UPPSALA
UNIVERSITET

UPTEC IT 12 001
Examensarbete 30 hp
February 2012

Toward Automated Timetabling at TekNat

Henning Hellkvist
William Sjöstedt

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Toward Automated Timetabling

Henning Hellkvist and William Sjöstedt

This is a report on our approach to use constraint programming to automatically solve the Uppsala University timetabling problem at the Faculty of Science and Technology (TekNat). The project has been successful as the program produced can quickly solve the problem of assigning courses to specific times and rooms. The task of scheduling the entire TekNat problem at once is in most cases unsuccessful, but the problem can be split up over multiple departments, each scheduled one by one. Scheduling an instance of roughly 40 courses which is the size of the IT department takes less than a minute while solving data resembling the entire TekNat instance of about 200 courses takes about 5 minutes if successful. However, the model has only hard constraints and may disregard solutions that would be acceptable when reviewed by human scheduling officers. Therefore the program will not work in all problem instances, and no solution is ever guaranteed.

The report contains a brief introduction on constraint programming in general as well as a more detailed overview of the specific parts we have utilized of the constraint programming library we choose to use for this task; Gecode. Furthermore we will present a fully functional prototype for a user interface we designed during the project that can benefit both teachers and schedulers alike. We present the mechanics of our program and discuss its advantages and drawbacks as well as outlining what areas a continuation of this project should focus on.

Handledare: Pierre Flener
Ämnesgranskare: Justin Pearson
Examinator: Arnold Pears och Anders Berglund
ISSN: 1401-5749, UPTEC IT 12 001
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Scheduling tasks belong to a set of problems where resource-requiring activities are to be arranged in a sequence such that there are no collisions between activities who share resources. An activity could be anything from a lecture at a school to processes in a computer, and resources may be personnel, facilities, processors etc. The scheduling problems have, as of today, no general, scalable algorithm for solving them within a predictable and reasonable amount of time. For an asymptotic amount of activities with sufficiently many resource collisions, and with little other heuristics, finding a solution may be no more efficient than trial and error search.

In the particular case this report describes the problem is that of scheduling lectures, labs and lessons (called *events*) of all courses given at TekNat at Uppsala University. This involves assigning time slots and allocate lecture halls for each event whilst making certain all constraints of the events are satisfied. These constraints may be that no teacher or student has two or more events at the same time or that there are at most one event per classroom at a given time, but may also be more general preferences set down by the teachers themselves or other less obvious restrictions. Our automated timetabling application handles all the natural constraints and also specific constraints on courses and events, to ensure that a preferred structure of a course is maintained and to allow teachers to exclude certain dates when they are otherwise preoccupied.

In recent years a programming paradigm known as CP has been researched and developed for solving scheduling- and other combinatorial problems, which we give a short introduction to in Section 2. Section 3 covers the particular scheduling instance at TekNat and what approach has been taken to solve the problem. In Section 4 a description of the actual structure of the implemented model can be found, both of the current version and older versions. Section 5 displays execution times and other information from running our scheduler, as well as descriptions of and motivations for the tested scheduling instances we use. Conclusions and discussions about the results can be found in Section 6, and proposed future work is found in Section 7.

2 About Constraint Programming

CP is a programming paradigm aimed to solve combinatorial problems. It is a form of declarative programming, which means that instead of describing the algorithm and step by step methods for solving a problem, it aims at describing what the problem is. The problem is often formulated as a set of integer variables, each related to a set of possible values to take, and restrictions upon and between these variables.

How to model a problem in CP and subsequently find solutions to it can be read in Sections 2.1 and 2.2 respectively. The CP library Gecode is covered in Section 2.3.

2.1 Modeling

To clarify a bit how CP works, and also introduce a bit of terminology, we consider a sudoku puzzle. A sudoku puzzle is a 9 by 9 square grid where some of these squares contain numbers in the range [1..9] and the others are empty. The grid is subsequently divided into nine 3 by 3 non-overlapping boxes called regions. The objective is to assign to each of the empty squares a number in the same range as specified such that for any row, column or region, all of the numbers in it are unique.

					6		9	
						5		
2	5	4		1				
				4		8	2	9
	9			6				
		7	8					3
5					2	9	7	
8								
3		2			7		6	

Figure 1: A typical sudoku puzzle.

First we should decide upon the integer variables, whom are called **decision variables** or just variables for short. For a problem on this form there is an obvious choice, namely being that we have one variable for each of the squares in the grid.

These decision variables each have a set of integers which represent the possible values that variable may take. This set is known as the **domain** of the variable. In sudoku the values any square is allowed to assume is an integer between 1 and 9, and thus it is beneficial to set the domain of each variable to contain these and only these values. The exception is for the squares that already contain a pre-specified integer, and the domains for these should be set to contain only that certain number.

The restrictions are known as **constraints** in CP and they specify relations that must be upheld for a solution to be valid. A first constraint one might identify for the sudoku puzzle is that no square may contain numbers outside the range from 1 to 9 since that is part of the problem description, but we already handled that by setting the domains of each variable contain exactly those numbers. Further constraints are those that specify that for each row, column and region each square has a unique value.

A model that contains all these elements, i.e., the decision variables, their associated domains and a set of constraints is called a Constraint Satisfaction Problem (CSP).

2.2 Search and Propagation

Given a CSP we want the CP system to find one or more **solutions** to it, where a solution is a state where each variable is assigned one value and there are no conflicts in the constraints. This is done through alternate searching and propagation as described below.

Propagation is the technique of decreasing the domains of the variables by using information gained from the constraints of the problem. As in the sudoku puzzle we know for example that no number may figure more than once in a given row. Therefore, if one variable is assigned a value we can at least remove that particular value from the domains of all other variables in that same row. For columns and regions it is analogous.

When the domains of the variables can no longer be diminished in size through propagation alone and there are still unassigned variables left we must **search** in order to find a solution. When searching we make some choice of action that decreases the size of a variable's domain, usually through assigning a value to one of the variables. Subsequently the system propagates afterwards in case the particular change causes conflicts with values in the other variables' domains, and the cycle is repeated.

Do note that propagation may reduce the domain of some variable to be empty, if either the original model has no feasible solutions or the search engine made a bad choice at some point. In this case the search engine has to backtrack to its latest choice and make a new one. Worth noting also is that a variable that has but one value in its domain is effectively an assigned variable. An assigned variable may also be called fixed.

7	1,8	1,8	5	2	6	3	9	4
6,9	3,6	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	1,3,6	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5,8	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	1,2,4,6	7	8	5,9	1,5	1,4,6	4,5	3
5	1,4,6	1,6	1,6	3,8	2	9	7	1,8
8	1,4,6,7	1,6,9	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	1,4	2	1,9	5,8,9	7	1,4	6	1,5,8

Figure 2: Search commences.

As an example of the procedure of search and propagation, consider the sudoku puzzle depicted in Figure 2. Each square displays its current domain. The squares with only one value are assigned, and the others have had the value of each assigned square with which they are in conflict removed from their initial domain of the integers 1 to 9, through the propagation process. Since propagation can not diminish the domains of the unassigned variables any more, search commences. In this example, we select the top-most left-most square and assign it the lowest value, in this case 1, from its domain.

Thereafter we begin propagating by removing that particular value from each domain of every variable in the same row, column and region. This in turn leads to other variables becoming fixed, and the process continues as depicted in the series of pictures nominated

by a below, i.e., Figure 3a, Figure 4a and Figure 5a, where the squares that have been assigned values through propagation are marked with an asterisk. Eventually however our choice leads to assignment of values for variables such that they conflict, as in Figure 6a. At this point we backtrack to our latest choice point and select another value, which happens to be 8.

By the same process as previously we can start eliminating values from the unassigned variables, as in Figure 3b, Figure 4b and Figure 5b. Eventually, in this case, we will reach the solution for this sudoku Figure 6b), without having to resort to further searching or backtracking.

7	1	8*	5	2	6	3	9	4
6,9	3,6	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	3,6	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5,8	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	2,4,6	7	8	5,9	1,5	1,4,6	4,5	3
5	4,6	1,6	1,6	3,8	2	9	7	1,8
8	4,6,7	1,6,9	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	4*	2	1,9	5,8,9	7	1,4	6	1,5,8

(a) Propagation.

7	8	1*	5	2	6	3	9	4
6,9	3,6	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	1,3,6	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5,8	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	1,2,4,6	7	8	5,9	1,5	1,4,6	4,5	3
5	1,4,6	1,6	1,6	3,8	2	9	7	1,8
8	1,4,6,7	1,6,9	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	1,4	2	1,9	5,8,9	7	1,4	6	1,5,8

(b) Propagation.

Figure 3: First step of propagation.

7	1	8	5	2	6	3	9	4
6,9	3,6	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	3,6	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	2,6	7	8	5,9	1,5	1,4,6	4,5	3
5	6*	1,6	1,6	3,8	2	9	7	1,8
8	6,7	1,6,9	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	4	2	1,9	5,8,9	7	1*	6	1,5,8

(a) Further propagation.

7	8	1	5	2	6	3	9	4
6,9	3,6	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	1,3,6	3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	3,5,8	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	1,2,4,6	7	8	5,9	1,5	1,4,6	4,5	3
5	1,4,6	6*	1,6	3,8	2	9	7	1,8
8	1,4,6,7	6,9	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	1,4	2	1,9	5,8,9	7	1,4	6	1,5,8

(b) Further propagation.

Figure 4: Second step of propagation.

7	1	8	5	2	6	3	9	4
6,9	3*	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	3*	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5	1,2,7	6	1,3,5	4,7	4,5	1,5,7
1,4,6	2*	7	8	5,9	1,5	4,6	4,5	3
5	6	1*	1*	3,8	2	9	7	8*
8	7*	1,9	1,6,9	3,5,9	1,3,4,5	2,4	3,4,5	2,5
3	4	2	9*	5,8	7	1	6	5,8

(a) Even further propagation.

7	8	1	5	2	6	3	9	4
6,9	3,6	3,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	1,3,6	3,5	1,7	4	1,3,5	8	2	9
1,4	9	3,5,8	1,2,7	6	1,3,5	1,4,7	4,5	1,5,7
1,4,6	1,2,4,6	7	8	5,9	1,5	1,4,6	4,5	3
5	1,4	6	1*	3,8	2	9	7	1,8
8	1,4,7	9*	1,6,9	3,5,9	1,3,4,5	1,2,4	3,4,5	1,2,5
3	1,4	2	1,9	5,8,9	7	1,4	6	1,5,8

(b) Lot of other propagation.

Figure 5: Third step of propagation.

7	1	8	5	2	6	3	9	4
6,9	3	3,6,9	4	7	8	5	1	2,6
2	5	4	3	1	9	6,7	8	6,7
1,6	3	1,3,5,6	1,7	4	1,3,5	8	2	9
1,4	9	1,3,5	1,2,7	6	1,3,5	4,7	4,5	1,5,7
1,4,6	2	7	8	5,9	1,5	4,6	4,5	3
5	6	1	1	3,8	2	9	7	8
8	7	1,9	1,6,9	3,5,9	1,3,4,5	2,4	3,4,5	2,5
3	4	2	9	5,8	7	1	6	5,8

(a) Propagation reveals contradictory variable assignments.

7	8	1	5	2	6	3	9	4
9	6	3	4	7	8	5	1	2
2	5	4	3	1	9	7	8	6
6	3	5	7	4	1	8	2	9
4	9	8	2	6	3	1	5	7
1	2	7	8	9	5	6	4	3
5	4	6	1	3	2	9	7	8
8	7	9	6	5	4	2	3	1
3	1	2	9	8	7	4	6	5

(b) Eventually propagation ends up finding a solution.

Figure 6: Last step of propagation.

2.3 Gecode

Gecode [2] is a toolkit developed in and for the C++ programming language and contains all tools necessary for modeling, developing and running a CP system. This includes setting up variables and their domains, posting the relevant constraints and searching for any and all solutions to the model. Gecode is an open and also very structured project, so modifying and extending the existing code to fit the needs of one's own problem is both possible and fairly easy to do. This section does not go into detail of exactly what Gecode is capable of, nor how it does it. For a more detailed description on how to use Gecode, view the Gecode manual [12].

2.3.1 Constraints and Propagators

Constraints in a problem are realized through what is known as **propagators**. A propagator implements some relation between variables and, when invoked by the search engine, trims away values from the domains of the concerned variables that will not figure in any solution. A constraint for a problem may thus either be implemented by one single propagator, or as multiple different propagators that effectively do the same thing.

There are quite a few propagators already implemented in Gecode. Some of these build up the basic necessities for modeling the constraints of a problem, while others are developed for more intuitive use or optimization. For example, if modeling that a set

of variables must have unique values it is possible to constrain every pair of variables to be unequal to each other. However, this produces more and unreadable code, plus each of the propagators can not utilize information about the fact that all of the variables must be unique, since the propagators have no communication or information exchange between each other apart from the modification of the domains of shared variables and thus one propagator doing this would be beneficial.

It is also possible to simply construct propagators of one's own design to cater to relations specific to the problem currently at hand. Even though this was a viable option, we never needed to use this functionality since Gecode supplied a fair amount of scheduling specific propagators. Here we list a few of the propagators that come with Gecode which are available for our convenience. This list represents the complete list of propagators that we use for our model, not the full range of propagators available in Gecode nor all of their functionality [13].

- Rel(*expr*)
  Most boolean and equational expressions *expr* can be expressed with the Rel constraint in ordinary C++ syntax. For instance, restraining x, y and z to fulfill the linear equation $x + y = z$ may be posted simply as Rel($x + y = z$).

- Distinct(v)
  The Distinct propagator states that $\forall v_i, v_j \in v \ . \ i \neq j \rightarrow v_i \neq v_j$, i.e., that each variable value in the given array v is unique.

- Element(l, r, c)
  Given the variable array l and the two variables r and c, the propagator Element restricts $c = l_r$. That is, element number r in array l is equal to the value of c.

- Count(x, y)
  Count restricts the number of occurrences of values in variable arrays. It certifies that variable array x contains any value j at most y times, i.e., no value in x occurs more than y times.

- Unary(s, d)
  This is one of the constraints/propagators specially made for scheduling. Given an array s of starting times and another array d of event durations, the Unary constraint schedules starting times for all events such that no two events overlap. Mathematically, it is equivalent to $\forall i \neq j \ . \ s_i \notin [s_j, s_j + d_j - 1]$.

- Cumulatives(s, r, d)
  Similar to the Unary propagator, Cumulatives is meant to schedule events such that they do not overlap in time, but only if they share required resources. Here, s is the starting times of the events, d is their respective duration and r is their resource allocation. This constraint has multiple other parameters, such as how many activities each resource can handle, but none that we used. Cumulatives can be described as $\forall i \neq j \ . \ r_i = r_j \rightarrow s_i \notin [s_j, s_j + d_j - 1]$.

- Div(a, b, c)
  Enforces that $\frac{a}{c} = b$. Supposedly implemented in a more efficient way than Rel($\frac{a}{c} = b$) in how it propagates.

2.3.2 Search and Branching Order

Gecode comes with a built-in search engine [11], but that does not limit the choices during search at all. First of all there exists a couple of predefined branching orders, i.e., what variable to choose and what to do with its domain for the search. This may be useful for finding a solution faster by searching the search tree in a direction that is more likely to yield an answer, or maybe to quickly shut down whole sections of the search tree that will never render solutions.

For example it may be beneficial to choose the variable that currently has the smallest domain to limit the number of branches in the search tree early on, or perhaps the variable that figures in the most constraints and therefore hopefully trigger plenty of propagation that removes values as fast as possible.

As for what to do with the variable when chosen, the most common thing to do is to assign a value to it from its domain. Here too it may be useful to apply any known heuristics about the problem at hand, such as picking the smallest value available or other value choosing strategy.

Should none of the already existing variable and value orderings reflect a desirable search order one can build one's own branching order decision structure. This may be some static choice order decided upon prior to the search, or it may be dynamically altered by the results acquired from previous search and propagation.

3 Problem Description

The current scheduling process at TekNat can be read about in Section 3.1, with both the positive and negative aspects of it. What we hoped to achieve with this project can be read about in Sections 3.2 and 3.3. Our work plan can be read about in Section 3.4, and all research we built this strategy upon is found in Section 3.5.

3.1 Scheduling at TekNat

TekNat currently has a staff of six people manually scheduling the courses for each semester. Each semester is composed of two study periods and courses are, with few exceptions, allocated to one or two study periods within one semester. Each scheduler is responsible for scheduling the smaller, basic courses at his or her own department. For the larger and therefore more complicated courses, some coordination between the scheduler officers is required as they might involve students, teachers and staff from different departments. Furthermore, because there are so many students attending these courses they also have a more limited choice of locations. There are only two large lecture rooms on Ångströmslaboratoriet and Polacksbacken, with a capacity of 220 and 300 respectively, which makes them occupied for most of the period ($\sim$85%). The scheduling staff therefore gathers 5-6 times each period to plan for the larger courses.

The tool the scheduling officers use is called TimeEdit [17]. TimeEdit stores information about the reservations made from every department. This tool will highlight reservations that use conflicting resources such as teachers and rooms. It does not however offer any suggestion as for where a course is to be held in terms of time or place.

The scheduling officers at TekNat have a great amount of experience with scheduling courses at TekNat. They will therefore recognize that some courses are repetitions of courses in the past, and that the information of how those courses were held can be brought back and reused to save time when constructing the new schedule. The staff have over time refined their methodology to the point where it takes approximately five person-month work to build a schedule for one study period. In the IT department this means scheduling about 40 courses, and for the entire TekNat 200 courses. Because the problem instance at the IT department is much smaller than the whole TekNat instance is has been more convenient to run simulations on just the IT department. Before the scheduling can begin, the specification for each course has to be collected and therefore the teachers have to know the details of a course quite far in advance. This information is relayed to the responsible scheduler via email in a free-text fashion. Since the teachers report to the scheduling officers this way, the scheduling officers can review and compare the information to a previous template of the same course. Thereafter the schedulers can give feedback on potentially missed information to the teacher and proceed to the actual scheduling of the course. Upon reviewing the course details, the scheduler can also detect unreasonable preferences and remove or replace them with more feasible substitutes. The schedulers also know which constraints are important so that when conflicts occur they can prioritize and ignore less important preferences. Because conflicts always occur. At our department, the IT-department, which schedules about 40 courses each period, the finished schedule have never been free of collision between different student groups [18].

The downside of having this system is the large amount of time required to build a

schedule. Furthermore, on occasion, conflicts are not necessarily resolved. This leads to resources being double-booked and teachers not getting their preferences fulfilled.

After a schedule is complete, it is frequently updated as problems are often encountered during the period. Teachers might leave to go on a conference, groups may grow until the course does no longer fit in its designated room etc. On average, the schedulers have to make slightly more than one amendment per course, where each amendment ranges greatly in complexity, taking anywhere between 2 minutes to several hours spread out over days to correct [18].

A troublesome issue with the TekNat scheduling problem is the number of courses to be scheduled. The scheduling officers have considered reducing the complexity of it by introducing blocks. Each block would represent a slot where a number of non-conflicting courses are grouped, these blocks would then be repeated each week during the semester. However, due to the high amount of courses and the students participating in them, no such schedule has yet been constructed without it requiring that each student extended his workday to start in the early morning and last until midnight, every day of the week, in order to fit in all of the blocks. [18]

3.2 Primary Goals

Our goal was to design a program that could solve a scheduling problem the size of a TekNat instance over one night. TekNat hold about 40 courses in the IT-department and about 200 courses in total every study period, spread over five different campuses.

The schedule has to fulfill all of the hard constraints identified, see section 4.2.2. We should create a user interface wherein the schedulers and teachers can input the information desired by the program in order to start producing a schedule.

3.3 Secondary Goals

In addition to finding *a* solution, we would like to find the best solution; the solution that fulfills as many soft constraints as possible. As teachers have preferences in terms of where and when they want to hold their courses, we should satisfy as many as possible of these soft constraints according to some mechanism for weighting them.

The user interface should be intuitive and easy to use when managing the courses to be scheduled, starting the program and retrieving the solution. The interface usability should be on level with the scheduling officers' current tools. If the interface is too technical or otherwise too difficult to use, the program may not be used by teachers and thus they would not realize some of the preferences or constraints they can put on a course.

3.4 Approach

In order to achieve the goals mentioned above, we decided to:

- construct a webpage that teachers can access to edit their courses.
- make the execution of the program as easy as possible through a single command.

- present our output in a standard calendar format: iCalender [14].

Having a webpage as an interface allows great accessibility. It is easy for the scheduling officers to distribute a link to the teachers who can then quickly input their courses without any form of installation required. Or, if this is still too troublesome, the teachers can still reply in their free-text fashion emails to the schedulers who can then input their course to the webpage based on their preferences.

The program itself should be intuitive. There should not be any tricky flags or option the user could get stuck with when executing the program. All information needed to run the program should be included in the instance data taken from the web site. The program should then run until told otherwise. If a solution is found it should immediately begin searching for a better solution without delay or user input.

The iCalendar format [14] is a good choice to present our output in since it can be read by a large number of calendar programs. It is an open standard format for keeping a calendar and it is one of two formats that TekNat students use, presently, when downloading their schedule from TimeEdit.

3.5 Prior Research

Most of the reports on the subject of scheduling with CP that we came across have been for weekly, repetitive schedules [9, 3, 1].

The courses at TekNat are however not designed like that, but rather set for an entire period or semester at a time. A lot of these papers also discuss the method of *local search* [9, 5, 19], which is a feature that our CP library of choice Gecode [12] does not support.

Prior to this project, a feasibility study of CP as a means to schedule courses for TekNat had been made by Johan Ludde Lundin [6]. The results of the study were positive and a preliminary model design was made in *MiniZinc* [7]. MiniZinc is a high-level CP language with intuitive syntax for quick and simple usage. However, due to its high abstraction level the Gecode model derived from it is not very efficient, and from there comes the interest in developing in a more efficient, low-level programming language, and Gecode fits the bill.

3.6 Development Model

We have had our eyes on achieving the goal laid out and have to some extent neglected to document the process of getting there.

Our developmental model has been an iterative extension of functionality with a priority on maintaining a working product.

At first we produced a simple constraint model that could only perform the most basic timetabling problems. We then continued to extend this model with a number of implementations of basic constraints until we were confident we had an overview on the different parts we had and what parts we would like to incorporate into the model. At that point we rewrote the entire program to make it more modular and also split it up into what we call the intermediate model (Section 4.1), and the CP model (Section 4.2). When that was finished, we spent a large amount of time refining search procedures and

replacing each constraint with faster equivalents, or less memory consuming equivalents, or both. Each design decision has been motivated as beneficial with regards to the runtime (and in some cases memory usage). While we will not demonstrate them all, we will give a few examples of different improvements made in Section 5.3.

In parallel to the evolution of the program, we also developed increasingly sophisticated test data in order to test new constraints and their effectiveness.

4 Model

Our model is code-wise split in two, one that maintains relevant data structures for the courses and their events such as the number of participants, general descriptions of special constraints and so forth, and another that is the actual CP model. The former (hereby referred to as the *intermediate model*) is meant to, in an object oriented fashion, contain an easy-to-use configuration of course schedule specifications as an intermediate step between input data and the *CP model*.

The details of the intermediate and CP model can be read about in Sections 4.1 and 4.2 respectively. The search and branching heuristics are covered in Section 4.3. The different ways to use the scheduler can be found in Section 4.4.

4.1 The Intermediate Model

The intermediate model is the pre-processed storage of course related information, such as the number of events in each course, the participants for each event and so forth. This model handles the import of instance data, and the output of solutions. When importing, the intermediate model makes every piece of information easily accessible for the CP model. And when the CP model is done, the intermediate model steps in and maps the solution of different integer indexes back to the courses and print them out in a meaningful way.

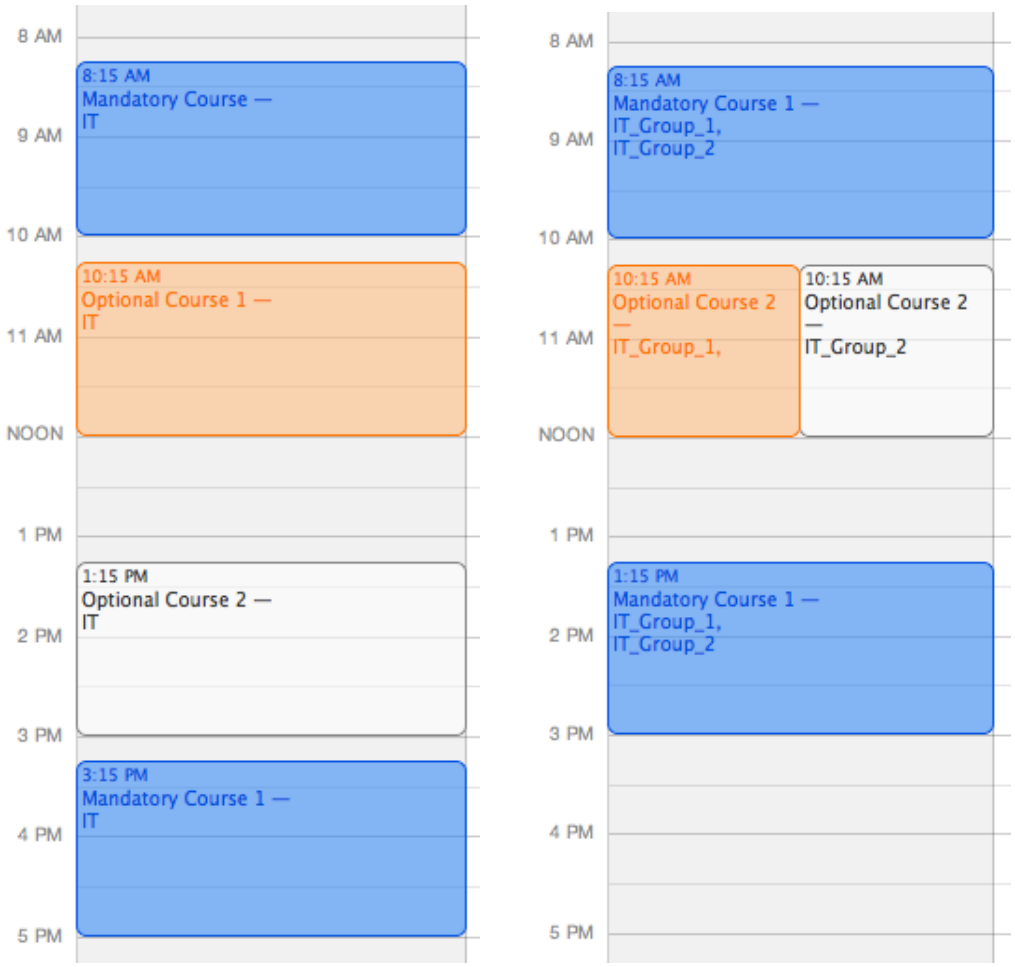
4.1.1 Terminology

When speaking of our model and implementation, there are a few words that sound rather general but in fact have a specific meaning.

Attendant groups & size Teachers and students of a specific group are called attendant groups. A teacher constitutes such a group by himself while attending groups made up of students usually involve several of them. Examples of these groups are IT3 and DV2. No attendant groups are allowed to have more than one event booked in a single time slot.

It is sometimes beneficial to split each attendant group down in smaller, more specialized groups. Using these specialized groups has the advantage that a subset of a group will not lock the same time slot for every other member of the group when participating in an event.

In Figure 7 this concept in visualized with an example. Here the attendant group IT has three courses, one mandatory and two optional ones. Even though the two optional courses are both a part of the set of optional courses belonging to IT, students very rarely choose to study them both, thus they are allowed to collide with each other. In Figure 7a the attendant groups are set up in an intuitive but sub-optimal way whereas in Figure 7b the IT-group has been specialized so that the optional courses can now be placed in the same time slot. The advantage of this restructuring of groups is more freedom in placing events, thereby increasing the chances of finding a solution in the entire search space.



(a) Using one attendant group.

(b) Using specialized attendant groups.

Figure 7: Advantages of using specialized attendant groups.

Size on the other hand is not at all connected to the different attending groups. Size simply denotes the number of different individuals expected to attend an event. This information is used to make sure a large enough room is chosen.

Event name The name of an event is used to separate two consecutive events in the same course from each other. If two events have different names, such as "Lecture 3" and "Lecture 4", they cannot be placed placed in the same time slot; "Lecture 3" has to be placed at an earlier time than "Lecture 4". However, when two consecutive events have the same name, the events are interpreted as being one lecture, lesson or lab session that is to be held in split class, and the two events are allowed to collide in time. This is useful when setting up lab sessions since the teacher can set up two or more events in the same time-frame using several rooms, each of which would have been too small to fit all of the students in.

4.1.2 Courses and Events

A course is a container for a list of the events associated with that particular course, plus additional information such as the name of the course, the total amount of events it

represents and such. The Course class also maintains the list of all previously specified courses.

Events are inserted into a given course in the exact same order they are meant to be scheduled. An event comprises a number of attributes, among which are the global index (known as id), name, duration and size, and also two lists of attendant groups and event-specific constraints respectively. The attendant groups are represented as integer indexes, and the Attendant class maintains a bijective mapping between strings (the name of the attendant groups) and a positive integer index.

Event-specific constraints are restrictions that a teacher puts on the events of her course. The class Econstraint handles a set of possible basic requirements a teacher might want to have. An event-specific constraint falls first into a category depending on what other variables and values it relates to. Then there is a sub-category that explains how they relate. E.g., if there has to be at least a week between a lab session and a lecture, we could constrain the second event to appear at least a number of time slots before the first event. Thus the event-specific constraint itself is placed on the second event, and the additional information it needs is the id of the first event and the amount of time to be dispositioned between them.

4.1.3 Data-flow

Instance data to the problem is stored at two locations: in a database and in a text file. The prototype web page communicates directly with the database, and the program reads from the text file. The text file can be generated from the database.

The database is never directly involved with the scheduling program, only transitively through the text file. The information stored in the database is tuned to be used with ease in the user interface.

The text file however is made for the scheduling program. All information on the courses' specifications, such as the number of events, constraints and attendants are stored in an XML-inspired text file [15]. The information has been compacted and any redundant information found in the database is eliminated. Upon execution, the program will import this data, process it and output a series of ics files. One file for either each course, or for each group. The ics files can later be imported to most calendar programs available.

The text file containing the information is generated from the user interface; the webpage. Currently, each time a modification is done to any course, event or constraint via the webpage, the webpage server composes a new text file whose content reflects the current database.

The XML-styled format is meant for easy parsing by the program and simple generation by other means. During our development process several ways of generating files in this format for testing and benchmark purposes were developed, depending on what was being tested.

4.2 The CP Model

The real-life problem that is scheduling has to be modeled as a CSP before it can be processed by any CP system. This can be done in any number of ways. The approach we

went for was the one most recommended by Lundin [6], at least in the way of deciding on the variable design. As for the constraints, a considerable amount of modifications and testing has been made to improve performance and usability.

4.2.1 Variables

The information in a schedule is in its most basic form three pieces of data; what the event is, when it occurs and where. We approach the problem from the point of view of assigning, to every event, a time and a place. By having a mapping from every specific event to a sequence of natural numbers, starting at 0, we can create two variable arrays which correspond to the time and place at which the indexed event is to be held. These two arrays are referred to as the **time** and **place** variable arrays, respectively.

At TekNat, every event must start at either one of four given times in a day, namely 8.15, 10.15, 13.15 and 15.15. Respectively, they must end the same day at any of the given times 10.00, 12.00, 15.00 and 17.00. This effectively translates into four time slots during a day when an event may occur, of roughly two hours each. These time slots are thus mapped to consecutive positive integers, and represent only the times of the workdays. The domains of the variables in time are set to be the range 0..200, as is the equivalence of a ten week period with 4 time slots per workday.

The rooms in which the events may be held are just mappings from numbers to a list of rooms. They are ordered in increasing order according to their population capacity. This is beneficial for the constraint of making sure no events are held in rooms of inadequate volume.

Apart from these two fundamental variable constructs, we have a few other variable arrays that are derivates of time and place. These include **day**, **week**, **halfDay**, and **campus**. The first two are simply which day and week (respectively) that a given event is held. **halfDay** is the groupings of morning (8.15-12.00) and afternoon (13.15-17.00) time slots. The last, **campus**, is merely at which campus the event presides. All of these can be and are derived from either **time** or **place** and are not as such a necessity for our model, but are beneficial when constructing constraints.

4.2.2 Constraints

The problem formulation consists of a number of constraints where each one implements an important aspect of what makes a schedule valid. All constraints are hard, meaning every constraint has to be satisfied in a solution. These are constraints for evaluating the derived variables and also the constraints for valid TekNat schedules.

The different types of constraints have been identified through reviewing the feasibility study [6], interviews with schedulers, reviewing past specifications sent from teachers to schedulers as well the constraint breakdown by Thomás Müller [9] and Rhydian Lewis [5].

To envision how we formulated the constraints in the form of propagators below we use a pseudocode which is explained here:

- When posting a propagator PROP with arguments *args* it is written as PROP(*args*). The propagators used in the pseudocode can be read about in 2.3.1.
- Constant values and pre-determined arrays are written as A CONSTANT VALUE, with a self-explanatory name.

- Variables (both temporary and global) are written as *variable*. The global ones are *time*, *place*, *day*, *halfday*, *week* and *campus*, and are explained in 4.2.1.
- When referring to a cell in an array ARRAY (both variable and constant) with index i, it is written as ARRAY_i.
- Temporary arrays in GeCode are a bit special, as they can be constructed by inserting variables to the end of them and thus increase their lengths. This is helpful when wanting to post constraints on subsets of variable arrays. For temporary arrays A the sentence 'Add *variable* to the array A' means that *variable* is inserted at the end of A. Likewise, 'Clear A' resets the temporary variable to length zero. Even if an array is cleared, constraints posted prior to the reset is still remembered by the system.
- $v \leftarrow val$ assigns the value val to the variable v. **for** $v \leftarrow min$ **to** max initiates a loop where v is assigned every integer from min to max.
- $\triangleright$ indicates that the following text is a comment.

Linking Constraints There are several constraints whose whole purpose is to link the **time** and **place** to the derivate variables. They are done so as follows:

```
LINKING CONSTRAINTS
1  for i ← 0 to NUMBER OF ROOMS
2      do campuses_i ← the campus value of room i
3  for i ← 0 to NUMBER OF EVENTS
4      do DIV(day_i, time_i, 4)          ▷ 4 time slots in a day
5         DIV(week_i, day_i, 5)          ▷ 5 days in a week
6         DIV(halfDay_i, time_i, 2)      ▷ Morning and afternoon
7         ▷ Map room to correct campus
8         ELEMENT(campuses, place_i, campus_i)
```

Room Requirement The room requirements are split into two parts where one part is compulsory and one part is optional.

The compulsory part is that an event of a given size may not be scheduled in a room with a lower capacity than that size. Since this constraint must be held for any and all events, it was deemed favourable to index the rooms in sorted non-decreasing order according to their capacities. This effectively translates this particular constraint into finding the index of the first room with minimum acceptable storage capacity and removing all values below this.

ROOM REQUIREMENTS

```
1  ▷ ROOMS is the list of rooms. It is ordered such that
2  ▷        ∀i < j . capacity[ROOMS_i] ≤ capacity[ROOMS_j]
3  for i ← 0 to NUMBER OF EVENTS
4       do j ← 0
5          while capacity[ROOMS_j] < size[EVENTS_i]
6                do j ← j + 1
7          REL(place_i ≥ ROOMS_j)
```

The optional part states whether or not specific attributes related to the room must or must not be present. Examples of these attributes could be that a room must have a blackboard, a video-projector or that the room must be a computer hall. These are handled by event-specific constraints.

Room Collision No two events may be in the same room at the same time. The current model we use assumes that a given course is expected to be held at a specified set of campuses. This is beneficial for this particular constraint (if we assume that every course may not be held at very many campuses) since we need only constrain relations between the courses that share at least one possible campus.

However, in the case where many events span multiple campuses, they will all participate in multiple cumulative constraints (one for each campus). In the extreme case where all events can be held at all campuses this will result in a multitude of identical constraints. The constraint clones will only slow down the scheduling process without any contribution to the process at all. Therefore this implementation of the room-collision constraint will weaken as the percentage of campus-heavy events increases.

The alternative would be to use one global constraint where all events are told not to collide with any other event. While that implementation would also work it would also mean that a lot of time is spent on enforcing events that do not even share a campus to be in different rooms, which is redundant in every case apart from the extreme case above.

ROOM COLLISION

```
1  for each campus c
2       do Clear arrays tmpduration, tmptime and tmpplace
3          for each event e_i that can be held at campus c
4               do Add time_i to the array tmptime
5                  Add duration_i to the array tmpduration
6                  Add place_i to the array tmpplace
7          CUMULATIVES(tmptime, tmpplace, tmpduration)
```

- **Campus Distance**
  Due to the geographical distance between campuses it is not fair on neither the students nor teachers to force them to move between campuses in the 15 minute break between either the two time slots in the morning or those in the afternoon.

For everyone's convenience we decided that campuses may not be changed at all between these passes.

Our implementation of this constraint is to go through all pairwise match-ups of events that share attendant groups and set constraints depending on their relations. We make use of our decision that courses must specify what campus their events may be held at, and thus there are three discernible cases that each lead to different actions:

- *Case 1: The two events do not share potential campuses to be held at.*
  This effectively means these two events may never be held after each other on the same morning or afternoon.
- *Case 2: The two events share at least one possible campus, and at least one event may be held at multiple campuses.*
  If this occurs, we must ensure that either the two given events do not end up in the same half of a day or, if they do, that they are held at the same campus.
- *Case 3: The two events may be held at only one campus, which is the same for both.*
  If the only possible campus for two events to be held at is the same for both, there can never be any unreasonable travel time for students or teachers when moving between them, should they now end up close to each other in time.

Campus Distance

```
1  for each event e_i and e_j where i < j and e_i shares attendant groups with e_j
2      do if possibleCampus[e_i] ≠ possibleCampus[e_j]          ▷ Case 1
3          then Rel(halfDay_i ≠ halfDay_j)
4        elseif e_i or e_j has more than 1 possible campus      ▷ Case 2
5          then Rel(halfDay_i ≠ halfDay_j ∨ campus_i = campus_j)
           ▷ Case 3 posts no constraint
```

- **Ordered Events**
  Suppose a course has two lectures to be scheduled. The resulting schedule will effectively be the same even if the two lectures switch place with each other since when scheduled they are just allocated time slots and the lecturer will still present his material in the order he had already intended. A solver does not know this however and may thus find solutions it considers to be different but that are essentially the same.

  To avoid this predicament we ensure that events must come in the order they are entered into the intermediate model. The downside of this approach is that our scheduler is incapable of allowing even the slightest change in the schedule, i.e., even if the teacher in charge of a course does not care if he has two or three lectures before a lab session he must still decide upon this when designing his course plan.

Ordered Events

```
1  for each course c
2      do for each sequential event pair e_i, e_{i+1} in c
3             do Rel(time_i ≤ time_{i+1})
```

Note that we may not have a strict inequality in this case since we handle activities that are to be held in two or more classrooms (such as lab sessions and lectures) as separate, consecutive events.

- **Participant Collision**
  Two events sharing an attendant group can usually not overlap in time. The exceptions arise when one particular activity of a course is meant to be held in multiple rooms with split classes. Our model handles these anomalies by creating multiple events with the same name and treating them as separate events that still may be scheduled at the same time. In order to handle these irregularities we have separated the constraint of making attendant groups into two parts, one for the events without duplicate names and one for the remaining.

PARTICIPANT COLLISION (NON-DUPLICATE EVENTS)

```
1  for each attendant group a
2       do Clear arrays tmpduration and tmptime
3          for each event e_i that has attendant group a
4               do if e_{i+1} is not of the same name as e_i
5                     then Add time_i to the array tmptime
6                          Add duration_i to the array tmpduration
7          UNARY(tmptime, tmpduration)
```

PARTICIPANT COLLISION (DUPLICATE EVENTS)

```
1  for each events e_i, e_j where i ≠ j
       do ▷ For every e_i that is a duplicate event
2          if e_{i+1} is of the same course and category as e_i, and
3             e_i shares attendants with e_j, and
4             e_i is not of the same course and does not have the same name as e_j
              then ▷ The two events may not start before the previous ends.
5                  REL(time_i + duration_i ≤ time_j ∨
                       time_j + duration_j ≤ time_i)
```

- **Event Specific Constraints**
  Anything that goes under the jurisdiction of teacher preferences is dealt with as Event Specific Constraints. What this means is that requirements that are not general restrictions that can be handled by global constraints for all courses are handled here. It may be specific interspersing of time between events in a course and such. Essentially these constraints have to be handled separately and specifically, and since there can only be relations between one event (the one the given constraint has been posted for) and either another event, a certain day or other similar restrictions, they are posted as REL constraints.

- **Course Specific Constraints**
  Lastly we've also got constraints that are individual on a course level. There are two such constraints; the daily cap and the weekly cap. These constraints limit the

maximum number of events that are allowed to fit in a day and week respectively. If these constraints are not set it is probable that all of the events in a large course are all placed right after each other making the course very event-heavy in the earlier weeks.

4.3 The Search

The bottlenecks, as identified by the schedulers[18] of TekNat and in the feasibility study [6], are the courses of a large size and with many different attending groups. The search in our constraint program uses this fact and repetitively tries to schedule the events with the largest product of size and attending groups among the first unscheduled event in every course. The chosen event will be placed at the earliest possible time and in the smallest possible room (with regards to room requirements).

These steps are well motivated and deviations have shown to increase the time needed to find a solution in our tests (see Section 5.3).

Changing which event to schedule to anything that does not take into account both the number of participants and attendees in a balanced way will make them increasingly difficult the further down the search tree they get chosen. The problem is that those large events will either have no room to be in, or that there is always a collision with some other event for at least one of the participants. This pushes the event towards the end of the period until these big events just won't fit into the schedule at all.

If one tries to choose an event that has other unscheduled events prior to itself in the same course, it gets difficult to guess the time component since one has to be sure that the other events fit while at the same time making sure not to overestimate or else the schedule might not be compact enough for every course to fit in the period.

Trying to place the event as early as possible is a way to make sure that schedule becomes compact. Since the events in a course have to appear in chronological order, picking a later time will make it more difficult to fit the succeeding events in the same course.

Finally the event should not occupy a greater room resource than needed in order to leave as many options open for other events to be scheduled the same day. This is achieved by assigning the smallest available room that can still fit the number of participants of the event it is assigned to the biggest events first. Biggest in this case is the event with the greatest product of the number of participants and the number of different attendant groups.

4.4 Usability

The user's experience of the program is split in three: the user interface, the input needed to start the program, and the output produced. This section will cover each aspect in detail.

4.4.1 User Interface

Scheduling officers, as well as teachers, will set-up and manage the instance data through a webpage. A prototype webpage is written in html [16]/php [10]/javascript [4] and is

connected to a mysql [8] database. The webpage is available in both Swedish and English, and provides the means to create, edit and delete courses, events, constraints and groups. The user chooses a course to inspect and is thereafter presented with five areas: the navigation area, the course area, the template area, the event area and the constraint area.

In the navigation area the user can choose to inspect a course, to create and edit a course or to review what attendant groups there are and create/edit/delete those groups. Figure 8 illustrates this.

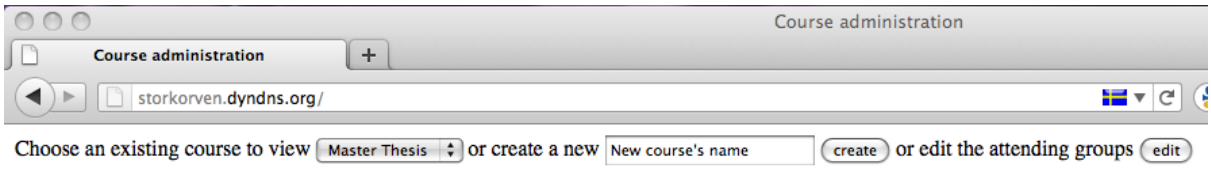


Figure 8: The navigation area.

In the course area, the user can edit information governing the entire course. These activities are setting constraints as for how many events can take place in a day or a week, changing the name of the course and get a summary of how many events are currently in the course. Figure 9 illustrates this.

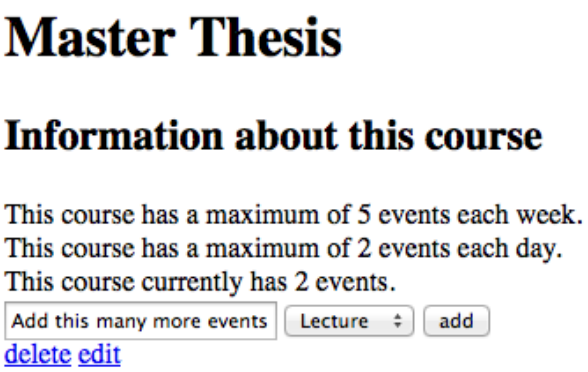


Figure 9: The course area.

In the template area, the user can store information that is later copied to each *new* event created. The template area therefore acts as a mould for all events that will be created in the future. The copied information includes start & end times, size, attending groups and constraints. Using the template is a handy way to quickly create a first draft of all the events in a course. Figure 10 illustrates this.

Event template

The information beneath is used as a template for new events.

The template does not have a standard amount of attendants.

The template does not have a defined timeslot to start in.

The template does not have a defined timeslot to end in.

edit

These constraints apply to the template:

ROOM_HAS

Has: blackboard
delete

SPECIFIC_CAMPUS

Campus: Polacksbacken
delete

Figure 10: The template area.

In the event area the user can further tune the details regarding individual events. Since not every event in a course needs to be of the same size nor have the same attending groups etc, we have provided tools to tweak these events on a one by one basis. Figure 11 illustrates this.

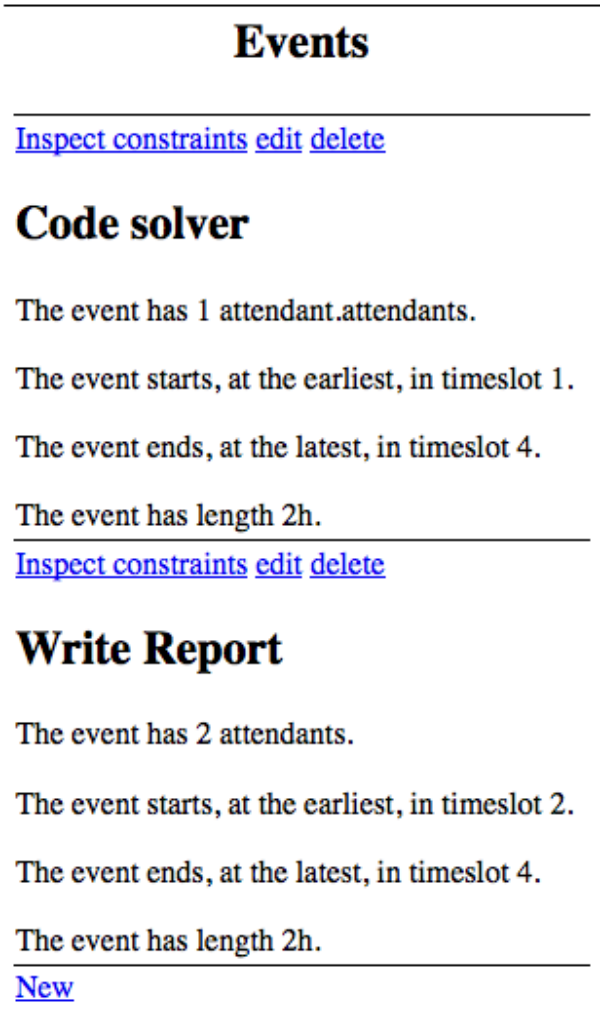


Figure 11: The event area.

The last area, the constraint area, is only visible if the user has chosen to inspect / edit an event. Here the user can further restrain the time frame or location of the event by adding any number of constraints from a list of 10 available constraints. The event these constraints are relating to is highlighted with a green background in the lists of events. Figure 12 illustrates this.

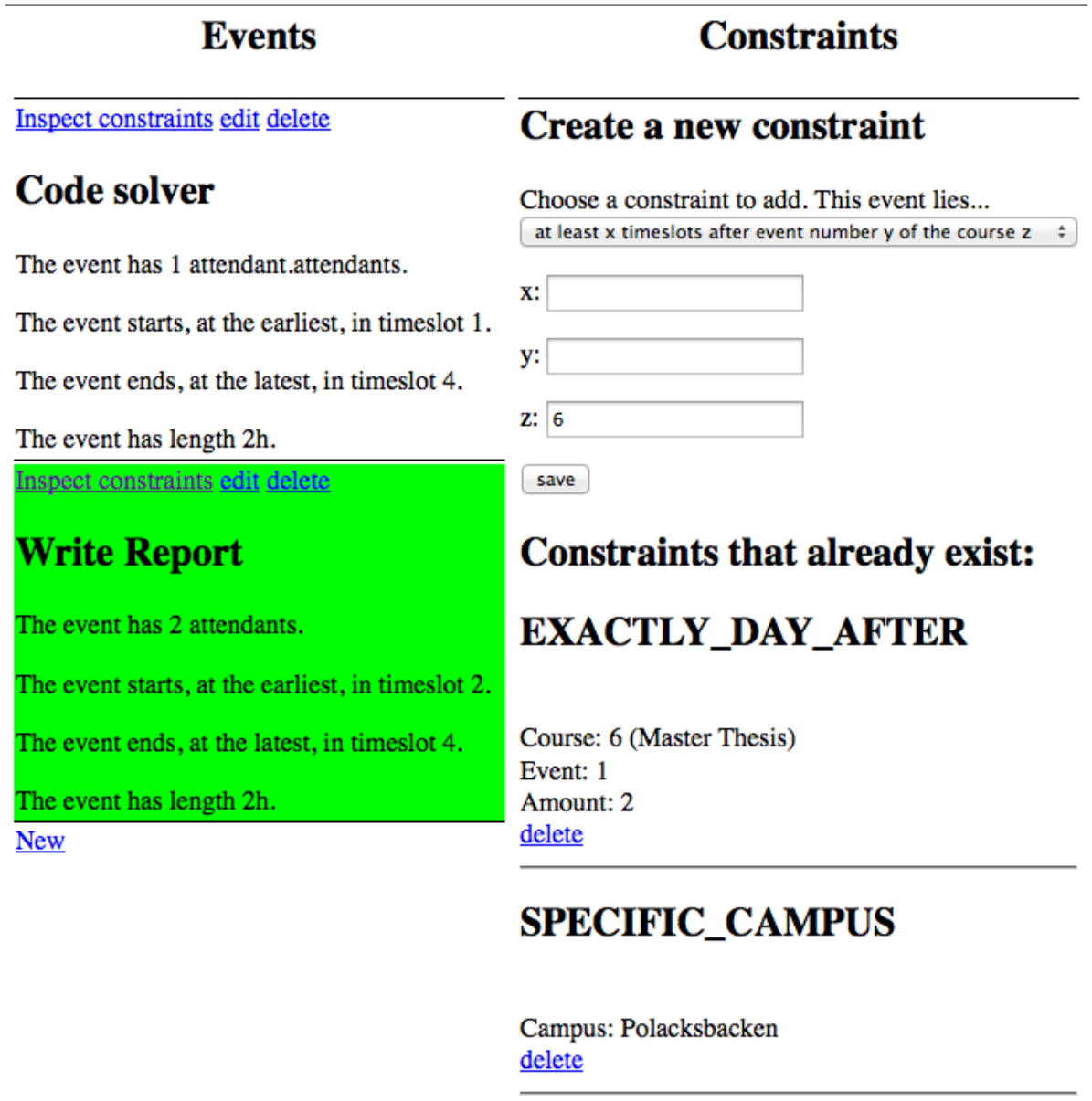


Figure 12: The constraint area.

4.4.2 Input

When changes to the instance data have been conveyed through the webpage, a function that maps the database to the program readable text file is activated. Therefore, running the program is a rather simple task. All a user has to do is to run a command in a terminal to start the scheduling program. No flags are needed nor allowed. The program

then exits upon termination.

4.4.3 Output

After the program has finished searching for a solution, some meta-data relating to Gecode is presented in the terminal. This data contains information that is mainly irrelevant to the user, such as the depth of the search tree and how many propagations were perfumed, but also interesting bits of data such as the runtime and whether or not a solution was found.

If a solution has been found, the finished schedules are written in iCalender [14] format to a number of files. The events in the files produced can be grouped by either course or by attending group. These files can later be distributed to teachers and students or further processed (such as adding extra information to events) with any other iCalendar supporting software.

5 Results

The program can solve some estimations of the typical problem instance for the IT-department within a minute. The entire TekNat is about 200 courses big, while the subset of just the IT-department contains about 40 courses. In this section we will explain our reasoning on estimating the instance data as well as covering different performance measures of our current program, both in its current state and in past states.

During the iterative development process we designed test data to test the specific functionality implemented and find limitations to the current model. Some of the earlier data were merely helpful to see whether our model did what was expected of it, and also to give us time consumption estimates of additions to our model. These mostly consisted of courses with lots of unconstrained lectures of two hours and randomly inserted lab sessions of four hours in two rooms, and the results of these test on half-finished models holds no interest for comparison with our later data.

It is almost always preferable to evaluate a product by testing it on real life problems and data. Course specifications by the teachers are unfortunately not stored in one single computer parsable format in a database somewhere, but written in free-form text for the human schedulers to read, interpret and schedule. Therefore the evaluation of our scheduler falls upon how well it handles generated data, and how realistic that data is.

How the courses are built up can be read about in Section 5.1. What variables and parameters that can differentiate and what values we assigned to them can be found in Section 5.2. Finally, the end results are presented as graphs in Section 5.3.

5.1 Course Patterns

Even though it would be difficult to acquire all of the course specifications we still inquired among teachers and schedulers about specifications of previous courses they've held. Among these we selected a few that had a fair amount of variation and deviations in structure and preferences from each other and made into mould patterns for our generation processes, as described below.

- *Lecture-only pattern*
  This format was based on lecture-heavy specifications. A specification could essentially be a request for a given number of lectures to be scheduled during the given period. Some teachers expect the schedulers to remember certain personal and sometimes course-related preferences, if the course has been held previous years.

  We assumed that any teacher would like at least some spread of the lecture placements so we made constraints that different events may figure at most two times per day and nine times per week. If we do not make this assumption, a course that is scheduled early and is on this format would most likely have sequential events on each time slot every day for the first few weeks due to our search order, which is seldom found in real life schedules.

- *Block pattern*
  This format describes clustering of events in block-like structures. The preference of this layout was that each block should have some spacing between them, so the last event of one block is not "too close" to the first event in the succeeding block.

The generated blocks will contain about 4 to 6 events. Every 5th to 8th event is a 4 hour laboratory session in split rooms. Since there was not a given amount of time the blocks needed to be separated by we set it to be at least 2 days. Furthermore we assumed that some spread of the events was desired here as well so we made an additional constraint that there may be at most 2 different events per day.

- *Weekly pattern*
  This format is based on specifications that state the exact weekdays the events should be held. These were usually repeating on a weekly basis, so we made clusters of 7 to 9 events and assigned a weekday to each of them. The last event of each of these clusters is a 4h laboratory session in split rooms, and the penultimate event is a 2h lesson in split rooms.

  The assignment of days is handled by setting how many lectures each of the five given workdays may contain, with a limit of at most 3 events. Additionally, the starting day of the first event is randomly assigned to a weekday so as to decrease the chances of every lab session session being on Thursdays or Fridays.

  The very rigid structure of this design would make it possible for it to, when many courses are generated, construct data that does not yield a solution, e.g., if a lot of courses has events to be held only on Mondays and Thursdays. In order to prevent this we eased the restrictions of a course having to be on a specific day to being on a specified day or the day after that.

5.2 Test Data Generation

Apart from the structure of the courses there are quite a lot of different parameters and constants to manipulate. We do not try out all of the combinations of these because of various impracticalities of it, like time consumption, presentation and analysis. Instead we test values which hopefully represent real data or merely test the limits of our software. The constants that are up for modification are listed below, as well as our reasoning behind whether we test different values for them or not, and why we pick exactly those fixed values.

According to the standard set by TekNat, all courses are either 15-, 10- or 5-point courses, hereafter sometimes referred to as big, medium and small courses, respectively. For simplicity during rendering of test data we assume that each course is held during one period only and does not stretch over an entire semester.

- **Number of courses.**
  We have a pretty good estimate of the number of courses held each semester at TekNat, which could essentially be used when making our test data. However, if we do not manage to solve an instance of that particular size, how big a problem *can* we solve? Also it is interesting to see how well our model scales with changes to this parameter.

  For the distribution between big, medium and small courses we assume that for every two big courses there are three medium and three small ones, and scale thusly.

- **Number of events per course.**
  Just like for the course patterns we can look at previous courses and see how many events they consist of and make estimates of average values for the different sized courses. The current values are set to 23, 38 and 61 activities for the small, medium and big courses respectively. Every lab session and lesson generated that is to be held in half-class generates two events in the model instead of one.

- **Class size.**
  How many people actually take a course varies quite a lot, from single digit to over a hundred. Despite our efforts in contacting schedulers and other related officials at TekNat the exact limits are to us or them not known, nor is the actual distribution. For simplicity we exclude a few values from the extremes and go for a simple, uniform random distribution in the range 30 to 70. This unfortunately relieves us from some of the greater bottlenecks the current schedulers have, being that the very large courses has very few possible lecture halls big enough for the entire class, but this approach was chosen for simple and parametrised generation of test data and would be too specific should we endeavour to incorporate sporadic exceptions and irregularities.

- **Attendant groups.**
  The number of attendant groups are slightly difficult to predict or calculate as it is an abstract concept. What may be interesting is to see how big an impact it has to modify this value; how much faster or slower will the program become with changes to the attendant groups?

  When generating data we make it so that neither students nor teachers should be involved in more courses than one when the course is big, since one big course per period is evaluated as a full-time job. Therefore big courses just get one attendant group that is distinct from all other attendant groups in the entire period.

- **Campuses.**
  Most courses hardly ever switch campuses between events, even though some may do so in special circumstances and exceptions. For each course, we randomly select one of the three available campuses to be held at, with a 1/8 chance to also be allowed to be held at one of the other two campuses.

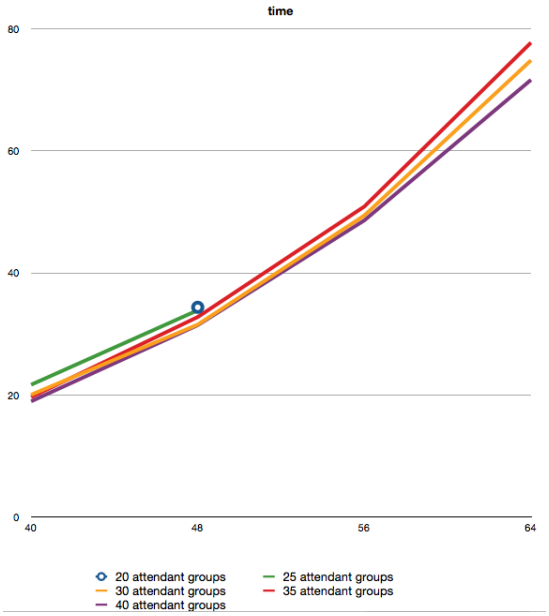
- **Distribution of course patterns.**
  With no real grasp of the multitude of course design patterns, since we could only acquire a small subset of all the course specifications from various teachers and the scheduling officers, we simply assume that our specifications are a fair representation of the average designs, and that they are uniformly distributed among the courses.

5.3 Graphs

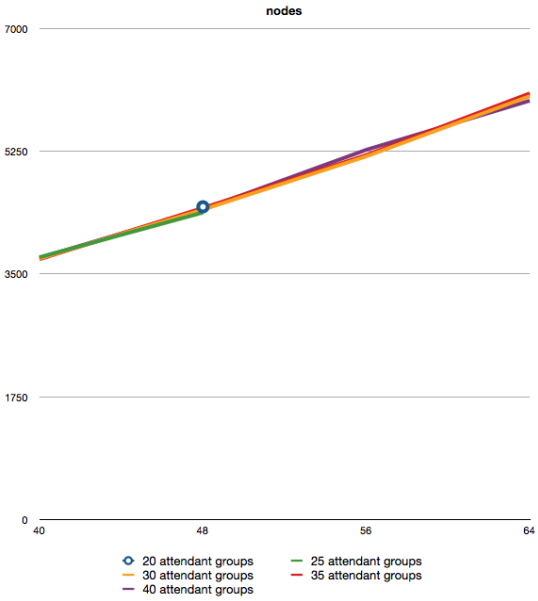
We will here present a number of graphs visualizing different performance metrics relating to our program. In every graph, each line represents a specific number of different attendant groups. Furthermore, each line represents not only one instance but an average of the successful instances for that type of data set. The reason for this is that each

failed instance has very extreme measurements in every dimension and would pollute the graphs to the point where they did would not convey any information of our program's performance. However, we have included graphs showing the success rate for different combinations of data sets The success rate can be used as a pointer to how reliable the program is for different amounts of workload.

Performance on the Department of Information Technology The following measurements were taken on data simulating the amount of courses given by the IT-department and to be scheduled in a single campus, namely Polacksbacken. Figure 13 and 14shows the time and different time derivations as well as the maximum memory usage for different instances of the problem.



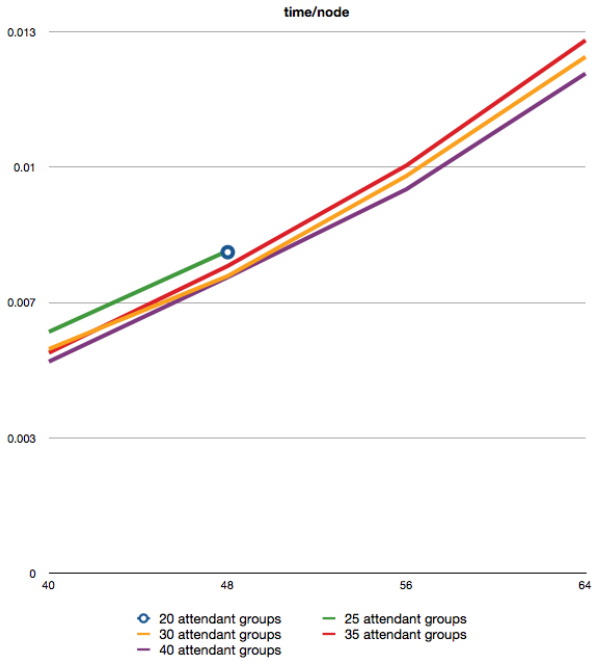
(a) Time consumption for various sizes of attendant groups.



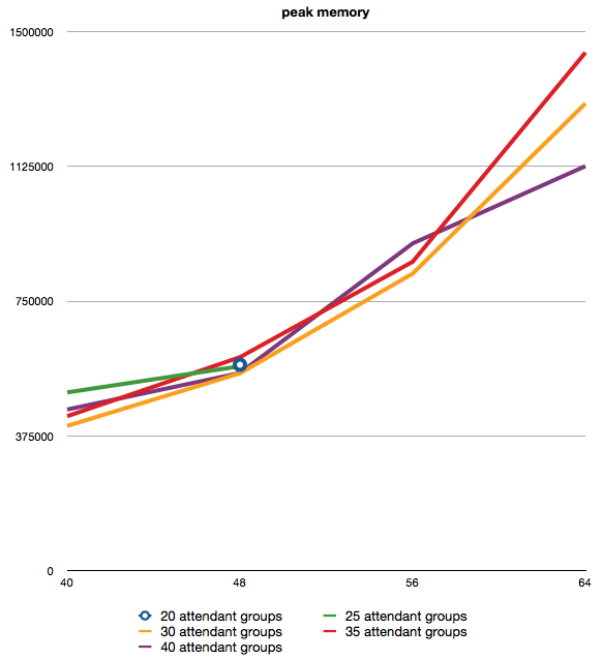
(b) Amount of nodes for various sizes of attendant groups.

Figure 13: Different performance metrics on data sets resembling that of the IT-department.

Figure 15 shows the amount of propagators and their propagations for different instances of the problem. Figure 16 and 16 also includes graphs depicting the number of failures and the success rate for different amounts of workload.

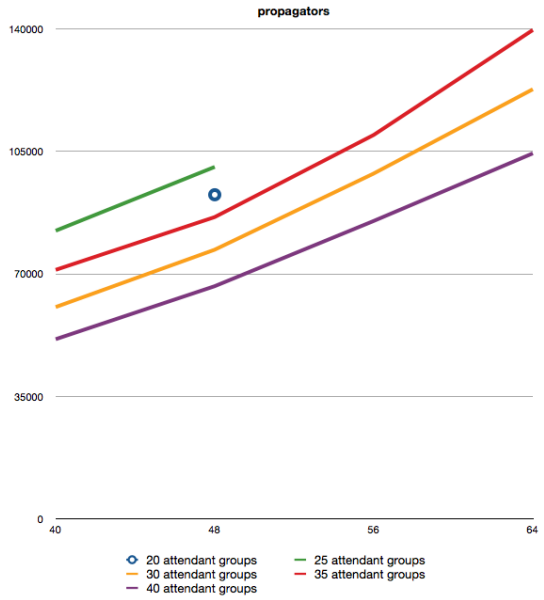


(a) Time consumption per node for various sizes of attendant groups.

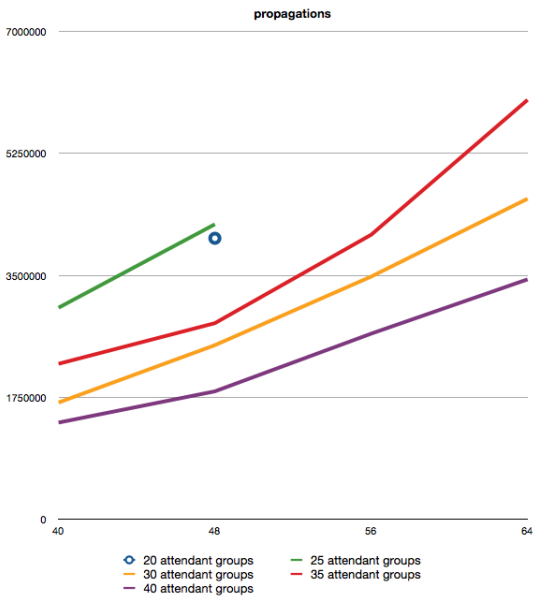


(b) Maximum memory consumption for various sizes of attendant groups.

Figure 14: Different performance metrics on data sets resembling that of the IT-department.

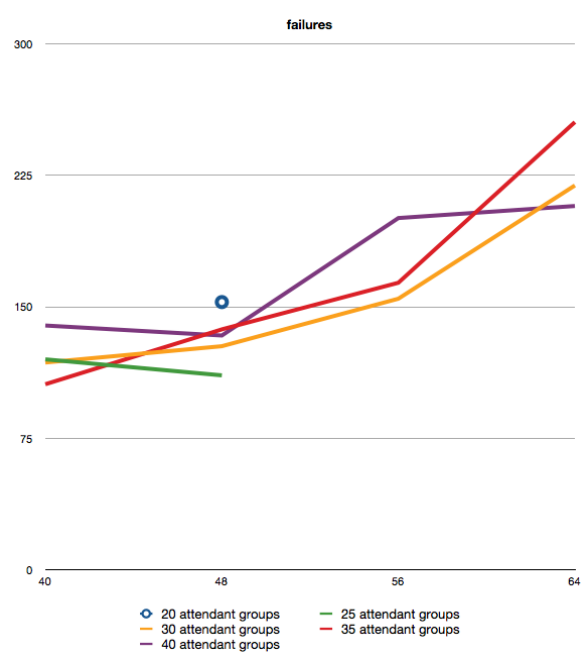


(a) Amount of propagators for various sizes of attendant groups.

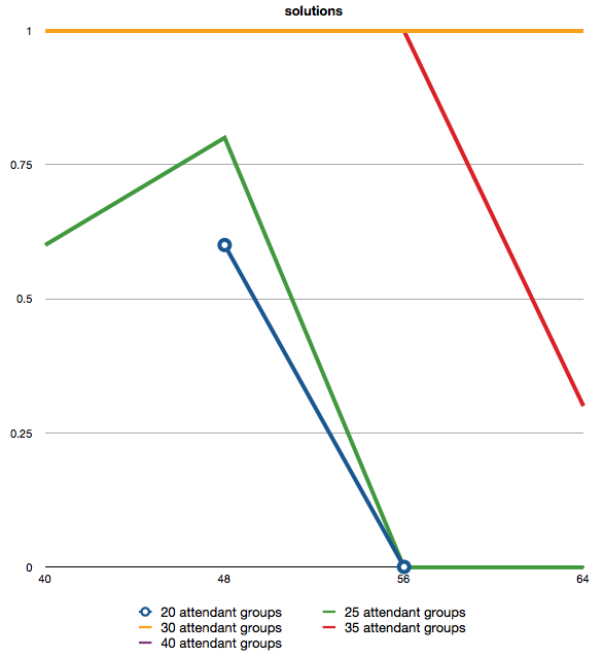


(b) Amount of propagations for various sizes of attendant groups.

Figure 15: More performance metrics on data sets resembling that of the IT-department.



(a) Amount of failures for various sizes of attendant groups.



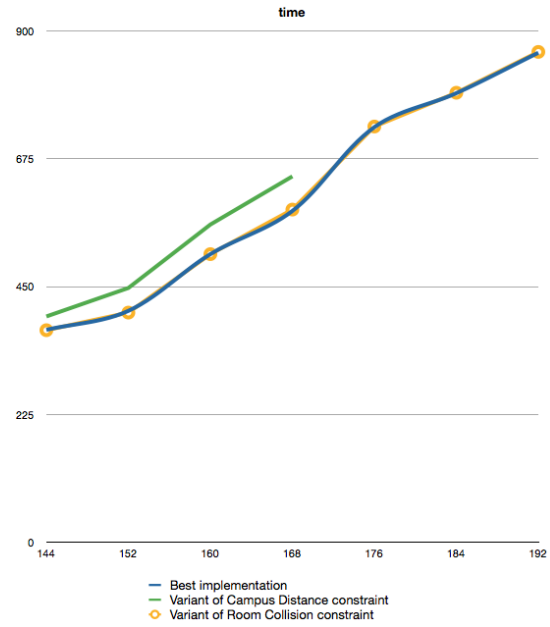
(b) The success rate for various sizes of attendant groups.

Figure 16: More performance metrics on data sets resembling that of the IT-department.

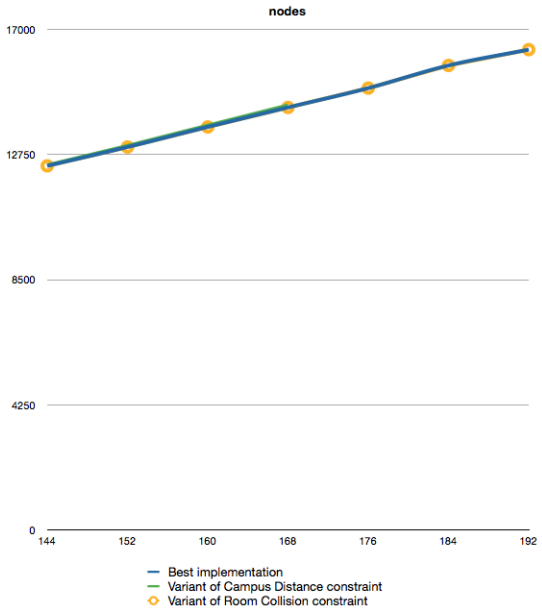
These results are very positive in that a solution is found within a minute, which is above our expectations. While the success-rate may look grim at a first glance, one should keep in mind that these graphs go beyond the scope of 40 courses and further on until the program could no longer easily solve the problem.

Performance on TekNat As in the previously, the following measurements were taken on data simulating the amount of courses given by the TekNat and to be scheduled at three campuses. Every test involves 200 attendant groups. Figure 17 and 18shows the time and different time derivations as well as the maximum memory usage for different instances of the problem.

Figure 19 shows the amount of propagators and their propagations for different instances and different implementations. Figure 20 and 20 also includes graphs depicting the number of failures and the success rate for the same tests.

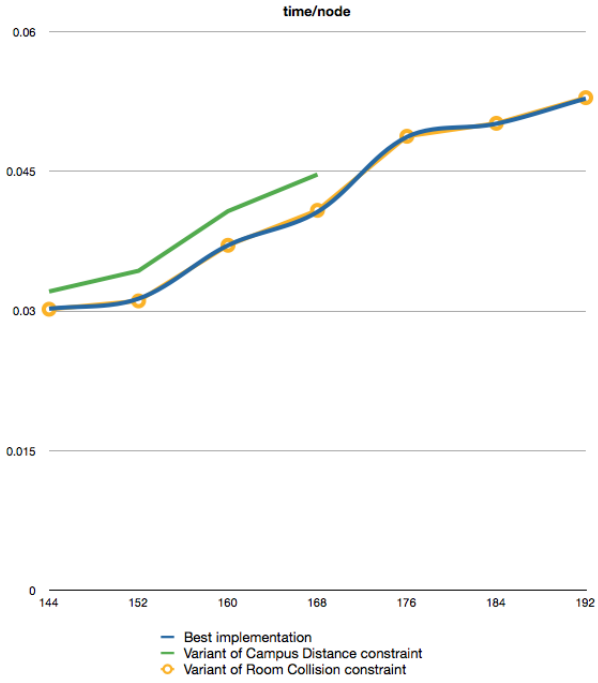


(a) Time consumption for various implementations and problem sizes.

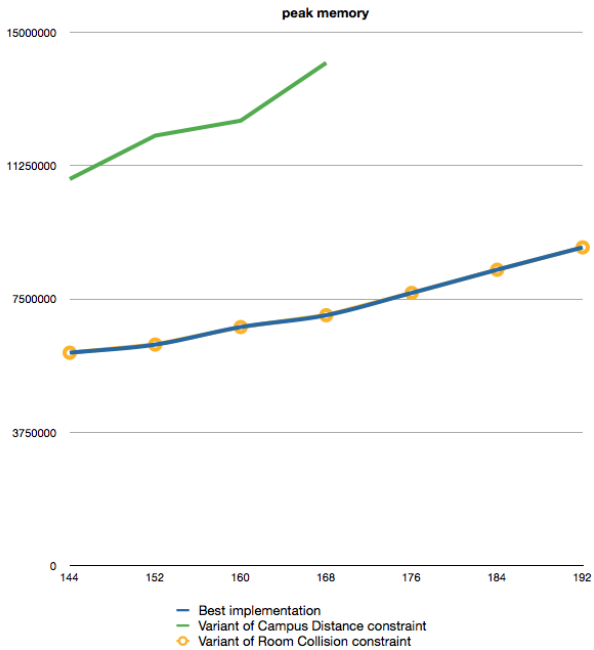


(b) Amount of nodes for various implementations and problem sizes.

Figure 17: Different performance metrics on data sets resembling that of generic TekNat scheduling instances.

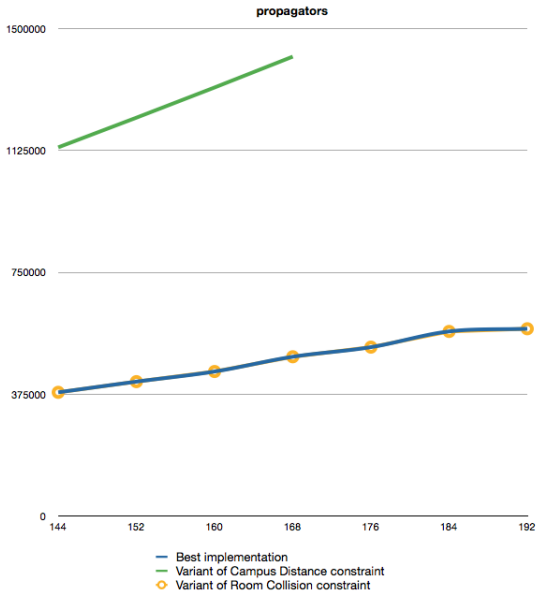


(a) Time consumption per node for various implementations and problem sizes.

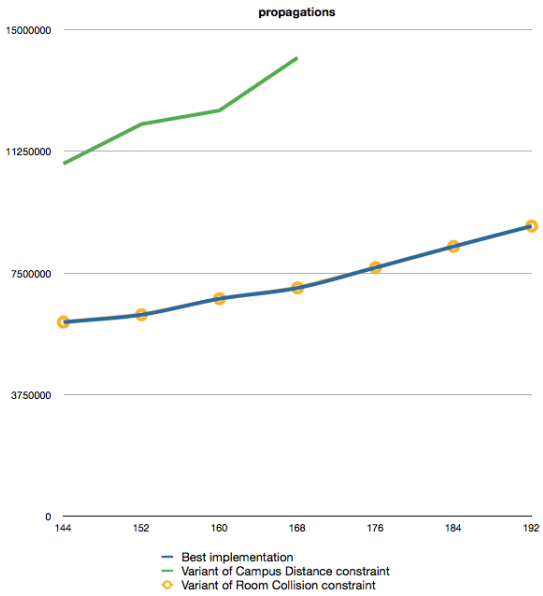


(b) Maximum memory consumption for various implementations and problem sizes.

Figure 18: Different performance metrics on data sets resembling that of generic TekNat scheduling instances.

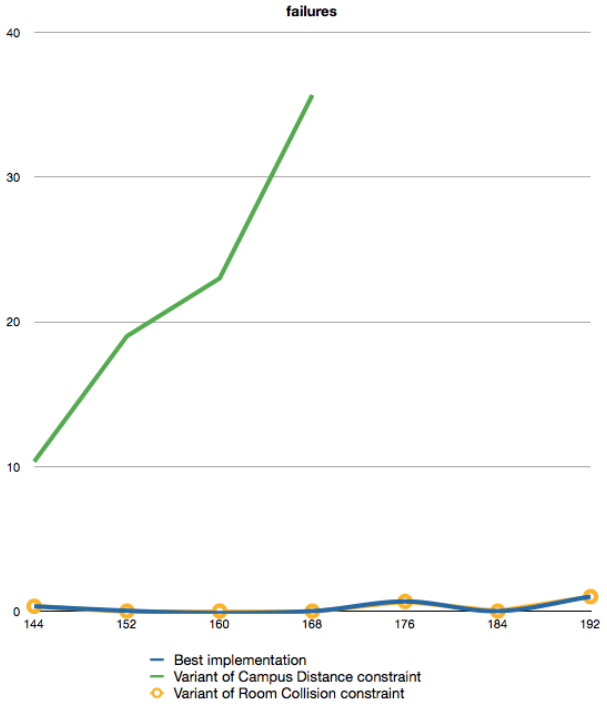


(a) Amount of propagators for various implementations and problem sizes.

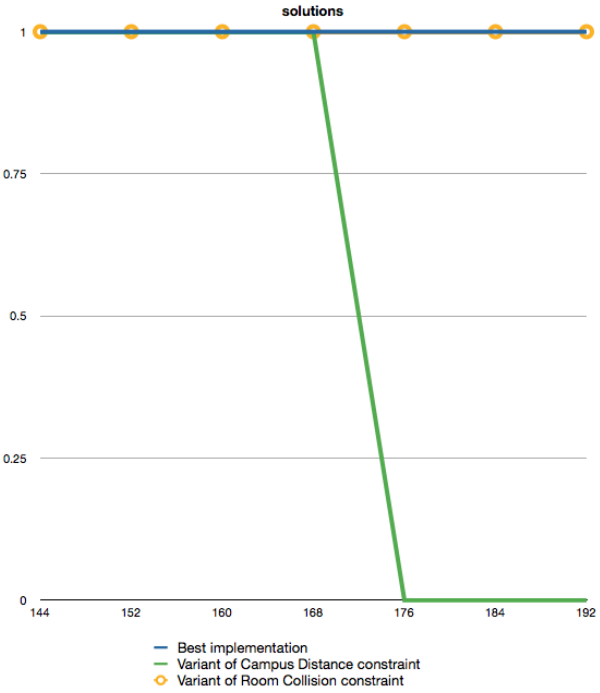


(b) Amount of propagations for various implementations and problem sizes.

Figure 19: More performance metrics on data sets resembling that of generic TekNat scheduling instances.



(a) Amount of failures for various implementations and problem sizes.



(b) The success rate for various implementations and problem sizes.

Figure 20: More performance metrics on data sets resembling that of generic TekNat scheduling instances.

Our goal was to make a program that can schedule the entire TekNat courses. For now our program can handle most such instances. For these data sets there were only successful instances in our most recent implementation. We have however encountered instances that could not be solved, but no such instance appeared when generating the data final data sets for these benchmarks.

When the test runs did fail, it was due to memory exhaustion on our server within 15 minutes. This will not be a real problem however, since a lot of the courses are actually held on one specific campus for that course, which means that one can essentially break down the problem and schedule each course on its specific campus, and subsequently schedule the courses that has to be held at different campuses.

Current program versus old versions The main model has been the same throughout the development period in that we have one variable for the time and room of each event. What changes are the implementations of the constraints. We have tried to improve the performance of our model in both time and memory consumption.

These improvements have also been made with respect to solving entire TekNat instances. In hindsight not all of these changes may be helpful at all when running campus-specific data.

The graphs 17, 18, 19 and 20 detail current and old implementations of the Room Collision constraint 4.2.2 and the Campus Distance Constraint 4.2.2. As can be seen in the graphs, changes to the the Room Collision implementation had a negligent impact on every metric while the modifications made to the Campus Distance implementation made for a drastic overall improvement. When trying out different variations of a constraint, the payoff could differ substantially.

6 Discussion

There are both advantages and disadvantages of using this program as opposed to the current hand-made schedules. While the limitations section may seem dominating at first, the benefits are, in our opinion, of greater magnitude.

6.1 Benefits

When the instance data is similar to our tests, a solution is found very quickly. While it usually takes one person-month to schedule one period of 40 courses (such as the IT-department of TekNat from start to finish, our program completes the puzzle in a minute. For the entire TekNat instance it takes a few minutes, work that currently takes about 5 person-months. There is no need to supervise the machine during this time, it is done fully automatically.

When the instance is solved, it is guaranteed that there are no collisions in the resulting schedule.

If successful the staff can then further improve the quality of the schedule by modifying the result by hand, or simply use it as is and focus on other tasks.

However, if the program is unsuccessful in finding a solution, the scheduling officers can still make use of the specifications collected through the webpage instead of collecting

them over email. Furthermore, practically no time is lost in trying to use the program when it is unsuccessful, all a user has to do is to start the program, and then check back after a few hours to see if a solution has been found or not. Furthermore, if the program is unsuccessful in its first attempt the scheduler can easily revise the input data by going through courses one by one and alter constraints known to be difficult to realize. Afterwards, the scheduler can start the program once more, with a greater chance of success. Therefore the only obstacle in using the program as an aid would therefore be the user interface which is still a prototype and could use further polishing, but it is still functional in its current condition.

6.2 Limitations

There are several aspects of using this model for solving this particular scheduling problem that will impact whether or not solutions are found, their quality, and the scope of the model in general.

6.2.1 Bad Branches

A drawback of using this kind of systematic CP search technique is that if we make a bad decision at some point during the search, a decision that later has to be reversed, is that a lot of time might be consumed before the erroneous decision is located since we might have to examine the entire subtree that branch leads us to.

In the TekNat scheduling problem, with our own generated data, this behavior will manifest itself if there are too few teachers or if student groups are attending too many courses, in which case there might not be a solution at all. If student groups are booked to participate in too many courses, the hard constraint of no participant overlapping will delay events until a time-slot is found where all the participants are free. In this scenario the search will practically only find invalid leaves because the time frame of the period will not be enough. Thus, almost every branching decision has to be backtracked and a lot of time spent exploring bad branches.

Eventually, of course, the search will result in either finding a solution or invalidating the entire search tree. Because the typical problem instance is so large, choosing a bad branch will in reality mean that the program will run for an indefinite amount of time before reaching a conclusion.

6.2.2 All or Nothing

As soon as the search engine realizes it is on a bad branch, backtracking will occur. In other words: as soon as one constraint cannot be satisfied, backtracking occurs. Therefore if a problem instance does not have any solution, we can not retrieve a “good try” where the least amount of constraints were broken. That information would have been useful as a starting point for further manipulation by hand or local search, to achieve a valid or better schedule.

6.2.3 Schedule Quality

There are many parameters for how to measure the quality of a schedule. We could for example try to minimize the amount of free periods for every class, decrease the distance students have to walk between events or how many early mornings teachers have to hold lectures. It becomes very subjective as to what a good schedule actually is.

Gecode offers tools for finding the best solution by using an objective function (or cost function), and a technique known as branch and bound. The branch and bound assumes that a desired solution will be of at least a certain quality (evaluated by the cost function), and excludes branches of the search tree that could never yield such solutions. Should a solution of better quality than what was desired be found, the search engine looks only for solutions better than the last one found.

The one problem for this approach is that in order to exclude branches of the search tree prematurely the cost function must, in some way, be evaluatable in non-leaf nodes. If the value of the cost function can not be evaluated except for in the utmost nodes of the tree then finding the best solution is reduced to finding all solutions and comparing them. A cost function that evaluates the number of free periods for the attendant groups will effectively be reduced to this.

Even so, the schedules produced should be of fairly good quality. Events are placed at their earliest convenience during the search, hopefully making the schedule as compact as possible, at least for the big courses.

6.2.4 Robustness

A minor inconvenience with the current search is that we always put events in the smallest possible room. However, the number of students planning to attend a course is not necessarily the number of students that signed up for it. Often, students will want to join in even though they were not signed up in advance. On these occasions, the room chosen for the event might be too small to support the late-comers while there might still be rooms that are unoccupied at the same campus, at the same time slot. It would therefore be slightly beneficial to perform a post-search manipulation of the solution and upgrade the rooms whenever possible.

6.3 Inflexible Event Orders

For both technical and practical reasons, our program does not allow for changes in the order of the events of each given course. Even if a teacher allows for some interchangeability between different events, e.g. if it does not matter if there are two or three lectures preceding a lesson, the program does not allow for it.

7 Future Work

While we feel we have put a lot of work into automating the scheduling process for TekNat, we have encountered several features that could not be implemented within the time frame of our project. Future continuation of this project could pick up either of the tasks outlined here to greatly improve the model as well as the user experience.

7.1 Constraint Implementation

Is Section 4.2.2 we mentioned that we do have several implementations of the same constraint to choose from, and that the performance of each depends on the problem instance. Instead of picking the version that has the best performance in general, one could analyze the instance data and then choose what implementation to use in order to get the best performance.

7.2 User Interface

We recommend tuning the user interface to be more intuitive and helpful. While it currently implements the functionality required to construct the courses to be scheduled, it is not as easy to use nor as graphical as desired. The web site is at the moment a sophisticated interface to a database and that view should be changed to better fit the tasks of a teacher or scheduler.

Some of these desired features would be:

- Drag-and-drop events to redefine their order. Currently the order field of each event has to be specifically edited when reordering several events.
- A 2-d grid of weekdays and time slots where a user could more easily define when, during any week, an event can be scheduled.
- When editing any part of a course or event or when inspecting constraints of an event the web site redirects the user to a new page. The view in the new page is not centered at the task that the user chose. It would be beneficial if instead the task would "slide" into view without leaving the current page. That would make the webpage easier to work with and reduce the time it takes to complete these tasks.

7.3 Starting Point

In the current implementation the program will start from scratch when trying to solve the scheduling problem. If the program could also be extended to read iCalender files for courses that have already been scheduled, either by the our program or in some other way, there is a potential to save time when scheduling as well as easing the tasks of making amendments to a schedule. The schedulers could then simply add the events or constraints needed to fulfill the new course-specification, add the other, already scheduled courses to the mix and rerun the program to extract the new schedule for the modified course while keeping the other courses intact.

Because the text file that the intermediate model imports all the instance data from is rather similar to the iCalender files produced as output, this functionality is indeed not too far away. This would also mean that the program is more integrated with any other scheduling software capable of importing and exporting iCalender files.

7.4 Cost Function

More thought could go into designing a cost function. While we could not find a way to assert that the branches made would lead to the highest quality schedule, according to certain parameters, it might still be possible to define such an evaluator. If indeed such a cost function is defined the program could just keep running while it attempts to find better solutions until a set time has passed or the schedule is needed.

A successful definition would enable the shifting of teacher preferences from hard constraints to soft constraints. This shift would greatly impact on how usable the program would be as it would always find some sort of solution according to the now reduced amount of hard constraints, while still potentially finding even better schedules according to the soft constraints.

7.5 Local Search

Local search is a search technique that might have proven better for this problem. A systematic search with zero-tolerance for broken constraints, such as ours, will end up invalidating the entire search tree and not present any attempt at a solution if the instance data is not solvable, even though there exist solutions where only one constraint is broken. If local search were used it could be used to step in and take command over the search once the systematic approach fails. The benefit of this would be that local search will never deny the user some form of output, even if the solution is faulty. Since the problem instance of TekNat may very well be unsolvable in many cases local search may be a good option.

It would also be a good idea to combine a tabu search (an enhanced local search method) with the existing constraint search as well as a cost function. The local search could then operate on the solution of the constraint search in order to produce a high quality schedule in a relatively short amount of time as shown by George M. White and Junhan Zhang [19].

7.6 Iterating to Solution

If a problem instance turns out to be unsolvable one could remove a few constraints/courses and try the search again. This pruning could be repeated until a solution is found. For this to work the schedulers have to define the importance of the constraints/courses so that the least important ones are removed in every iteration. Once a solution has been found the remaining courses could be filled in by using either a local search method, or by hand when the collisions could be chosen to take place at the least inconvenient time.

7.7 Handling Failures

Presently, when a search fails no information is relayed back to the user as to why. A continuation of this project should also involve an analyzer that understands why the search failed. The analyzer should also recommend actions leading to a successful search.

For example, when an event receives a domain-wipout the output could be:

```
The search failed because the course ‘‘Data-mining’’, event ‘Lecture 10’’ could
not be placed in this period.
The constraints that pruned the most of this event’s time-domain are, in descending
order:  only on half day 1, not on weekday 4, not on weekday 3, not on weekday
0.  You should consider loosening some of these constraints and then run the
scheduler again.
```

This information, or something similar, could be forwarded to the user as soon as the scheduler fails in one node or after a certain time, or when the search space is fully explored. In the latter case the problem could lie in several constraint or events and consequently the analyzer should deem how many of them are appropriate to present as problem areas to the user.

7.7.1 Exams

Our program does not schedule examinations since they behave differently than teaching events. With examination events, one would like to minimize the number of exams a student has to take in a week. So instead of a compact schedule, a loose one is preferred. The amount of examination officers and the use of examination halls should also be minimized. Furthermore, there can be several exams at a time in an examination hall, and a teacher could hold several exams at once. Although scheduling exams were not in the scope of this project, it would be convenient if those too were taken care of in the same program.

8 Division of Labor

While much of our work was done in unison, we've also been working on different areas individually. Henning Hellkvist engaged in the development of the interface and the underlying databases while William Sjöstedt focused on generating data for testing purposes as well as running benchmarks. In the intermediate model Henning handled the input coding, while William took care of the output. The CP-model however, is entirely a joint effort.

9 Acknowledgesments

During our development we have met several people who have helped us in different ways. We would like to thank the following people for their contributions to our program and report; Pierre Flener for his assistance in constructing the model and providing feedback on this report, Ulrika Jaresund for providing background information and insight to the methods used by the scheduling officers, Johan Ludde Lundin for proving us with the

model he derived during the feasibility study, Farshid Hassani Bijarbooneh for helping us with technical troubleshooting on the application server.

References

[1] S. Deris, S. Omatu, H. Ohta, and P. Saad. Incorporating constraint propagation in genetic algorithm for university timetable planning. *Engineering Applications of Artificial Intelligence*, 12(3), 1999.

[2] Gecode. http://www.gecode.org.

[3] H.-J. Goltz and D. Matzke. University timetabling using constraint logic programming. In *Practical Aspects of Declarative Languages*. Springer Berlin / Heidelberg, 1998.

[4] Ecmascript language specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[5] R. Lewis. A survey of metaheuristic-based techniques for university timetabling problems. *OR Spectrum*, 30(1), 2008.

[6] J. L. Lundin. Discussion and walkthrough of feasability study 2011-01-26.

[7] Minizinc. http://www.g12.cs.mu.oz.au/minizinc/.

[8] Mysql 5.6 reference manual. http://dev.mysql.com/doc/refman/5.6/en/introduction.html.

[9] T. Müller. Constraint-based timetabling. http://muller.unitime.org/phd-thesis.pdf, 2005.

[10] Php manual. http://www.php.net/manual/en/.

[11] C. Schulte. Programming branchers. In C. Schulte, G. Tack, and M. Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2011. Corresponds to Gecode 3.6.0.

[12] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and programming with gecode. http://www.gecode.org/doc-latest/MPG.pdf.

[13] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling. In C. Schulte, G. Tack, and M. Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2011. Corresponds to Gecode 3.6.0.

[14] The Internet Engineering Taskforce. Internet calendaring and scheduling core object specification (icalendar). http://tools.ietf.org/html/rfc5545 2011-09-22.

[15] The World Wide Web Consortium. Extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/REC-xml/.

[16] The World Wide Web Consortium. Hypertext markup language 4.01 specification. http://www.w3.org/TR/1999/REC-html401-19991224/.

[17] Timeedit. http://www.evolvera.se/se/produkter.

[18] Ulrika jaresund, scheduling officer at teknat. Interview 2011-09-06.

[19] G. M. White and J. Zhang. Generating complete university timetables by combining tabu search with constraint logic. In *Practical Aspects of Declarative Languages*. Springer Berlin / Heidelberg, 1998.