University Course Timetabling
Solver Evolution

Tomáš Müller

Abstract This paper describes a number of improvements that have been made to the course timetabling solver used in the open source university timetabling system UniTime since our last paper on this topic (Rudová et al, 2011). This progress is demonstrated on benchmark data sets from Purdue University that were introduced in the earlier paper and that are available online.

Keywords Course timetabling · Local search · UniTime

1 Introduction

University course timetabling is an important and well studied educational timetabling problem (McCollum, 2007). The open source system UniTime[1] is one of the most advanced and publicly accessible timetabling systems available. It has been used at Purdue University for course timetabling since 2005.

In Rudová et al (2011), we have published an extensive study of the course timetabling problem at Purdue University. The paper included two large and quite complex benchmark data sets (from Fall 2007 and Spring 2007, 8 departmental problems and up to 2,500 classes each) on which our algorithm was demonstrated. While the two data sets were made available for other researches to use[2], to our knowledge, there have been no other papers published using these data, or others of comparable complexity, for comparison of results.

In this paper, we would like to revisit these two data sets to demonstrate how incremental advances in solution algorithms and improvements in constraint handling have been made that can be applied to improve solutions to

Tomáš Müller
Student System Competency Center, Purdue University, USA
E-mail: muller@unitime.org

[1] `http://www.unitime.org`
[2] `http://www.unitime.org/uct_datasets.php`

real-world course timetabling problems. The fact that it has been possible to both broaden the applicability of the solver to meet the needs of more institutions and improve the results on complex benchmark data sets are of note. Both improving results and broadening the applicability of solution methods are key to making automated timetabling methods practical.

The ability of UniTime to model most real-world constraints, combined with the ability to insert different solver techniques into the solver engine can make it a valuable platform for comparing a variety of solution methods on problems of realistic size and complexity. This can help in finding methods that are most successful in meeting the needs of university timetablers in actual practice. It is our hope that other researchers will have an interest in making use of the problem data sets we have made available and that there will be an interest in expanding the number and variety of data sets as more institutions worldwide make use of UniTime. They can provide a valuable addition to the course timetabling benchmarks from the International Timetabling Competition (McCollum et al, 2010; Bonutti et al, 2012; McCollum et al, 2012).

The paper is organized as follows: a short description of the course timetabling problem is presented in the next section (Section 2), this is followed by a brief description of the algorithm (Section 3), the two benchmark data sets are presented (Section 4) and the improvements made over the last decade are demonstrated (Section 5), the most important changes are discussed (Section 6) and followed by conclusions (Section 7).

2 Problem Description

Course timetabling is a resource allocation problem with the aim of assigning classes to times and spaces in such a way that no two classes are placed in the same room or taught by the same instructor at any given time. Additional hard distribution constraints must be respected. In the following experiments, we are concerned with four objectives: time preferences, room preferences, distribution preferences, and student conflicts. Time and room preferences measure how well particular desires for times and rooms are satisfied, distribution preferences measure how well the soft distribution constraints are satisfied, and student conflicts measure the degree to which student conflicts are minimized. Each objective is expressed as a number (e.g., a total time penalty or a number of students conflicts) and the solver is minimizing a weighted sum of these objectives. For more details about the problem, including an XML data format, please see `http://www.unitime.org/uct_datasets.php`.

2.1 Time

While the same schedule is typically repeated every week during the term, there can be exceptions. For example, some classes may only meet bi-weekly or only during the first N weeks of the term. Other classes may meet multiple

times a week, usually following a strict pattern. For example, a class meeting three times a week for 50 minutes always meets on Monday - Wednesday - Friday, with all meetings starting at one of a given set of times.

To allow for all these cases, our model of the academic term is divided into weeks, each week is split into seven days (Monday - Sunday), and each day of the week has 288 time slots, starting at midnight, with 5 minute increments. Each possible time placement of a class consists of a date pattern, one or more days of the week, a start slot and a length. The date pattern is a bit string defining the dates of the term on which a class can meet. Typical examples of a date pattern are: full term (a class meets every week of the term), odd/even weeks (a class meets every other week), or week $N - M$ (a class meets during weeks $N, N+1, \ldots, M$). All meetings of a class, which are based on the combination of the date pattern and the selected days of the week, start at the same time, have the same length, and are placed in the same room or rooms.

Figure 1 shows a sample date pattern. A class following this pattern meets during odd weeks. Classes do not meet on vacations and during holidays (Labor day on September 5, Fall break on October 10-11 and Thanksgiving on November 23-27). Moreover, weeks between Fall break and Thanksgiving are shifted by two days (a week starts on Wednesday). A class that meets on Mondays and Wednesdays during odd weeks would meet on those Mondays and Wednesdays that are marked yellow.

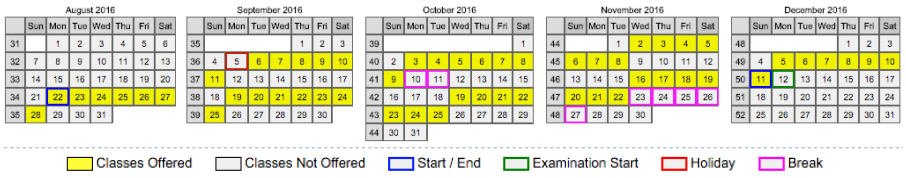


Fig. 1 Example *Odd Weeks* date pattern

Figure 2 shows an example time pattern. It defines two weekly meetings of 50 minutes each. The time pattern allows for the two meetings to be on Mondays-Wednesdays, Mondays-Fridays, Tuesdays-Thursdays, or on Wednesdays-Fridays, starting at 7:30 am, 8:30 am, ... 4:30 pm. Time preferences can be used to prohibit or require certain combinations (hard constraint), or to prefer or discouraged them (soft constraint).

Two classes overlap in time if and only if the selected date patterns, days of the week, and meeting times are overlapping. This means that there is at least one week, one day of the week, and one time slot in common. Based on these date and time patterns and the preferences given for each class, a class has a list of possible time placements together with penalties for assigning the particular time placement to the class. This means that times that are marked as prohibited are excluded from the list[3] and soft time constraints

[3] Required preference is a shortcut for prohibiting all times except those that are marked as required.

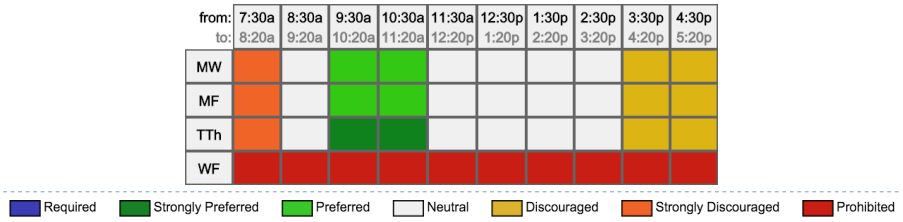


Fig. 2 Example 2 x 50 time pattern

(preferred/discouraged times) are used to compute penalizations for assigning a particular time placement to a class. In particular, there is a penalty of -4 for a strongly preferred time, -1 for a preferred time, +1 for a discouraged time, +4 for a strongly discouraged time, and zero penalty for a time with no preference (neutral). The time preference objective minimizes the overall time penalty.

For example, a class requiring 150 minutes of instruction per week can meet as follows:

- 3 times per week for 50 minutes each week, e.g., Monday - Wednesday - Friday between 8:30 am and 9:20 am
- 2 times per week for 75 minutes each week, e.g., Tuesday - Thursday between 9:00 am and 11:15 am
- 1 time per week for 300 minutes every other week, e.g., Monday between 11:30 am and 5:20 pm on Week 1, 3, 5, ...

Each time pattern also defines a break time (typically 10 or 15 minutes). This is the expected time for the students to be able to move from one class to the next. For example, a 50 minute class has typically a length of 12 slots (allocating the whole hour) with 10 minutes of break time at the end. This means that the students, the instructor, and the room are blocked during the whole hour, while the last 10 minutes can be used by the students and the instructor to get to some other room for another class.

2.2 Rooms

Each class may need a given number of rooms (typically one). A class may need zero rooms, in which case it is only to be placed in time. If a class requires two or more rooms, it is expected that the class meets in all its rooms during each meeting (all the assigned rooms are blocked by the class when the class takes place). Similar with times, each class has a list of rooms that the class can use together with their penalties, based on the size of the class as well as the requirements and preferences put on the class in the user interface. These room preferences use the same schema as time preferences and they can be set on buildings, room equipment/features, user defined room groups, or on individual rooms. For example, if a data projector is required, only rooms with

a data projector are on the list. Please note that a penalty for a particular room can be based on a combination of various soft constraints. For example, if a building A is preferred (penalty -1) and a room A 101 which is in the building A is strongly preferred (-4), the room penalty for the room A 101 is -5, while all the other rooms of building A have a penalty of -1.

A room may be available only during certain times or it may be allocated to a particular department at a certain time. This means that only classes of a certain department may be able to use a particular room during a given time. Also, it is not allowed for two classes to occupy the same room at the same time.

Figure 3 shows a room sharing matrix for a particular room. Each time can be allocated to a particular department (see Monday or Tuesday in the example), it can be marked as free for all departments that are allowed to use the room, or it can be marked as not available for course timetabling at all.

from: / to:	7:30a 8:00a	8:00a 8:30a	8:30a 9:00a	9:00a 9:30a	9:30a 10:00a	10:00a 10:30a	10:30a 11:00a	11:00a 11:30a	11:30a 12:00p	12:00p 12:30p	12:30p 1:00p	1:00p 1:30p	1:30p 2:00p	2:00p 2:30p	2:30p 3:00p	3:00p 3:30p	3:30p 4:00p	4:00p 4:30p	4:30p 5:00p	5:00p 5:30p	5:30p 6:00p	6:00p 6:30p
Mon	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101	0101
Tue	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100	0100
Wed																						
Thu																						
Fri	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

	Free For All
N/A	Not Available
0101	Student Instructional Planning
0100	Central Office (EXT: Large Lecture Room)

Fig. 3 Example room sharing matrix for a particular room

Distances between rooms are computed using the provided room coordinates or by a travel time matrix. There is a student speed constant which is included in the solver configuration (together with the weights of each of the individual objectives, etc.) that is used to convert distances in meters to travel times in minutes. Typically, we assume that during a 15 minute break time students can travel up to 1,000 meters.

2.3 Instructors

A class may have one or more instructors assigned. It is not allowed for an instructor to teach two or more classes at the same time. If an instructor teaches two consecutive classes, such an assignment is prohibited if the two classes are placed in rooms that are too far apart and there is a penalty when the two classes are close enough but still in a different building.

Typically, it is discouraged (penalty of +1) for an instructor to teach two consecutive classes that are not in the same location (distance is not zero) but are within 50 meters of each other. Distances over 50 meters are strongly discouraged (penalty of +4) and over 200 meters are prohibited.

2.4 Students

In our model, we work with individual student course demands. This means that each student has a list of courses he/she needs to attend. These course demands may be generated based on curricula (Müller and Rudová, 2014), taken from the previous year enrollments, student registrations, or a combination of these.

The problem is complicated by the fact that courses tend to have multiple classes. Courses with many students are usually split into several seminar groups and/or lectures. Furthermore, a course can be offered in various configurations (e.g., a lecture only, a lecture, and a lab), with multiple lectures and labs available and some restrictions on what combinations of lecture and lab students are allowed to take.

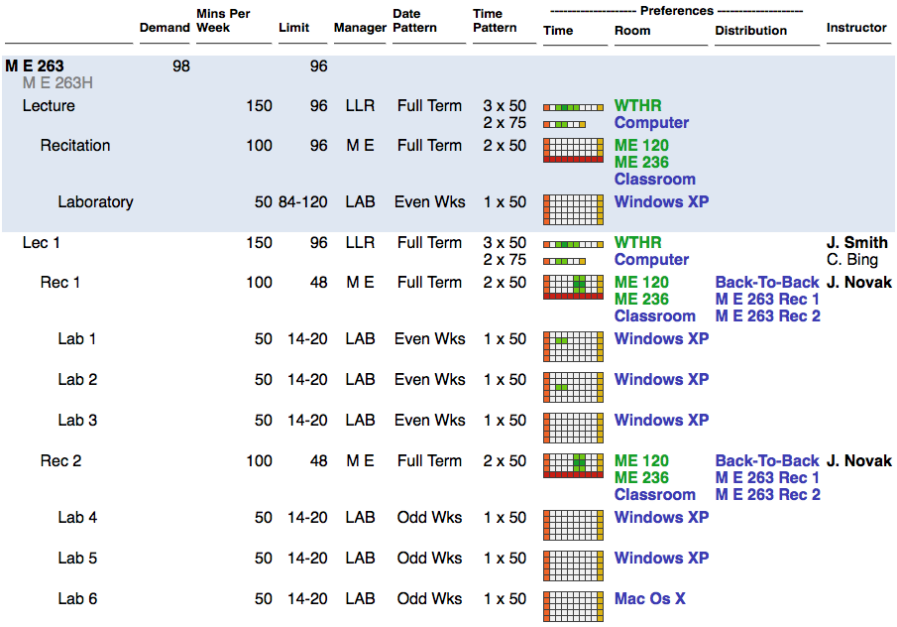
Figure 4 shows an example course structure. It shows an *instructional offering* that can be offered under two *course names* (M E 263 and M E 263H) and one *configuration* with a lecture, a recitation, and a laboratory *subpart*. A student needs to take one class of each subpart from one of the available configurations. The nesting of the three subparts (Lec 1 is a parent of both Rec 1 and Rec 2, Rec 1 is a parent of Lab 1, Lab 2 and Lab 3, etc.) defines what combinations of these classes are allowed: for example, a student taking Lab 5 must also take Rec 2 and Lec 1. If the course does not have a nesting defined, any combination of a lecture, a recitation, and a laboratory would be allowed. Preferences and requirements can be set on any level of the course structure (on a subpart, on an instructor, or an individual class) and overridden on a lower level.

We use an initial student scheduling algorithm that assigns students to classes in a way that tries to keep students with similar sets of course choices together, akin to the homogeneous sectioning introduced by in (Carter, 2001). Besides minimizing student conflicts during the search, there is a final sectioning algorithm executed at the end of the search that can further decrease the number of student conflicts by swapping students among alternative classes of a course. More details about this process are provided in (Müller and Murray, 2010).

There is a student conflict when a student is enrolled in two classes that are overlapping in time or that are placed in consecutive times and in rooms that are too far apart. The distance conflict occurs when the second class starts just after the first class and the travel time between the two classes is greater than the break time at the end of the first class[4].

During the search, the solver can distinguish between student conflicts that cannot be removed by student sectioning (called *hard student conflicts*) and student conflicts that may be removed by moving students around the course. It can also distinguish between student conflicts that are caused by the two classes overlapping in time or by the distance between the two consecutive

[4] One of the improvements made to the solver is the ability to consider distance conflicts between classes that are not at adjacent times, bringing both the break time that is at the end of the first class as well as any time between the two classes into the equation.



	Demand	Mins Per Week	Limit	Manager	Date Pattern	Time Pattern	Preferences: Time	Preferences: Room	Preferences: Distribution	Instructor
M E 263 M E 263H	98		96							
Lecture		150	96	LLR	Full Term	3 x 50 2 x 75		WTHR Computer		
Recitation		100	96	M E	Full Term	2 x 50		ME 120 ME 236 Classroom		
Laboratory		50	84-120	LAB	Even Wks	1 x 50		Windows XP		
Lec 1		150	96	LLR	Full Term	3 x 50 2 x 75		WTHR Computer		**J. Smith** C. Bing
Rec 1		100	48	M E	Full Term	2 x 50		ME 120 ME 236 Classroom	Back-To-Back M E 263 Rec 1 M E 263 Rec 2	**J. Novak**
Lab 1		50	14-20	LAB	Even Wks	1 x 50		Windows XP		
Lab 2		50	14-20	LAB	Even Wks	1 x 50		Windows XP		
Lab 3		50	14-20	LAB	Even Wks	1 x 50		Windows XP		
Rec 2		100	48	M E	Full Term	2 x 50		ME 120 ME 236 Classroom	Back-To-Back M E 263 Rec 1 M E 263 Rec 2	**J. Novak**
Lab 4		50	14-20	LAB	Odd Wks	1 x 50		Windows XP		
Lab 5		50	14-20	LAB	Odd Wks	1 x 50		Windows XP		
Lab 6		50	14-20	LAB	Odd Wks	1 x 50		Mac Os X		

Fig. 4 Example course structure

classes (called *distance student conflicts*). A different weight can be put on each of these categories. For example, the weight of a student conflict that may be avoided by moving students around can be proportional to the anticipated success of the final sectioning algorithm in removing a certain percentage of these conflicts.

2.5 Distribution Constraints

There are additional distribution constraints that may be set between individual classes, which may be soft or hard. Hard distribution constraints must be satisfied, violations of soft distribution constraints are penalized. These constraints include[5]

- back-to-back (given classes must be placed on the same day, one just after the other),
- precedence (given classes must be placed in the given order),
- different time (given classes cannot overlap in time),
- spread in time (given classes must be spread in time as much as possible),

[5] The full list of distribution constraints that are included in the published 2007 data set from Purdue University is available at `http://www.unitime.org/uct_grconstraints_v23.php`. There have been more constraints added into the solver and the UniTime software since then, but those are not included in the presented data. The up to date list of available distribution constraints, including their description, can be found on the Distribution Types page in the UniTime application.

– can share room (given classes are allowed to share the same room, if the room is big enough),
– meet together (given classes must meet at the same time in the same room),
– same or different meeting days, times, or rooms.

Some distribution constraints are automatically derived from the course structure. For example, a class cannot overlap with its parent and all classes of a subpart are to be spread in time as much as possible to allow more choice for the students.

The same penalization schema of required, strongly preferred, preferred, discouraged, strongly discouraged, and prohibited is used for distribution constraints as it is for the time and room requirements. Required and prohibited constraints are considered hard (cannot be violated) and the others are soft (violations are to be minimized). It is important, however, to realize that a negative preference (either prohibited or discouraged) of a constraint can be reformulated as a positive preference of the opposite constraint. For example, prohibited *same room* has a meaning of required *different rooms*, where any two classes that are in the constraint cannot be placed in the same room. Similarly, prohibited back-to-back requires all classes to be placed on the same day, with no two classes overlapping in time or being back-to-back (one just after the other). Violations of a preferred or a discouraged constraint have a penalty of 1, violations of a strongly preferred or a strongly discouraged constraint have a penalty of 4. Some constraints do not have a soft or an opposite (discouraged or prohibited) version. For example, the *meet together* constraint can be only required.

3 Algorithm

The solver is based on an iterative forward search (IFS) algorithm (Rudová et al, 2011). This algorithm is similar to local search methods; however, in contrast to classical local search techniques, it operates over feasible, though not necessarily complete, solutions. In these solutions, some classes may be left unassigned. All hard constraints on assigned classes must be satisfied, however. Such solutions are easier to visualize and more meaningful to human users than complete but infeasible solutions. Because of the iterative character of the algorithm, the solver can also easily start, stop, or continue from any feasible timetable, either complete or incomplete.

The search is processed iteratively (see Fig. 5 for the algorithm), starting from an initial timetable ω that can be empty. During each step, a class is selected (Line 4). Typically an unassigned class is chosen. An assigned class may be selected when all classes are assigned but the timetable is not good enough, e.g., when there are still many violations of soft constraints. Once a class is selected, a placement from its domain is chosen for assignment containing both time placement and the desired number of rooms (Line 5). Even if the 'best' placement is selected, its assignment to the selected class may cause some hard conflicts with already assigned classes. Such conflicting classes are

```
1: function IFS(ω)
2:     σ = ω
3:     while canContinue(ω) do
4:         v = selectVariable(ω)
5:         d = selectValue(ω, v)
6:         γ = hardConflicts(ω, v/d)
7:         ω = ω\γ ∪ {v/d}
8:         if better(ω, σ) then σ = ω
9:     end while
10:    return σ
11: end function
```

Fig. 5 Pseudo-code of the iterative forward search algorithm.

computed (Line 6) and removed from the timetable and become unassigned. Finally, the selected placement is assigned to the class. The algorithm attempts to move from one (partial) feasible timetable to another via repetitive assignment of a selected placement to a selected class (Line 7). During this search, the feasibility of all hard constraints in each iteration step is enforced by removing conflicting classes (Line 7). The search is terminated when the desired timetable is found or when there is a timeout (Line 3). The best timetable found σ is remembered during the search (Line 8) and returned at the end (Line 10).

The algorithm makes use of a learning technique called Conflict-based Statistics (CBS) that has been developed to prevent cycling and to improve the quality of the final solution (Müller et al, 2004). There is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency and the assignments that caused them. More precisely, it is an array

$$CBS[A/a \ \rightarrow \ \neg\, B/b] = c_{ab}.$$

This means that the assignment of a class A with a placement a (denoted A/a) has caused a hard conflict c_{ab} times in the past with the assignment B/b. Note that this does not imply that the assignments A/a and B/b cannot be used together in the case of non-binary constraints. In the value selection function, each hard conflict is then weighted by its frequency, i.e., by the number of past unassignments of the current value of the conflicting variable caused by the selected assignment.

If a complete solution is not found, it is very likely due to an inconsistency in the input data. The content of the Conflict-based Statistics provides very valuable information back to the user which can be used to identify the problematic hard constraint or constraints that need to be relaxed.

It should be noted that while the algorithm is based on local search, the whole solver library is written in a constraint oriented manner, working with variables (classes), values (placement of a class in time and space), assignments (of values to variables), and hard and soft constraints. The solver framework does not allow for a hard constraint to be violated at any time. Moreover, Müller (2005) shows how arc consistency can be maintained by the framework

through the search and how the presented iterative forward search algorithm can be transformed into a complete search using dynamic backtracking, just like the algorithm described by Jussien et al (2000). While this is very interesting from the theoretical standpoint, maintaining full arc consistency does not seem to be very practical for course timetabling instances of any decent size as it is extremely memory and CPU consuming.

4 Data Sets

The Purdue University timetabling problem is naturally decomposed into a centrally timetabled large lecture problem, individually timetabled departmental problems, and a centrally timetabled computer laboratory problem. The data sets in our experiments consist of 8 departmental problems each. Each of these sub-problems may have characteristics that are different from other problems in the set. Some of the different attributes for selected problems are listed in Table 1, all data sets for these problems are available from `http://www.unitime.org/uct_datasets.php`.

Table 1 Main characteristics of the considered problems.

Problem	Classes	Meetings per week	Hours per class	Classes per subpart	Students	Classes per students	Rooms	Room capacity (min-max)	Distribution constraints per class	Times per class	Rooms per class
pu-spr07-llr	803	2.09	2.40	1.25	27881	3.15	55	40-474	0.69	10.80	16.49
pu-fal07-llr	891	2.07	2.32	1.26	30855	3.23	55	40-474	0.71	13.00	18.89
pu-spr07-ms	440	2.32	2.43	3.52	11992	1.11	25	24-51	2.74	8.00	8.82
pu-fal07-ms	525	2.35	2.40	4.45	14331	1.10	33	24-61	2.18	8.79	4.92
pu-spr07-cs	93	1.63	2.14	1.82	725	2.03	13	17-61	2.83	13.77	1.94
pu-fal07-cs	174	1.31	1.92	2.72	2002	1.57	13	22-61	2.49	17.00	2.24
pu-spr07-ecet	107	1.17	2.57	2.55	648	2.30	9	16-49	2.78	25.14	1.00
pu-fal07-ecet	113	1.21	2.54	2.46	662	2.60	9	16-49	2.54	24.62	1.00
pu-spr07-sa	69	1.67	2.30	1.50	1312	1.40	4	40-48	1.25	18.44	2.15
pu-fal07-sa	63	2.00	2.43	1.47	925	1.13	6	14-48	0.92	20.60	3.27
pu-spr07-cfs	214	1.44	2.91	2.21	1610	1.94	29	10-71	1.79	11.24	2.19
pu-fal07-cfs	201	1.38	2.90	2.14	1936	1.75	28	10-71	3.28	11.00	2.86
pu-spr07-vpa	249	1.71	3.24	1.64	1836	2.17	47	10-45	2.06	12.98	2.48
pu-fal07-vpa	290	1.59	2.92	1.72	1747	2.22	41	10-45	1.26	1.23	1.10
pu-spr07-lab	443	1.25	1.97	4.82	8421	1.14	36	20-45	2.05	16.66	16.65
pu-fal07-lab	200	1.20	1.81	3.70	4835	1.08	31	20-45	3.29	19.79	16.80
pu-spr07-c8	2418	1.81	2.45	1.95	29514	4.16	213	10-474	1.76	12.85	10.82
pu-fal07-c8	2457	1.85	2.40	1.90	32399	4.10	208	10-474	1.74	12.49	10.22

Problems with the suffix llr and lab stand for the large lecture room problem and the computer laboratory problem, respectively. The suffixes ms, cs, ecet, sa, cfs, and vpa denote data from departmental problems. The abbreviations fal07 and spr07 indicate the Fall 2007 and Spring 2007 semester. In addition to these problems, a combined data instance has been created containing classes from all of these problems (suffix c8). This problem contains nearly 2,500 classes with more than 32,000 students and 200 rooms.

The problems presented in Table 1 are related to each other. The large lecture problem is solved first. It includes classes from all departments requiring students to be scheduled into the pool of shared large lecture rooms. Consequently, departmental problems can be solved taking into account their large classes assigned earlier. Finally, the shared computer laboratory problem is solved. Since this is an easy problem (many computer labs have similar equipment and capacity, and there are fewer restrictions on time) it can be solved after the departmental problems.

Table 1 gives the main characteristics of each problem, including the number of *classes*, the average number of *meetings per week*, and the average number of *hours per class*. Alternative classes of a course that are of the same instructional type are grouped into subparts (see the average *classes per subpart*). Other important characteristics given are the number of *students* and the classes to which these students are enrolled (see the average *classes per students*). Each problem also has a specified set of *rooms* where classes should be scheduled and each class has a specified number of seats. Table 1 includes information about minimum and maximum *room capacity* and the average number of *distribution constraints per class*. The last two columns, named *Times per class* and *Rooms per class*, show the average number of time and room placements that are available to a class respectively.

5 Experiments

Results computed on the two combined problems (pu-fal07-c8 and pu-spr07-c8) by different solver builds are presented in Figure 6. The first point shows the solver that was used in the experiments in the previous publication (Rudová et al, 2011). The later points show how the solver improved on these data sets since then. Figure 6 shows a simplified objective of the solver, only combining the four major characteristics of a solution. These are: the number of student conflicts, satisfaction of time and room preferences, and satisfaction of soft distribution constraints. The value of zero would mean a solution with no student conflicts, with all the classes assigned to their best available times and rooms and with no soft distribution constraints violated. On the horizontal axis there are solvers from different times, starting in March 2008 and ending with the latest version of the solver (CPSolver 1.3 as of December 2015, that is used in UniTime 4.1).

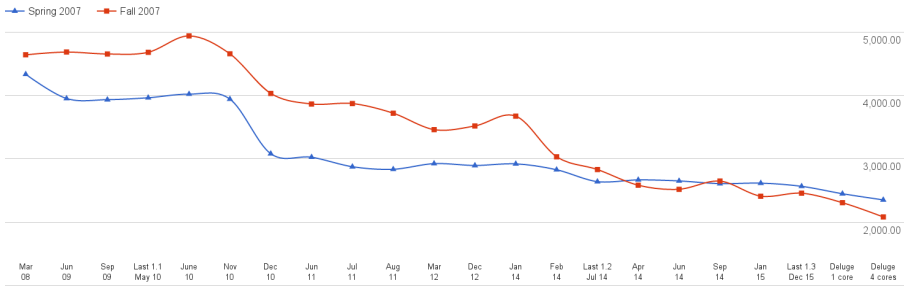


Fig. 6 Value of the objective function, compared using the latest solver.

All the instances use the same (*default*) configuration[6], except for the two last data points (denoted *Deluge*) which use a different algorithm once a complete solution is found (see Section 6.3). The very last data point is the same algorithm, but making use of 4 CPU cores instead of just one. The highest impact of the new algorithm can be seen in the soft distribution constraints as it does a better job on those that link three and more classes together. Each date was selected because there was a significant change made to the solver at that time. This is to be able to show how much each individual change helped. It also shows that certain changes had a negative impact (on these particular data sets), which can be seen on the charts by the objective going up.

It should be noted that each data point is an average of 10 independent runs with a 30 minute time limit, using a Mac Pro (Mid 2012) with two 6-Core Intel Xeon processors running at 3.06 GHz, 64 GB memory, OS X 10.11 and Java 8. To eliminate any changes in computing how well a particular objective is satisfied, the latest solver is used to validate and evaluate all of the resulting timetables.

For the Spring 2007 data set, the objective went down by 45.7% and it went down by 55.1% for Fall 2007. These changes are 38.0% and 31.6%, respectively, when evaluated by the solver from March 2008 (see Table 2 below). The most interesting improvements that stand behind these results are discussed in the following section.

Figure 7 shows a breakdown of the objective from Figure 6 into individual characteristics. *Student conflicts* shows the absolute number of student conflicts after the final sectioning has been run. *Time preferences* and *room preferences* are expressed as a percentage satisfied by the assigned class times and rooms. That is, 0% would mean all classes assigned in the worst possible time or room and 100% would mean all classes assigned to the best possible time or room respectively. Similarly, 100% for the *distribution preferences* would mean all soft distribution constraints are satisfied. Violations of soft distribution constraints are computed between individual pairs of classes even when a constraint is created between three or more classes. For example, a

[6] This configuration is also used as the default solver configuration in UniTime and has not changed significantly in many years, except for new weights and parameters covering the new constraints, criteria, and added complexity of the solver.

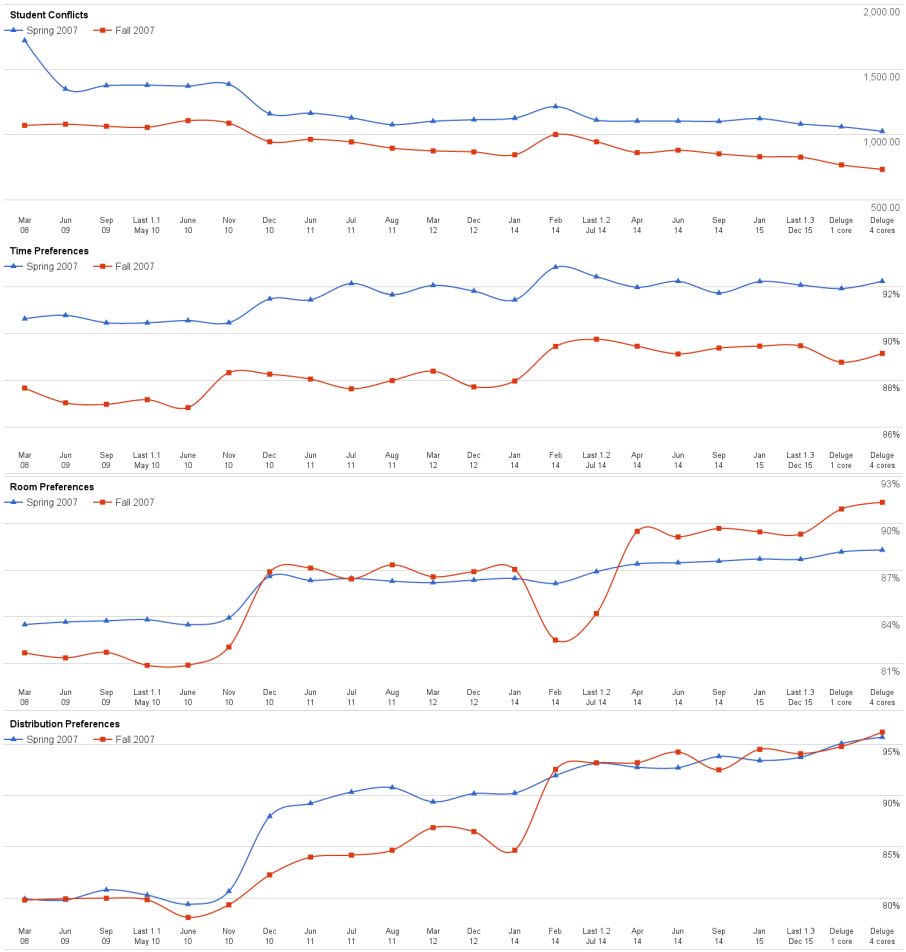


Fig. 7 Value of the four optimization critera, compared using the lastest solver.

different room distribution constraint that is created between four classes is 83.3% satisfied when there are two of the four classes in the same room, expressing that only one of the six pairs of the classes that are in the constraint is violated (see Figure 8).

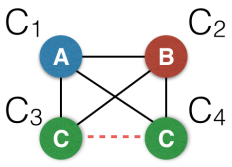


Fig. 8 Different room constraint between 4 classes: five of six class pairs are satisfied.

Table 2 shows the average results for a selected subset of the solvers used in Figures 6 and 7. As in the previous figures, the first solver (denoted **Mar 08**) corresponds to the solver used in the previous paper (Rudová et al, 2011). The solver denoted **Dec 15** is the most recent version of the solver library in the experiment. The last two columns correspond to the solver using a different algorithm once a complete solution is found (based on Great Deluge, see Section 6.3). All the solvers in the experiment use just one CPU core, except for the last test which makes use of four CPU cores. Results from both data sets are included (Spring 2007 and Fall 2007).

Table 2 Results of selected solver versions.

Characteristics	Solver Version	CPSolver 1.1 Mar 08	May 10	CPSolver 1.2 Dec 10	Aug 11	Jul 14	CPSolver 1.3 Dec 15	Great Deluge	Great Deluge 4 cores
Objective	Spring 07	4333.6	3963.2	3081.0	2835.3	2640.3	2566.9	2450.1	2353.1
using Dec 15 solver	Fall 07	4640.4	4679.7	4032.3	3720.3	2832.0	2459.3	2308.5	2083.6
Improvement	Spring 07		8.55	28.90	34.57	39.07	40.77	43.46	45.70
of Mar 08 [%]	Fall 07		-0.85	13.10	19.83	38.97	47.00	50.25	55.10
Student	Spring 07	1721.9	1379.1	1159.7	1075.7	1111.4	1080.9	1059.4	1024.4
Conflicts	Fall 07	1069.8	1055.6	944.6	895.3	944.7	825.9	766.9	732.9
Time	Spring 07	90.64	90.46	91.49	91.67	92.44	92.08	91.93	92.25
Preferences [%]	Fall 07	87.67	87.17	88.26	87.99	89.76	89.48	88.78	89.15
Room	Spring 07	83.51	83.82	86.60	86.29	86.90	87.69	88.17	88.28
Preferences [%]	Fall 07	81.70	80.90	86.89	87.33	84.22	89.30	90.91	91.33
Distribution	Spring 07	79.95	80.31	87.97	90.77	93.13	93.72	95.05	95.68
Preferences [%]	Fall 07	79.83	79.86	82.28	84.68	93.19	94.06	94.77	96.17
Objective	Spring 07	3650.0	3288.9	2637.8	2529.5	2459.6	2404.6	2345.3	2264.8
using Mar 08 solver	Fall 07	3390.0	3438.1	2911.9	2824.0	2899.7	2548.8	2396.4	2318.0
Improvement	Spring 07		9.89	27.73	30.70	32.61	34.12	35.74	37.95
of Mar 08 [%]	Fall 07		-1.42	14.10	16.70	14.46	24.81	29.31	31.62

The *Objective* lines show the average value of the solution found, presented in the same way as in Figure 6. Smaller values mean a better solution; a perfect solution with no student conflicts, no soft distribution constraint violations and with all classes having the best possible times and rooms assigned would have a zero value. The *Improvement* lines show improvement of the solution over the first solver in the test (denoted **Mar 08**). The remaining lines show the individual characteristics of the solution, expressed as percentages computed in the same way as in Figure 7, except for student conflicts which are shown in absolute numbers.

Like in the previous figures, all the results in the upper part of the Table 2 (above the double horizontal line) are evaluated using the most recent solver in the test (denoted **Dec 15**). The bottom part of the table shows the same results evaluated by the first solver in the test (denoted **Mar 08**). Most of the differences are caused by the way soft distribution constraints are evaluated

(see Section 6.2) and in the way that weighted student conflicts are counted (see Section 6.1).

While there have been a lot of changes done in the solver over the years, the solver remains backward compatible. In the above experiments, we have taken the data sets (pu-spr07-c8 and pu-fal07-c8) from our earlier works (Rudová et al, 2011) and we were able to run the different builds of the solver on them without making any changes to the data sets or the solvers. All the produced results were validated by the first and the last solver in the test. On the other hand, this means that we were not able to demonstrate any improvements on the constraints that were added later and that are not present in the discussed data sets.

6 Improvements

The solver framework[7], called CPSolver thanks to its constraint-based modeling, is written in Java and freely available under the GNU Lesser General Public License. It contains a constraint layer where most of the search algorithms and heuristics are implemented and three separate modules for course timetabling, examination timetabling, and student scheduling, where the problem specific aspects of these problems are implemented.

Since the last paper (Rudová et al, 2011) featuring CPSolver 1.1, there have been two major releases of the solver library and most of the code has been rewritten. In CPSolver 1.2, the solver framework was rewritten to make use of Java 5 Generics and to allow the solver objective to be split into individual criteria. This has helped in making the solver much easier to customize and extend. In CPSolver 1.3, a more versatile assignment model was implemented allowing better support for multiple solver threads and parallel algorithms. A few new algorithms have also been implemented in this version. Besides these more general improvements, a lot of work has also been done on the individual modules, making the UniTime system capable of handling problem aspects from more institutions with different needs.

6.1 Student Sectioning

A lot of the improvements that have been made revolve around the student sectioning algorithm and its ability to move students around before, during or after the search.

Before the **Jun 09** release, the final student sectioning algorithm was only able to swap two students between a pair of alternative classes (classes of the same subpart). Now the solver can also move a student from one section to another, given that the other section has space available in it.

Please note that this change only impacts the Spring 2007 data set as the Fall 2007 data set is using weighted students. This means that students are

[7] `http://www.cpsolver.org`

weighted such that all spaces in a course are completely filled. For example, if there is a course with 30 spaces, but only 20 student requests, each student takes up 1.5 spaces in the course. This was done to ensure there are no alternative sections left with no students and was used in production for the first time in Fall 2007.

The **Aug 11** release also added the ability to swap students between different parent classes. Before, a student swap was initiated only between classes with either no parent or the same parent. For example, consider two lectures (Lec 1 and Lec 2), each with two labs (Lab 1 and Lab 2 for Lec 1 and Lab 3 and Lab 4 for Lec 2). Before the change, a student was only swapped between Lec 1 and Lec 2 if there was an improvement between the two lectures (the swap was never initiated between two labs of different parents). Now, an improvement between Lab 1 and Lab 3 (different lecture) is also considered and propagated upwards.

6.2 Distribution Constraints

In the **Dec 10** release, the computation and assignment of the *Meet Together* constraint was improved. This change ensured that classes that were set up to meet together were not partially assigned to a time and room that would not work for all of the classes in the constraint. This did not have any impact on the Spring 2007 data set as there are no Meet Together constraints in that set. There are 25 such constraints in Fall 2007, however.

The most important change in the **Jun 11** release is a revision of the weighting schema for the distribution preferences. Before, there was a penalty considered every time a soft distribution constraint was violated, regardless of its violation. Because some constraints span many classes, a weighting model counting the number of violations, (e.g., pairs of classes in the constraint that are in a conflict) was adopted.

Consider, for example, a different room constraint between four classes. The former model put the same penalty in the objective function regardless of how many of the four classes were put in the same room. The new model tries to capture how badly the constraint is violated instead. While this change put a little less pressure on the whole constraint being satisfied, the overall acceptance of the solution improved.

Forward checking over all hard distribution constraints was added in the **Feb 14** release. When a potential placement of a class is checked for conflicts, for each hard distribution constraint that includes the class, it is not only checked that the placement satisfies the constraint, but also whether the remaining variables (classes) of the constraint can be assigned in a way that satisfies the constraint. This means that the domain of each of the remaining variables is checked for a value (placement of the class in time and space) that supports the proposed assignment of the constraint. If there is only one supporting value in the domain, all the other hard constraints are checked for conflicts with this placement as well. Many additional improvements have

been made to the forward checking algorithm in the following release **Jul 14**, including limiting search depth.

Sequence checking for the *Back-to-Back* constraint was also improved in the **Jul 14** release. The Back-to-Back constraint puts all classes on the same day or days in a sequence with no break times in between. A partially assigned constraint is considered satisfied if and only if the partial assignment can be extended in a way that would satisfy the constraints. For example, consider a Back-to-Back constraint between three hour long classes. Such a constraint is satisfied if two of these three classes are assigned next to each other while it is possible to attach the last class before or after the pair. The constraint is also satisfied if the two classes have an hour in between while the last class can be placed in that hour. In this release, the sequence check of the Back-to-Back constraint was changed to return the smallest set of variables that must be unassigned in order to make the rest of the constraint plus the proposed new assignment satisfied that does not contain the new assignment (the selected value of the selected variable in the iterative forward search algorithm, see Figure 5). Previously, it could have returned the new assignment if any other possibility would have had to unassign more classes.

6.3 Search Algorithms

One of the improvements in CPSolver 1.1 was the ability to easily plug-in different search algorithms that do not necessarily follow the iterative forward search schema (one variable is selected and re-assigned in each step, possibly with some additional unassignments automatically carried over by the framework). Many search algorithms and heuristics have been included over the years. In CPSolver 1.3, a hybrid algorithm based on the International Timetabling Competition entry by Müller (2009) was introduced. It combines the iterative forward search during the construction phase with the *Great Deluge* (GD) algorithm (Dueck, 1993) once a complete solution is found.

In each iteration, there is a change to the solution generated and proposed (see Fig. 9 for the algorithm, Line 5). There is a bound B on the objective function which starts higher than the best known value (given by the upper bound UB, e.g., with a 5% slack, Line 2). A proposed change is only accepted when the solution does not go over the bound B (Line 6). As the search progresses, the bound goes down using the *cooling rate* parameter CR (Line 9). When the bound gets below the objective, only changes that are not worsening the solution are accepted (Line 6). The idea is to give the search a bit of slack at the beginning (to be able to escape a local optimum) but force it to settle as the slack between the solution value and the bound disappears. The bound is increased again (reheated) when the solver is not able to improve the solution for some time. This occurs when the bound B gets bellow the lower bound (computed from the best known solution using the LB parameter, e.g., when the bound gets below the objective by 5%, Line 10).

```
1: function GD(ω)
2:     B = value(ω) · UB,  σ = ω,  at = 1
3:     while CANCONTINUE(ω) do
4:         n = SELECTNEIGHBORHOOD()
5:         δ = GENERATE(σ, n)
6:         if δ ≠ NULL and (value(ω ⊗ δ) ≤ value(σ) or value(ω ⊗ δ) ≤ B) then
7:             ω = ω ⊗ δ
8:             if value(ω) < value(σ) then σ = ω,  at = 1
9:         B = B · CR
10:        if B < LB^at · value(σ) then B = UB^at · value(σ),  at = at + 1
11:    end while
12:    return σ
13: end function
```

Fig. 9 Pseudo-code of the Great Deluge algorithm.

The parameter at is a counter starting at 1. It is increased by one every time the lower limit is reached and the bound is increased (Line 10). It is also reset back to 1 when a previous best solution is improved upon. This helps the solver to widen the search when it cannot find an improvement, allowing it to get out of a deep local minimum.

During the search, the algorithm makes use of various neighborhoods that can be called with variable probability (Line 4). Each neighborhood proposes a particular change to the current solution. There are some problem independent neighborhoods, including random reassignment of a variable, random swap of two or more variables, or a change generated by a branch-and-bound algorithm of a limited depth. There is also a few problem specific neighborhoods, moving a class to a different time or room, or swapping two or more classes in time or space. All of the swaps work in a chain. For example, in a time swap an available time is randomly selected for a class. If there is no conflict, the proposed change is returned, if there is just one conflict, it tries to resolve the conflict in the same manner (propose some other time for the conflicting class), if there are two or more conflicts the following time for the class is selected instead. The search fails if there are no more available times for the class, or when the number of times with just one conflict that it failed to resolve reaches a given limit (typically three).

A lot of the effectiveness of this algorithm lays in how quickly the bound B goes down. If it goes down too slowly, the algorithm may not have enough time to settle. If it goes down too fast, there may not be enough time for the algorithm to get far enough from the local optimum to be able to explore a different region of the search space as it settles. The objective is just a number and there is no indication how far it is from the optimum. While the cooling rate CR is relative to the bound (it does not matter if the solution value is 100 or 10000, the number of iterations it takes to decrease the bound by 1% is the same), it still has a big impact on how much time the solver should be given.

On the other hand, when the cooling rate is carefully selected, a hybrid algorithm that makes use of GD once a complete solution is found can out-

perform IFS as was demonstrated in Chapter 5. Please note that unlike with IFS, a solution always stays complete during the great deluge phase. This means that a change that would introduce a hard conflict is never generated or accepted.

6.4 **Multiple Solver Threads**

Some work has also been done in exploring the use of multiple CPU cores by running multiple solver threads in parallel. The CPSolver 1.3 allows for multiple solver threads working in two modes. In the first mode, all threads share a single solution, proposing and assigning changes in parallel. In the second mode, each solver thread has its own solution and only certain information is shared between these threads, e.g., the value of the best solution ever found.

The last data point displayed from the experiments presented in Section 5, Figures 6 and 7 and Table 2, makes use of 4 solver threads. Each solver thread works with its own solution (second mode) and the value of the best solution is shared between the solver threads during the great deluge phase.

7 Conclusions

Much work has been done on the CPSolver library and its components over the years since we have published the last study of this problem (Rudová et al, 2011). As the work presented on the two publicly available data sets from Purdue University shows, considerable progress continues to be made on improving the solutions achieved. This has been the result of work on both solution algorithms and improving how problem constraints are implemented.

More work can certainly be done on the multi-core approach of the solver, though the course timetabling problem does not seem to be very well suited for parallelization as there can be a lot of interaction between almost any pair of classes. There is also a great deal of data that the solver keeps associated with a particular assignment in our implementation (e.g., every room holds its own schedule for fast conflict checking) that needs to be synchronized, access controlled, or often invalidated and recomputed. On the other hand, the multiple solver threads working with a single assignment approach works very well on the student scheduling problem (Müller and Murray, 2010). If each thread is working on a different student, the only interaction is through the class and course limits when space in the course starts running low.

The evolution of UniTime over the past several years indicates that there is much to be confident about moving forward with automated timetabling. The existence of such frameworks that allow real-life problems to be modeled and solved, yet allow the flexibility to introduce new solution methods as research in the field evolves, provides a means of proving the efficacy of new techniques on practical problems and comparing results for problems with a wide range of characteristics.

References

Bonutti A, De Cesco F, Di Gaspero L, Schaerf A (2012) Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results. Annals of Operations Research 194(1):59–70

Carter MW (2001) A comprehensive course timetabling and student scheduling system at the University of Waterloo. In: Burke E, Erben W (eds) Practice and Theory of Automated Timetabling III, Springer-Verlag LNCS 2079, pp 64–82

Dueck G (1993) New optimization heuristics: The great deluge algorithm and the record-to record travel. Journal of Computational Physics 104:86–92

Jussien N, Debruyne R, Boizumault P (2000) Maintaining arc-consistency within dynamic backtracking. In: Principles and Practice of Constraint Programming–CP 2000, Springer, pp 249–261

McCollum B (2007) A perspective on bridging the gap between theory and practice in university timetabling. In: Burke E, Rudová H (eds) Practice and Theory of Automated Timetabling VI, Springer-Verlag LNCS 3867, pp 3–23

McCollum B, Schaerf A, Paechter B, McMullan P, Lewis R, Parkes AJ, Di Gaspero L, Qu R, Burke EK (2010) Setting the research agenda in automated timetabling: The second international timetabling competition. INFORMS Journal on Computing 22(1):120–130

McCollum B, McMullan P, Parkes AJ, Burke EK, Qu R (2012) A new model for automated examination timetabling. Annals of Operations Research 194(1):291–315

Müller T (2005) Constraint-based timetabling. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, URL `http://muller.unitime.org/phd-thesis.pdf`

Müller T (2009) Itc2007 solver description: A hybrid approach. Annals of Operations Research 172(1):429–446

Müller T, Murray K (2010) Comprehensive approach to student sectioning. Annals of Operations Research 181:249–269

Müller T, Murray K (2010) Comprehensive approach to student sectioning. Annals of Operations Research 181(1):249–269

Müller T, Rudová H (2014) Real-life curriculum-based timetabling with elective courses and course sections. Annals of Operations Research pp 1–18

Müller T, Barták R, Rudová H (2004) Conflict-based statistics. In: Gottlieb J, Silva DL, Musliu N, Soubeiga E (eds) EU/ME Workshop on Design and Evaluation of Advanced Hybrid Meta-Heuristics, University of Nottingham

Rudová H, Müller T, Murray K (2011) Complex university course timetabling. Journal of Scheduling 14(2):187–207