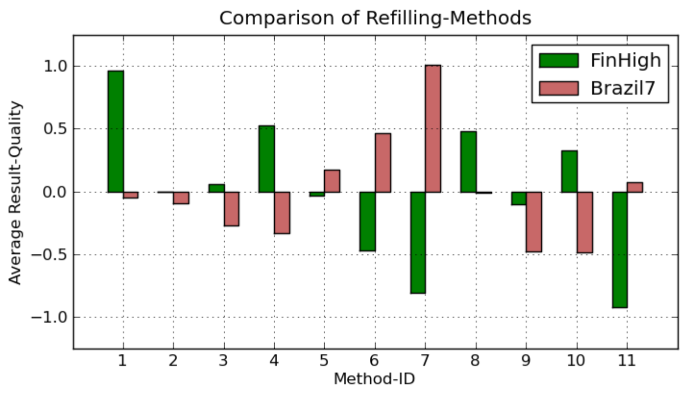


Figure 4.1: Comparison of the suitability of refilling-methods for different instances

and another group that profits from applying all other strategies (i.e. FinHigh, Brazil1, Italy). This seems to be dependent on the difficulty of completely solving the instance: The iterative refilling strategies are good in completely filling a timetable, whereas the other strategies aim at improving the solution. This fails if it is hard to create a valid solution for the instance, and results in infeasible solutions.

4.5 Discussion of suitable Parameter-Settings

We will here describe observations we made regarding the (non) suitability of specific parameters.

As can be seen in the parameters for the best solutions we achieved, section A.2, the **softConstraintLevel** roughly represents the difficulty of achieving valid solutions of a certain instance. Combined with the guided hill climbing, this parameter can assist in quickly determining the difficulty of solving a certain instance with our algorithm.

Whether activating **local fix** is desirable depends on the instance. If there are only a few or no resources that have a tight assignment, i.e. as for the instances Brazil1 and FinSecondary, the algorithm profits from resolving those local resource-conflicts. If the whole instance is hard to solve because of tight resource-assignments, i.e. as in FinCollege, the local fix has a negative effect: When locally resolving such resource-conflicts, we normally avoid assigning certain resources in favor of assigning more urgent, tighter resources. On the long run, this leads to a larger number of resources with a tight assignment, and hence should only be activated if the majority of the resources has some slack - which is having more timeslots than their workload (sum of all assignments).

The weights and exponents of the constraint-related parameters are more diverse than we expected. Still, some effects are notable: For the limitIdleTimes, exponents and weights of 1 seem suitable. The urgency sometimes compensates penalty, which is biased when having a weight that does not equal 1. We allowed the hill-climbing procedure to apply also negative parameters. As disabled or negative constraint-related parameters only appear twice in the best results - the binFitHard-weight of FinHigh and the spreadEventsMax of KT2003 - we generally consider our grading function to be appropriate, altough there surely are a lot of improvement possibilities.

4.6 Discussion of our Approach

Our approach - completely filling timeslots - is a general one, and applicable to all existing instances. Still, the results we achieved are diverse and strongly dependent on characteristics of the instance. The reason is that in some aspects, our algorithm lacks flexibility. Altough the procedure may not reach top results, we were able to solve a major part of the real-world instances existing (except the australian ones, see section 4.6). Because of the seperate grading-function, new constraints are easy to implement as long as they do not require fundamental changes of the datastructure. Also, the evaluation of existing constraints can be changed easily.

The **grading-procedure** was implemented (and completely revised) to maintain a relation to the constraint-penalties as direct as possible. However, maintaining the connection to the constraint-evaluation on the one hand and hard-constraints on the other hand, while keeping the balance between all constraints, is really tough. The grading is, besides a reliable clique-search, the core of this approach. Altough giving much attention and thought in the design, implementation and test of the grading-process, it still has many weaknesses, as can be seen by the diversity of suitable parameters for each instance, see appendix A.2. This partly may be flaws of the design, but possibly also are implementation errors of the grading function, which itself has 3500 lines of code.

The results of the **clique search** are satisfying. Altough we can not measure achieving adequate results also on larger graphs, the tests performed in section 3.3 are better than we expected. The problems we have with the larger instances, getting invalid or high-penalized results, are not caused by the clique search. Tests with single filling - which completely bypasses the clique-search - lead to even worse results.

We will now describe the two main instance-specific problems we had with the italian and the australian instances, followed by the description of implementation-specific issues and improvement possibilities.

Italian Instances

The new formulation of the italy-instance, having a spreadEvents-minimum for single timeslots, is problematic for our algorithm. As we pick and fill one single timeslot, we can either favor all or no events of the event group the spreadEvents-minimum applies to. Only favoring some of these events would torpedoe the idea of letting the clique-search decide which lessons to choose.

Altough this was not a big problem when solving the italy-instance, it is a general disadvantage of this approach, and may present a bottleneck for future instances having similar constraints.

Problems with australian instances

The australian instances were neglected by us because of two main points:

First, the current definition of the limitWorkload-constraint seems unnatural, as discussed in section 2.6. This constraint is one of the key points and bottlenecks the australian instances have.

Second, our algorithm is not suitable for dealing with the multitude of open roles as they appear in those instances. As we close the open roles after the search for the maximum weight clique, we lose control. We then have to fill the open roles of the chosen lessons, no matter how preferable assigning some of the resources is. Even more, when the open roles are bound by hard avoidSplit-constraints, we prematurely close this open role with certain resources. We lose a lot of flexibility this way, and have only very limited control when dealing with this bottleneck. This is an implementation-specific issue, which will also described in the next section.

Implementation-specific aspects

As mentioned in the discussion of the australian instances, **closing open roles** when assigning lessons to timeslots is a clear disadvantage. Instead, the roles could be left open, and closed later. Kingston [19] described closing open roles for meetings that have times assigned in 2010.

The necessity of the parameter **newSessionOnHardSpreadEvents** is not induced by the problem or any instance, but by our definition of sessions. In example, the the instance FinHighSchool has the event groups gr_C003 and gr_C004. Each three events of this event groups exactly require the same resources. If we combine them to the same session, the session-availability is inexact: We will have to mark the session available if at least one of its iteration is available - altough a major part of the session could be in fact inavailable: The availability of one iteration of event group gr_C003 can lead to having many timeslots left, altough there are in reality very few timeslots left for event group gr_C004. This disturbs the session-urgency, which does not distinguish which of its iterations are available in the timeslots. On the other hand, creating two distinct sessions is also not precise, because both sessions are unaware of each other, and would have a too low urgency. The session-urgency would have to consider which iterations of a session are available in specific slots to being more exact. This issue is not relevant for the instances of brazil, australia and the italy-instance.

The **runtime** of the whole solving process is insatisfactory for larger instances. This could be reduced by comparing specifics of the instances and further restricting the range of some parameters, or conducting tests with parameter-sets that performed good on similar instances. Also, the core functions of the algorithm - the grading and the clique-search - could be implemented in a faster programming language. Still, we are very content having chosen python as programming language. We often had to restructure our program because of changes of the format or knew insights, which was facilitated by the flexibility of python.

Because of the long runtimes, we had to limit the number of steps of the **hill climber**. Performing the “steepest step” instead of the “first step” we implemented could improve the directedness and the ability of escaping local minima.

We did not implement a classical local or neighborhood search on a meeting- or event-based level, because the variety of constraints and their evaluation is not only time consuming but also hard to implement: Additional to our grading procedure, delta-functions for all constraints would have been necessary to achieve reasonable runtimes. For the same reasons, and because of the amount of meetings we assign simultaneously, we left out a more profound backtracking than the local fix of section 3.5. Whereas a more sophisticated backtracking would have helped for filling the timetable, local searches are often helpful for polishing existing solutions, or even creating whole timetables, dependent on the neighborhood definition.

4.7 Conclusion

In this work, we developed an algorithm for solving high-school timetabling problems. We used the newly available instances of the School Benchmarking Project, which was started in 2007. These real-world instances were collected from scientists from various countries, and so the constraints and possible bottlenecks drastically vary.

The algorithm we developed takes timeslots as basis. A non-full timeslot is chosen, and the favorability of all meetings that can be held in this timeslot is graded. We then construct a weighted graph out of the meeting-grades, where all meetings that can be held simultaneously are connected. A heuristic maximum-weight clique search with some other side-constraints is performed. The clique found represents a set of meetings which are then assigned to the specific timeslot. This are the two core functions of our approach: The grading-procedure and the maximum-weight clique search. On top of this basic procedures, we implemented some strategies to refill the timetable, in example by reassigning resources that cause high penalty. To find suitable parameters for an instance, we applied a hill climbing procedure.

Not only designing and implementing the algorithm, but also parsing the instances was a challenge. Partly, the instances contained errors, and during our work, fundamental changes of the format were introduced with a new version. Still, we highly appreciate the commitment of the authors. Up to now, the field of high-school timetabling clearly lacks common, interchangeable real-world instances that are not bound to a specific institution or country. Altough some points of the file format may be worthy of discussion, we hope and expect the School Benchmarking Project to have a large, positive impact to the field of high school timetabling.

Because of some limitations of our approach and the implementation we chose, combined with the diversity of the instances, we were not able to create valid timetables for all instances available. A general algorithm for solving high-school timetabling problems has to bring a

high flexibility and the ability to shift its focus to the varying bottlenecks this problem possibly contains.
We did not fully exhaust the timeslot-based approach: There are some implementation details that could be improved, above all separating closing open roles from assigning meetings to a timeslot, which would make the algorithm better applicable to the australian instances. But there are also some approach-inherent disadvantages, in example as encountered in the italian instance. We believe that the strategy of picking single meetings and assigning those to suitable timeslots yields more flexibility to be adapted to the diversity of bottlenecks. Especially, it facilitates backtracking, which we only applied in a very limited extent.

Bibliography

[1] L. Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52:31–38, 1994.

[2] Egon Balas and William Niehaus. Optimized crossover-based genetic algorithms for the maximum cardinality and maximum weight clique problems. *Journal of Heuristics*, 4:107–122, 1998.

[3] G. S. Bello, M. C. Rangel, and M. C. S. Boeres. An approach for the class/teacher timetabling problem using graph coloring. In Edmund K Burke and Michel Gendreau, editors, *The Practice and Theory of Automated Timetabling VII*, August 2008.

[4] Immanuel M. Bomze, Marcello Pelillo, and Alessio Massaro. A complementary pivoting approach to the maximum weight clique problem. In *SIAM Journal for Optimization*, volume 12, pages 928–948, 2002.

[5] Edmund Kieran Burke and Michel Gendreau, editors. *The Practice and Theory of Automated Timetabling VII*, PATAT, 2008.

[6] Edmund Kieran Burke and Sanja Petrovic. Recent research directions in automated timetabling. *European Journal of Operational Research*, 140(2):266–280, 2002.

[7] Edmund Kieran Burke and H. Rudova, editors. *Practice and Theory of Automated Timetabling VI*, volume 3867 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Stanislav Busygin. A new trust region technique for the maximum weight clique problem. *Discrete Appl. Math.*, 154:2080–2096, October 2006.

[9] D. de Werra. Construction of school timetabley by flow methods. *INFOR - Canadian Journal of Operations Research and Information Processing*, 9:12–22, 1971.

[10] D. de Werra. Some combinatorial models for course scheduling. In Edmund K Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 296–308. Springer, 1996.

[11] S. Even, A. Itai, and A. Shamir. On the complexity of timetabling and multicommodity flow problems. *SIAM Journal of Computation*, 5:691–703, 1976.

[12] Cristina Fernández and Matilde Santos. A non-standard genetic algorithm approach to solve constrained school timetabling problems. In Roberto Moreno-Díaz and Franz Pichler, editors, *Computer Aided Systems Theory - EUROCAST 2003*, volume 2809 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2004.

[13] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–38, March 1986.

[14] C. C. Gotlieb. The construction of class-teacher time-tables. In C. M. Popplewell, editor, *Proc. IFIP Congress 62*, volume 4 of *Information Processing*, pages 73–77. North-Holland Publishing Co., 1963.

[15] Jantien Hartog. *Timetabling on Dutch High Schools*. PhD thesis, TU Delft, March 2007.

[16] D. S. Johnson and M. R. Garey. *A Guide to the Theory of NP-Completeness*. Computers and Intractability. W.H. Freeman and Company, 1979.

[17] David Stifler Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, June 1973.

[18] Richard Manning Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103., 1972.

[19] Jeffrey Kingston. Resource assignment in high school timetabling. *Annals of Operations Research*, pages 1–14, 2010. 10.1007/s10479-010-0695-0.

[20] Jeffrey H. Kingston and Tim B. Cooper. The complexity of timetable construction problems. In Edmund K Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 283–295. Springer, 1996.

[21] Deniss Kumlander. A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search. In Thi Hoai An Le and Dinh Tao Pham, editors, *5th International Conference on Mod- elling, Computation and Optimization in Information Systems and Management Sciences: MCO '04*, pages 202–208. Hermes Science Publishing Ltd., July 2004.

[22] Deniss Kumlander. Improving the maximum-weight clique algorithm for the dense graphs. In *Proceedings of the 10th WSEAS International Conference on COMPUTERS*, pages 938–943, July 2006.

[23] Wojciech Legierski. Search strategy for constraint-based class-teacher timetabling. In Edmund K Burke and P. de Causmaecker, editors, *The Practice and Theory of Automated Timetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2002.

[24] Michael Marte. *Models and Algorithms for School Timetabling – A Constraint-Programming Approach.* PhD thesis, Ludwig-Maximilians-Universität München, July 2002.

[25] Michael Marte. Towards constraint-based school timetabling. *Ann Oper Res*, 155:207–225, August 2007.

[26] Patric R. J. Östergård. A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8(4):424–436, 2001.

[27] Gerhard Post, Samad Ahmadi, Sophia Daskalaki, Jeffrey Kingston, Jari Kyngas, Cimmo Nurmi, and David Ranson. An xml format for benchmarks in high school timetabling. *Annals of Operations Research*, pages 1–13, February 2010.

[28] Gerhard Post, Samad Ahmadi, Sophia Daskalaki, Jeffrey H. Kingston, Jari Kyngas, Cimmo Nurmi, Henri Ruizenaar, and David Ranson. An xml format for benchmarks in high school timetabling. In *Proceeding of the 7th international conference on the practice and theory of automated timetabling*, PATAT, 2008.

[29] Haroldo Santos, Eduardo Uchoa, Luiz Ochi, and Nelson Maculan. Strong bounds with cut and column generation for class-teacher timetabling. In *Proceedings of the 7th international conference on the practice and theory of automated timetabling*, PATAT, 2008.

[30] Haroldo G. Santos, Luiz S. Ochi, and Marcone J.F. Souza. A tabu search heuristic with efficient diversification strategies for the class/teacher timetabling problem. In Edmund K Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling V*, volume 3616 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2004.

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

[32] G. Schmidt and T. Ströhlein. Timetable construction - an annotated bibliography. *The Computer Journal*, 23(4), 1979.

[33] Alok Singh and Ashok Kumar Gupta. A hybrid evolutionary approach to maximum weight clique problem. *International Journal of Computational Intelligence Research*, 2(4):349–355, 2006.

[34] Marcone Jamilson Freitas Souza, Nelson Maculan, and Luis Satoru Ochi. A grasp-tabu search algorithm for solving school timetabling problems. In M Resende and Marcone Jamilson Freitas Souza, editors, *Meta- heuristics: Computer Decision-Making*, pages 659–672. Kluwer Academic Publishers, 2003.

[35] Robertus Johannes Willemen. *School timetable construction: Algorithms and complexity.* PhD thesis, Technische Universiteit Eindhoven, May 2002.

APPENDIX A

Appendix

A.1 Overview of the Instances and their Constraints

Tables A.1 and A.2 show which constraints occur in which instances. “both” means that the constraint occurs as soft- and as hard-contraint within this instance. The AssignResource-, AssignTime- and AvoidClashConstraints have been left out as they equal for all Instances described here.

Instances 1/3	FinHigh	FinSec.	FinColl.	StPaul	Greece	KT2003	KT2005	GEPRO
spreadEvents	hard	hard	hard	hard	hard	hard	hard	hard
preferTimes	hard	hard	hard	hard	-	hard	hard	hard
distrib.SplitEvents	-	-	-	-	-	-	-	-
splitEvents	hard[1]	hard[1]	hard[1]	hard[1]	hard[1]	hard[1]	hard[1]	hard[1]
limitIdleTimes	soft	soft	soft	soft	-	soft	soft	soft
clusterBusyTimes	-	-	-	-	-	soft	soft	soft
limitBusyTimes	soft	soft	soft	-	-	soft	soft	soft
limitWorkload	-	-	-	-	-	-	-	-
avoidUnav.Times	both	both	hard	soft	hard	both	both	soft
avoidSplitAss.	-	-	-	-	-	-	-	-
preferResource	-	-	-	hard	-	hard	hard	hard
country	Finland	Finland	Finland	England	Greece	Netherl.	Netherl.	Netherl.
total event-duration	319	306	854	1227	372	1203	1272	2675
open roles for	-	-	-	rooms	-	rooms	rooms	-

Table A.1: Constraints in instances. [1] splitting events is forbidden by this instance

Instances 2/3	Brazil1	Brazil7	Italy1	TES99	BGHS	SAHS96	Abr15
spreadEvents	hard	hard	both	soft	soft	soft	hard
preferTimes	hard	soft	hard	-	hard	hard	hard
distrib.SplitEvents	soft	soft	-	hard	hard	hard	-
splitEvents	hard	hard	hard	hard[1]	hard	hard	hard[1]
limitIdleTimes	soft	soft	soft	-	-	-	soft
clusterBusyTimes	soft	soft	-	-	-	-	-
limitBusyTimes	-	-	soft	soft	soft	soft	-
limitWorkload	-	-	-	hard	hard	hard	-
avoidUnav.Times	hard	-	both	hard	hard	hard	-
avoidSplitAss.	-	-	-	soft	both	soft	-
preferResource	-	-	-	hard	hard	hard	-
country	Brazil	Brazil	Italy	Australia	Australia	Australia	Artificial
total event-duration	75	500	133	1564	806	-	450
open roles for	-	-	-	rooms,teach.	rooms,teach.	rooms,teach.	-

Table A.2: Constraints in instances. [1] splitting events is forbidden by this instance

Instances 3/3	Brazil4	Brazil5	Brazil6
spreadEvents	hard	hard	hard
preferTimes	soft	soft	soft
distrib.SplitEvents	soft	soft	soft
splitEvents	hard	hard	hard
limitIdleTimes	soft	soft	soft
clusterBusyTimes	soft	soft	soft
limitBusyTimes	-	-	-
limitWorkload	-	-	-
avoidUnav.Times	hard	-	hard
avoidSplitAss.	-	-	-
preferResource	-	-	-
country	Brazil	Brazil	Brazil
total event-duration	300	325	350
open roles for	-	-	-

Table A.3: Constraints in instances

A.2 Parameters for finding the best Results

Tables A.4 and A.5 present the settings of parameters used when finding the respective best result. Beware that for a certain instance, only the parameters of constraints this instance has - see section A.1 - are of relevance.

Instances 1/2	FinHigh	FinSec	FinColl	StPaul	Greece	KT2003	KT2005	GEPRO
hardConstraint	10000	10000	10000	10000	10000	10000	10000	10000
deepnessBonus	688	-2400	2200	2000	0	1800	0	0
durationBonus	492	1700	500	-400	0	-1300	0	-834
noUrgencyForTypes	1	1	0	1	0	1	0	2
hardConstraintResol.	0	0.9	0.1	0.1	0.1	0.1	0.1	0.1
makeResourcesUnav.Below	-0.16	-0.9	-1	-0.5	-2	-0.5	-1.3	-0.9
failedAssignmentBonus	16	30.3	1000	1000	1000	1000	1000	1000
newSessionOnHardSpread.	0	0	0	1	1	1	1	1
fillingMethod	1	1	1	0	0	1	1	1
localFix	0	1	0	1	0	1	1	1
softConstraintLevel	1	3.7	0	0.1	0.2	0.3	0	0.7
resourceUrgency: weight	1.4	4.65	8.85	3.3	1.65	7.2	7.2	1.2
resourceUrgency: expon.	0.6	1.4	1.55	1.25	2.15	3.05	1.55	0.8
sessionUrgency: weight	6.8	8.1	5.55	8.4	2	7.8	7.8	4.05
sessionUrgency: expon.	1.6	0.65	3.05	2.75	1	0.95	1.1	0.8
spreadEventsMin: weight	1	1	1	1	1	1	1	1
spreadEventsMin: expon.	1	1	1	1	1	1	1	1
spreadEventsMax: weight	1.1	1.2	0.5	2	1.3	-0.3	1.3	2
spreadEventsMax: expon.	1.4	1.3	1.5	1	1.3	1	1	1.5
binFitHard: weight	0	0.55	0.65	0.1	0.5	0.6	0.7	0.55
binFitHard: expon.	3.5	5	3	2.2	4	4	5	2
binFitSoft: weight	0.85	0.3	0.1	0.1	0.1	0.1	0.1	0.1
binFitSoft: expon.	5	2.5	4	4	4	4	4	4
limitIdle: weight	1.4	1	1	1	1	1	1	1
limitIdle: expon.	1	1.2	1	1	1	1	1	1
limitBusyMax: weight	1	1	1	1	1	1	1	1
limitBusyMax: expon.	3	3	3	3	3	3	3	3
limitBusyMin: weight	1	1	1	1	1	1	1	1
limitBusyMin: expon.	1	1	1	1	1	1	1	1
clusterBusyMaxWAss: weight	1	1	1	1	1	1	1	1
clusterBusyMaxWAss: expon.	2	2	2	2	2	2	2	2
clusterBusyMaxOverf: weight	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
clusterBusyMaxPress.: exp.	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
clusterBusyMax : weight	1	1	1	1	1	1	1	1
clusterBusyMin: weight	1	1	1	1	1	1	1	1
clusterBusyMin: expon.	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
avoidSplit: weight	1	1	1	1	1	1	1	1
avoidSplit: expon.	2	2	2	2	2	2	2	2

Table A.4: Detailed parameter-settings for achieving the best solutions, section 4.2

Instances 2/2	Brazil1	Brazil4	Brazil5	Brazil6	Brazil7	Italy1
hardConstraint	10000	10000	10000	10000	10000	10000
deepnessBonus	-1200	0	0	0	-1500	1800
durationBonus	-962	0	0	0	580	-700
noUrgencyForTypes	0	0	0	0	1	0
hardConstraintResol.	0.1	0.1	0.1	0.1	0.1	0.1
makeResourcesUnav.Below	-1.6	-1.7	-0.6	-1.6	-0.5	-1
failedAssignmentBonus	1000	1000	1000	1000	1000	1000
newSessionOnHardSpread.	0	0	1	1	1	1
fillingMethod	0	0	1	0	0	1
localFix	1	1	0	1	1	1
softConstraintLevel	2.9	0	6	8.4	0.3	4
resourceUrgency: weight	8.4	2.55	5.4	4	9	1.5
resourceUrgency: expon.	0.55	2.6	1.4	1	2.4	1.25
sessionUrgency: weight	5.9	2	2	1.85	8.35	2.7
sessionUrgency: expon.	1.6	1	1	0.85	0.5	1.7
spreadEventsMin: weight	1	1	1	1	1	0.7
spreadEventsMin: expon.	1	1	1	1	1	1.2
spreadEventsMax: weight	1.0	1.1	1.8	0.9	0.4	1.2
spreadEventsMax: expon.	1.1	0.9	1.2	1.5	1	1.2
binFitHard: weight	0.3	0.35	0.3	0.65	0.15	1
binFitHard: expon.	2.9	3.2	2.4	4.8	4.4	4.6
binFitSoft: weight	0.3	0.1	0.1	0.1	0.1	0.4
binFitSoft: expon.	3.7	4	4	4	4	2
limitIdle: weight	1	1	1	1	1	0.6
limitIdle: expon.	1	1	1	1	1	1.4
limitBusyMax: weight	0.8	1	1	1	1	1
limitBusyMax: expon.	3.5	3	3	3	3	2.5
limitBusyMin: weight	1	1	1	1	1	1
limitBusyMin: expon.	1	1	1	1	1	0.6
clusterBusyMaxWAss: weight	1	1	1	1.2	1.2	1
clusterBusyMaxWAss: expon.	2	2	2	2	1.8	2
clusterBusyMaxOverf: weight	1.5	1.5	1.5	1.5	1.6	1.5
clusterBusyMaxPress.: exp.	2.5	2.5	2.5	2.4	2.4	2.5
clusterBusyMax : weight	1	1	1	1.2	1	1
clusterBusyMin: weight	1	1	1	1	1	1
clusterBusyMin: expon.	1.2	1.2	1.2	1.2	1.2	1.2
avoidSplit: weight	1	1	1	1	1	1
avoidSplit: expon.	2	2	2	2	2	2

Table A.5: Detailed parameter-settings for achieving the best solutions, section 4.2