

Minimal Perturbation Problem in Course Timetabling

Tomáš Müller¹, Hana Rudová² and Roman Barták¹

¹ Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, Prague, Czech Republic
{muller|bartak}@ktiml.mff.cuni.cz

² Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
hanka@fi.muni.cz

Abstract Many real-life problems are dynamic, with changes in the problem definition occurring after a solution to the initial formulation has been reached. A minimal perturbation problem incorporates these changes, along with the initial solution, as a new problem whose solution must be as close as possible to the initial solution. A new iterative forward search algorithm is proposed to solve minimal perturbation problems. Significant improvements to the solution quality are achieved by including new conflict-based statistics in this algorithm. The proposed methods were applied to find a new solution to an existing large scale class timetabling problem at Purdue University, incorporating the initial solution and additional input changes.

1 Introduction

Most existing solvers are designed for static problems. These problems can be expressed, solved by appropriate means, and the solution applied without any change to the problem statement. Many real-life problems [18, 11, 9, 16, 12], however, are subject to change. Additional input requirements produce a new problem derived from the original problem. The dynamics of such a problem may require changes during the solution process, or even after a solution is generated. In many real-life situations, it is necessary to alter the solution process so that the dynamic aspects of the problem definition are taken into account.

Problem changes may result from changes to environmental variables, such as broken machines, delayed flights, or other unexpected events. Users may also specify new properties based on the solution found so far. The goal is to find an improved solution for the user. Naturally, the problem solving process should continue as smoothly as possible after any change in the problem formulation. In particular, the solution of the altered problem should not differ significantly from the solution found for the original problem. There are several reasons to keep a new solution as close as possible to the existing solution. If the solution has already been published, such as the assignment of gates to flights, frequent changes would confuse passengers. Moreover, changes to a published solution

may necessitate other changes if initially satisfied wishes of users are violated by the proposed changes. This may create an avalanche reaction.

In this paper we focus on handling dynamic changes in course timetabling. In particular, our work is motivated by the class timetabling problem at Purdue University [15]. Here timetables for each semester are created nearly a semester in advance. Once timetables are published they require many changes based on additional input. These changes must be incorporated into the problem solution with minimal impact on any previously generated solution. Thus, the primary focus of our work is to provide support for making minimal changes to the generated timetable.

Our problem solver is based on constraint satisfaction techniques [4] which are frequently applied to solve timetabling problems [7, 3, 15, 12]. Moreover, dynamic constraint satisfaction [18, 11, 17] is able to cover dynamic aspects in the problem. The minimal perturbation problem as defined in [1, 16] allows us to express our desire to keep changes to the solution (perturbations) as small as possible.

We introduce a new iterative forward search algorithm to solve the minimal perturbation problem. It is based on our earlier work on solving methods for the static (initial) problem [12]. The method also allows us to solve the initial problem. The basic difference in application is that the optimization of the number of changes (perturbations) is not included while solving the initial problem. Our algorithm is close to local search methods [10]; however, it maintains partial feasible assignments as opposed to the complete conflicting assignments characteristic of local search. Similar to local search, we process local changes in the assignment. This allows us to generate a complete solution and to improve the quality of the assignment at the same time.

New conflict-based statistics are proposed to improve the quality of the final solution. Conflicts during the search are memorized and their potential repetition is minimized. Conflict-based heuristics have been successfully applied in earlier works [5, 8]. In our approach, the conflict-based statistics work as advice in the value selection criterion. They help to avoid repetitive, unsuitable assignments of the same value to a variable by memorizing conflicts caused by this assignment in the past. The proposed heuristics do not limit the number of conflicts and assignments that are memorized. We have extended our search algorithm using these conflict-based statistics. Note, however, that this is a general strategy that can be applied in other problem solvers.

The paper is organized as follows. Section 2 describes the timetabling problem at Purdue University that motivates our work. The subsequent section introduces the minimal perturbation problem and surveys related works on dynamic problems. Section 4 describes the iterative forward search algorithm. The subsequent section introduces conflict-based heuristics and defines how they have been included in the search algorithm. The solution of our class timetabling problem is discussed in Section 6. A short summary of the implemented system, along with experimental results for the initial and minimal perturbation problems, conclude the paper.

2 Motivation – Timetabling Problem

The primary purpose of our work is to solve a real timetabling problem at Purdue University (USA). Here the timetable for large lecture classes is constructed by a central scheduling office in order to balance the requirements of many departments offering large classes that serve students from across the university. Smaller classes, usually focused on students in a single discipline, are timetabled by “schedule deputies” in the individual departments. Such a complex timetabling process, including subsequent student registration, takes a rather long time. Initial timetables are generated about half a year before the semester starts. The importance of creating a solver for a dynamic problem increases with the length of this time period and the need to incorporate the various changes that arise.

Rescheduling of classes in the timetable for large lectures is the primary focus of this paper. This problem consists of about 830 classes (forming almost 1800 meetings) having a high density of interaction that must fit within 50 lecture rooms with capacities up to 474 students. Room availability is a major constraint for Purdue. Overall utilization of the time available in rooms exceeds 78%; moreover, it is around 94% for the four largest rooms. About 90,000 course requests by almost 30,000 students must also be considered. 8.4% of class pairs have at least one student enrollment in common.

The timetable maps classes (students, instructors) to meeting locations and times. A major objective in developing an automated system is to minimize the number of potential student course conflicts which occur during this process. This requirement substantially influences the automated timetable generation process since there are many specific course requirements in most programs of study offered by the University.

To minimize the potential for time conflicts, Purdue has historically subscribed to a set of standard meeting patterns. With few exceptions, 1 hour \times 3 day per week classes meet on Monday, Wednesday, and Friday at the half hour (7:30, 8:30, 9:30, ...). 1.5 hour \times 2 day per week classes meet on Tuesday and Thursday during set time blocks. 2 or 3 hours \times 1 day per week classes must also fit within specific blocks, etc. Generally, all meetings of a class should be taught in the same location. Such meeting patterns are of interest to the problem solution as they allow easier changes between classes having the same or similar meeting patterns.

Currently, the timetable for Purdue University is constructed by a manual process. We have proposed an automated timetabling system to solve the initial problem [15]. This solution was based on constraint logic programming (CLP) with soft constraints. The CLP solver is currently undergoing comparison with the new solver described in this paper.

3 Minimal Perturbation Problem and Related Works

Dynamic problems appear frequently in real-life planning and scheduling applications where the task is to “minimally reconfigure schedules in response to

a changing environment”[16]. Dynamic changes in the context of timetabling problems were first studied in [6]. Issues of interactive timetabling which needs to handle dynamic aspects of the problem were discussed in [3, 13, 12]. A survey of existing approaches to dynamic scheduling can be found in [9]. In an annotated bibliography included in the CP 2003 tutorial on dynamic constraint solving [17], it is notable that only four papers were devoted to the problem of minimal changes. An extended version of this tutorial will be published in [18]. The minimal perturbation problem was described formally in [16] and solved by a combination of linear and constraint programming. We have extended this definition in [1] and proposed a solution algorithm based on the Branch & Bound algorithm. An algorithm inspired by heuristic repair and limited discrepancy search has also been proposed in [14].

To define the minimal perturbation problem, we will consider an initial (original) problem, its solution, a new problem, and some distance function which allows us to compare solutions of the initial and the new problem. Subsequently we look for a solution of the new problem with minimal distance from the initial solution. Let us now look at particular components (for detail information look at [1]).

We can define both the initial and the new problem as a constraint satisfaction problem (CSP) [4]. It is a triple (V, D, C) where V is a finite set of variables, D is a set of possible values for variables (domain), and C is a finite set of constraints restricting the values of variables. A solution to a CSP is a complete assignment of the variables that satisfies all the constraints.

A distance function can be defined with the help of perturbations [1, 16, 14]. A perturbation is a variable that has a different value in the solutions of the initial and the new problem. Some perturbations must be present in each new solution. So called input perturbation means that a variable must have different values in the initial and changed problem because of some input changes (e.g., a course must be scheduled at a different time in the changed problem). The distance function can be defined as the number of additional perturbations. They are given by subtraction of the final number of perturbations and the number of input perturbations.

4 Iterative Forward Search Algorithm

In this section, an iterative forward search algorithm and its general setting are presented. It is based on local search methods [10], but in contrast to classical local search techniques, it operates over a feasible, though not necessarily complete, solution. In such a solution, some variables can be left unassigned; however, all hard constraints on assigned variables must be satisfied. Similar to backtracking-based algorithms, this means that there are no violations of hard constraints.

Working with feasible incomplete solutions has several advantages compared to the complete infeasible solutions that usually occur in local search techniques. For example, when the solver is not able to find a complete solution, a feasible

one can be returned, e.g., a solution with the least number of unassigned variables found. Especially in interactive timetabling applications, such solutions are much easier to visualize, even during the search, since no hard constraints are violated. For instance, two lectures never use a single resource (e.g., a classroom) at the same time. Moreover, because of the iterative character of the search, the algorithm can easily start, stop, or continue from any feasible solution, either complete or incomplete.

The search is processed iteratively (see Fig. 1 for the algorithm). During each

```

procedure SOLVE(initial)           // initial solution is the parameter
  iteration = 0;                   // iteration counter
  current = initial;               // current solution
  best = initial;                  // best solution
  while canContinue(current, iteration) do
    iteration = iteration + 1;
    variable = selectVariable(current);
    value = selectValue(current, variable);
    unassigned = CONFLICTING_VARIABLES(current, variable, value);
    UNASSIGN(current, unassigned);
    ASSIGN(current, variable, value);
    if better(current, best) then best = current
  end while
  return best
end procedure

```

Figure 1. Pseudo-code of the search algorithm.

step, either an unassigned or an assigned variable may be selected. Typically an unassigned variable is chosen. An assigned variable may be selected when all variables are assigned but the solution is not good enough, for example when there are still many violations of soft constraints. Once a variable is selected, a value from its domain is chosen for assignment. Even if the ‘best’ value is selected, its assignment to the selected variable may cause some hard conflicts with already assigned variables. Such conflicting variables are removed from the solution and become unassigned. Finally, the selected value is assigned to the variable. The search is terminated when the desired solution is found or when there is a timeout, expressed e.g., as a maximal number of iterations or available time being reached. The best solution found is then returned.

Each current solution must be feasible at all times, but an assignment of a value to a variable may cause conflicts with other variables. For example, let the values of variables A , B and C must be different, and variable A is assigned the value 3. When variable B , together with the value 3, are selected during the following step, the value of A becomes unassigned during the assignment $B = 3$. In our algorithm, the function `CONFLICTING_VARIABLES` computes the set of conflicting variables that will be unassigned in the subsequent step.

The above algorithm schema is parameterized by several functions, namely

- the variable selection (function *selectVariable*),
- the value selection (function *selectValue*),
- the termination condition (function *canContinue*) and
- the solution comparator (function *better*).

These functions are discussed in the following sections.

Termination Condition. The termination condition determines when the algorithm should finish. For example, the solver should terminate when the maximum number of iterations or another given timeout value is reached. Moreover, it can stop the search process when the current solution is good enough (e.g., all variables are assigned and/or some other solution parameters are in the required ranges). As an example, the solver can stop when all variables are assigned and less than 10% of soft constraints are violated. The user may also terminate the process.

Solution Comparator. The solution comparator compares two solutions: the current solution and the best solution found. This comparison can be based on several criteria. For example, it can lexicographically order solutions according to the following criteria: the number of unassigned variables (a smaller number is better) or the number of violated soft constraints. Soft constraints can be weighted according to their importance and/or preferences. Then, a sum of weights of violated soft constraints can be used as the second criterion.

Variable Selection. As mentioned above, the presented algorithm requires a function that selects a variable to be (re)assigned during the current iteration step. This problem is equivalent to a variable selection criterion in constraint programming. There are several guidelines for selecting a variable [4]. In local search, the variable participating in the largest number of violations is usually selected first. In backtracking-based algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is most complicated is selected first. This could be the variable involved in the largest set of constraints or the variable with the smallest domain, etc.

The variable selection criterion is split into two cases. If some variables remain unassigned, the first-fail principle can be applied as the basis for selection. Other choice could be the random selection of the unassigned variable. Because the algorithm does not need to stop when a complete feasible solution is found, the second case occurs when all variables are assigned but the solution does not meet the termination condition. Here we choose the variable for which a change of its value appears to offer the best opportunity for improvement of the solution. This may be, for example, a variable whose value violates the highest number of soft constraints.

Value Selection. After a variable is selected, we need to find a value to be assigned. This problem is usually called “value selection” in constraint programming [4]. Typically, the most useful advice is to select the best-fit value. So, we are looking for a value which is the most preferred for the variable and also which causes the least trouble during the future search. This means that we need to find a value with minimal potential for future conflicts with other variables. Note that we are not using constraint propagation explicitly in our algorithm. However, the power of constraint propagation can be substituted to some extent by sophisticated value selection. It can take into account possible future conflicts by analyzing the past conflicts.

For example, a value which violates the smallest number of soft constraints among values with the smallest number of hard conflicts (i.e., the values whose assignment to the selected variable violate the smallest number of hard constraints) can be selected.

4.1 Adjustment for Solving A Minimal Perturbation Problem

Despite the local search nature of the IFS algorithm, there are some adjustments needed to effectively solve the MPP. The goal of these adjustments is to minimize the number of additional perturbations. The easiest way to do this is to adopt variable and value selection heuristics that prefer the previous assignments (but not always, to avoid cycling).

For example, value selection heuristics can be adopted to select the initial value (if it exists) randomly with a probability P_{init} (it can be rather high, e.g., 50-80%). In cases where the initial value is not selected, original value selection can be applied. Also, if there is an initial value in the set of best-fit values (e.g., among values with the minimal number of hard and soft conflicts), the initial value can be preferred here as well. Otherwise, a value can be selected randomly from the constructed set of best-fit values. A disadvantage of such heuristics is that the probability P_{init} has to be selected carefully: if it is too small, the search can easily move away and the number of additional perturbations will grow during the search. If it is too high, the search will stick too much with the initial solution and, if there is no solution with a small number of additional perturbations, it will be hard to find a feasible solution. We have achieved the best results using this probability P_{init} of around 60% (see Section 7.2 for details).

Variable selection heuristics can also assist in finding a solution with a small number of perturbations. For example, when all variables are assigned, a variable that is not assigned its initial value should be selected (e.g., randomly among all variables that are not assigned their initial values), and that participates in the highest number of violated soft constraints.

5 Conflict-based Statistics

In this section, a very promising extension of the iterative forward search algorithm is presented. The idea behind it is to memorize conflicts and to avoid

their potential repetition. When a value v_0 is assigned to a variable V_0 , hard conflicts with previously assigned variables (e.g., $V_1 = v_1, V_2 = v_2, \dots, V_m = v_m$) may occur. These variables V_1, \dots, V_m have to be unassigned before the value v_0 is assigned to the variable V_0 . These unassignments, together with the reason for their unassignment (i.e., the assignment $V_0 = v_0$), and a counter tracking how many times such an event occurred in the past, is stored in memory.

Later, if a variable is selected for assignment again, the stored information about repetition of past hard conflicts can be taken into account, e.g., in the value selection heuristics. Assume that the variable V_0 is selected for an assignment again (e.g., because it became unassigned as a result of a later assignment), we can weight the number of hard conflicts created in the past for each possible value of this variable. In the above example, the existing assignment $V_1 = v_1$ can prohibit the selection of value v_0 for variable V_0 if there is again a conflict with the assignment $V_1 = v_1$.

Conflict-based statistics are a data structure which memorizes the number of hard conflicts that have occurred during the search (e.g., that assignment $V_0 = v_0$ resulted c_1 times in an unassignment of $V_1 = v_1$, c_2 times of $V_2 = v_2$, \dots and c_m times of $V_m = v_m$). More precisely, they form an array

$$CBS[V_a = v_a, V_b \neq v_b] = c_{ab},$$

stating that the assignment $V_a = v_a$ caused the unassignment of $V_b = v_b$ a total of c_{ab} times in the past. Note that in case of n-ary constraints (where $n > 2$), this does not imply that the assignments $V_a = v_a$ and $V_b = v_b$ cannot be used together. The proposed conflict-based statistics do not actually work with any constraint, they only memorize unassignments and the assignment that caused them. Let us consider a variable V_a selected by the *selectVariable* function and a value v_a selected by *selectValue*. Once the assignment $V_b = v_b$ is selected by CONFLICTING_VARIABLES to be unassigned, the array cell $CBS[V_a = v_a, V_b \neq v_b]$ is incremented by one.

The data structure is implemented as a hash table, storing information for conflict-based statistics. A counter is maintained for the tuple $A = a$ and $B \neq b$. This counter is increased when the value a is assigned to the variable A and b is unassigned from B . The example of this structure

$$A = a \Rightarrow \begin{cases} 3 \times B \neq b \\ 4 \times B \neq c \\ 2 \times C \neq a \\ 120 \times D \neq a \end{cases}$$

expresses that variable B lost its assignment b three times and its assignment c four times, variable C lost its assignment a two times, and D lost its assignment a 120 times, all because of later assignments of value a to variable A . This structure is being used in the value selection heuristics to evaluate existing conflicts with the assigned variables. For example, if there is a variable A selected and if the value a is in conflict with the assignment $B = b$, we know that a similar problem

has already occurred $3\times$ in the past, and hence the conflict $A = a$ is weighted with the number 3.

Then, a *min-conflict* value selection criterion, which selects a value with the minimal number of conflicts with the existing assignments, can be easily adapted to a *weighted min-conflict* criterion. The value with the smallest sum of the number of conflicts multiplied by their frequencies is selected.

Stated in another way, the weighted min-conflict approach helps the value selection heuristics to select a value that might cause more conflicts than another value, but these conflicts occurred less frequently, and therefore they have a lower weighted sum. As we will show in Section 7.1, this can help considerably with getting the search algorithm out of a local optimum.

Extensions. The presented approach can be successfully applied in other search algorithms as well. For example, in the local search, we can memorize the assignment $V_x = v_x$, which was selected to be changed (re-assigned). A reason for such selection can be retrieved and memorized together with the selected assignment $V_x = v_x$ as well. Note that typically an assignment which is in a conflict with some other assignments is selected.

Furthermore, the presented conflict-based statistics can be used not only inside the solving mechanism. The constructed ‘implications’ together with the information about frequency of their occurrences can be easily accessed by users or by some add-on deductive engine to identify inconsistencies¹ and/or hard parts of the input problem. The user can then modify the input requirements in order to eliminate problems found and let the solver continue the search with this modified input problem.

6 Solution for Timetabling Problem

In this section we will discuss an application of the above described algorithm for the large lecture timetabling problem at Purdue University. The modeling part will be described first, followed by the description of the algorithm.

6.1 Problem Representation

Due to the set of standardized time patterns and administrative rules in place at the university, it is generally possible to represent all meetings of a class by a single variable. This tying together of meetings considerably simplifies the problem constraints. Most classes have all meetings taught in the same room, by the same instructor, at the same time of day. Only the day of week differs. Moreover, these days and times are mapped together with the help of meeting patterns, e.g., a 2 hours \times 3 day per week class can be taught only on Monday, Wednesday, Friday, beginning at 5 possible times (7:30, 9:30, 11:30, 1:30, 3:30).

¹ Actually, this feature allows discovery of all inconsistent data inputs during solution of the Purdue University timetabling problem.

In addition, all valid placements of a course in the timetable have a one-to-one mapping with values in the variable's domain. This domain can be seen as a subset of the Cartesian product of the possible starting times, rooms, etc. for a class represented by these values. Therefore, each value encodes the selected time pattern (some alternatives may occur, e.g., 1.5 hour \times 2 day per week may be an alternative to 1 hour \times 3 day per week), selected days (e.g., a two meeting course can be taught in Monday-Wednesday, Tuesday-Thursday, Wednesday-Friday), and possible starting times. A value also encodes the instructor and selected meeting room. Each such placement also encodes its preferences (soft constraints), combined from the preference for time, room, building and available equipment of the room. Only placements with valid times and rooms are present in a domain. For example, when a computer (classroom equipment) is required, only placements in a room containing a computer are present. Also, only rooms large enough to accommodate all the enrolled students can be present in valid class placements. Similarly, if a time slice is prohibited, no placement containing this time slice is in the class's domain.

The variable and value encodings described above leave us with only two types of hard constraints to be implemented: resource constraints (expressing that only one course can be taught by an instructor or in a particular room at the same time), and group constraints (expressing relations between several classes, e.g., that two sections of the same lecture cannot be taught at the same time, or that some classes have to be taught immediately after another).

There are three types of soft constraints in this problem. First, there are soft requirements on possible times, buildings, rooms, and classroom equipment (e.g., computer or projector). These preferences are expressed as integers:

- 2 ... strongly preferred
- 1 ... preferred
- 0 ... neutral (no preference)
- 1 ... discouraged
- 2 ... strongly discouraged

As mentioned above, each value, besides encoding a class's placement (time, room, instructor), also contains information about the preference for the given time and room. Room preference is a combination of preferences on the choice of building, room, and classroom equipment. The second group of soft constraints is formed by student requirements. Each student can enroll in several classes, so the aim is to minimize the total number of student conflicts among these classes. Such conflicts occur if the student cannot attend two classes to which he or she has enrolled because these classes have overlapping times. Finally, there are some group constraints (additional relations between two or more classes). These may either be hard (required or prohibited), or soft (preferred), similar to the time and room preferences (from -2 to 2).

6.2 Search Algorithm

In Section 4, we described four functions which parameterize the proposed algorithm. Here we will describe their exact settings in our timetabling solver.

The quality of a solution is expressed as a weighted sum combining soft time and classroom preferences, satisfied soft group constraints and the total number of student conflicts. This allows us to express the importance of different types of soft constraints. The following weights are considered in the sum:

- W_{student} ... weight of a student conflict,
- W_{time} ... weight of a time preference of a placement,
- W_{room} ... weight of a classroom preference of a placement,
- W_{constr} ... weight of a preference of a satisfied soft group constraint.

Note that preferences of all time, classroom and group soft constraints go from -2 (strongly preferred) to 2 (strongly discouraged). So, for instance, the value of the weighted sum is increased when there is a discouraged time or room selected or a discouraged group constraint satisfied. Therefore, if there are two solutions, the better of them has the lower weighted sum of the above criteria. Moreover, additional solution parameters can be included in this comparison as well. For instance, we can also discourage empty half-hour time segments between classes (such half-hours cannot be used since all events require at least one hour) or usage of classrooms that are too large (having more than 50% excess seats).

The termination condition stops the search when the solution is complete and good enough (expressed as the number of perturbations and the solution quality described above). It also allows for the solver to be stopped by the user. Characteristics of the current and the best achieved solution, describing the number of assigned variables, time and classroom preferences, the total number of student conflicts, etc., are visible to the user during the search.

The solution comparator prefers a more complete solution (with a smaller number of unassigned variables) and a solution with a smaller number of perturbations among solutions with the same number of unassigned variables. If both solutions have the same number of unassigned variables and perturbations, the solution of better quality is selected.

If there are one or more variables unassigned, the variable selection criterion picks one of them randomly. We have compared several approaches for variable selection using domain sizes, number of previous assignments, number of constraints in which the variable participates, etc. However, there was no significant improvement in this timetabling problem in comparison with the random selection of an unassigned variable. The reason could be, that it is easy to go back when a wrong variable is picked – such a variable is unassigned when there is a conflict with it in some of the subsequent iterations.

When all variables are assigned, an evaluation is made for each variable according to the above described weights. The variable with the worst evaluation is selected because this variable promises the best improvement in optimization.

We have implemented a hierarchical handling of the value selection criteria. There are three levels of comparison. At each level a weighted sum of the criteria described below is computed. Only solutions with the smallest sum are considered in the next level. The weights express how quickly a complete solution should be found. Only hard constraints are satisfied in the first level sum. Distance from the initial solution (MPP), and a weighting of major preferences (including time, classroom requirements and student conflicts), are considered

in the next level. In the third level, other minor criteria are considered. Such criterion could be for instance a usage of a room that is too large or a number of empty half-hour time segments between classes. In general, a criterion can be used in more than one level, e.g., with different weights.

The above sums order the values lexicographically: the best value having the smallest first level sum, the smallest second level sum among values with the smallest first level sum, and the smallest third level sum among these values. As mentioned above, this allows diversification of the importance of individual criteria. In general, there can be more than three levels of these weighted sums, however three of them seem to be sufficient for spreading weights of various criteria for our problem.

The value selection heuristics also allow for random selection of a value with a given probability P_{rw} (random walk, e.g., 2%) and, in the case of MPP, to select the initial value (if it exists) with a given probability P_{init} (e.g., 60%).

Criteria used in the value selection heuristics can be divided into two sets. Criteria in the first set are intended to generate a complete assignment:

1. Number of hard conflicts, weighted by $V_{conf,1}$ in the first level, $V_{conf,2}$ in the second level and $V_{conf,3}$ in the third level.
2. Number of hard conflicts, weighted by their previous occurrences (see conflict-based statistics section) and by $V_{wconf,1..3}$.

Additional criteria allow better results to be achieved during optimization:

3. Number of student conflicts caused by the value if it is assigned to the variable, weighted by $V_{student,1..3}$.
4. Soft time preference caused by a value if it is assigned to the variable, weighted by $V_{time,1..3}$.
5. Soft classroom preference caused by a value if it is assigned to the variable (combination of the placement's building, room, and classroom equipment compared with preferences), weighted by $V_{room,1..3}$.
6. Preferences of satisfied soft group constraints caused by the value if it is assigned to the variable, weighted by $V_{constr,1..3}$.
7. Difference in the number of assigned initial values if the value is assigned to the variable (weighted by $V_{\Delta init,1..3}$): -1 if the value is initial, 0 otherwise, increased by the number of initial values assigned to variables with hard conflicts with the value.

Let us emphasize that the criteria 3–7 are needed for optimization only, i.e., they are not needed to find a feasible² solution. Furthermore, assigning a different weight to a particular criteria influences the value of the corresponding objective function (see Fig. 3 with comparison between criteria 3 and 4). The solver returns good results in reasonable time (e.g., in 30 minutes time limit) when the total sum of the weights used in additional criteria (3–7) in the first level corresponds to one half of the weight $V_{wconf,1}$ (2). The weights in the second level usually correspond to the weights used for the solution quality comparison ($W_{student}$, W_{time} , W_{room} , and W_{constr}).

² Feasible solution must satisfy hard constraints.

7 Implementation and Experiments

The timetabling system is implemented in Java. It contains a general implementation of the iterative forward search algorithm described above. The general solver operates over variables and values with a selection of basic general heuristics, comparison, and termination functions. It may be customized to fit a particular problem (as it has been extended for Purdue University timetabling) by implementing variable and value definitions, adding hard and soft constraints, and extending the algorithm's parametric functions.

Besides the above discussed solver, the timetabling application for Purdue University also contains a web-based graphical user interface (written using Java Server Pages) which allows management of several versions of the data sets (input requirements, solutions, changes, etc.), browsing the resultant solutions (see Fig. 2), and tracking and managing changes between them.

The screenshot shows a web browser window titled "Large Lecture Room Timetabling v1.0 - Microsoft Internet Explorer". The address bar shows the URL "https://www.smas.purdue.edu/Tmtbl2004fa/index.jsp". The main content area is titled "Timetable" and displays a grid of class schedules for three different room sets: EE 270, EE 170, and EE 129. The grid is organized by day of the week (Mon, Tue, Wed, Thu, Fri) and time slots (7:30a, 8:30a, 9:30a, 10:30a, 11:30a, 12:30a, 1:30p, 2:30p, 3:30p). Each cell in the grid contains a course ID and a small table of values representing constraints or preferences. For example, in the EE 270 room set, on Monday at 8:30a, the course MSE230 1001 is scheduled with values (0, 2, 0). The sidebar on the left contains various configuration options, including "Purdue Timetable", "Input Configuration", "Administration", and "Solver".

Figure 2. Generated timetable at web-based graphical user interface.

The following experiments were performed on the complete Fall 2004 data set, including 830 classes to be placed in 50 classrooms. The classes included represent 89,677 course requirements for 29,808 students. The results presented here were computed on 1GHz Pentium III PC running Windows 2000, with

512 MB RAM and J2SDK 1.4.2. We have achieved similar results with Fall 2001 and Spring 2005 data sets as well, even though they are quite different in the number of requirements (Fall 2004 is the most constrained one out of these three data sets).

Below, we present two types of experiments. The first experiment investigates finding an initial solution. This is followed by experiments on the minimal perturbation problem (i.e., where there is an existing solution plus a set of changes to be applied to it). Solving an initial problem can be seen as a special case of MPP where all variables are new and therefore have no initial values.

If not stated otherwise, the solution quality weights W_{student} , W_{time} , W_{room} and W_{constr} in the solution quality weighted sum are set to zero in the following experiments. First level weight for the weighted hard conflicts $V_{\text{wconf},1}$ is set to 1, all other weights in the value selection criterion are set to zero. Also, there is no random value selection ($P_{\text{rw}} = 0$). This way, by default, only the hard constraints are considered during the search. We will show how the other weights influence the search process and the overall solution quality.

7.1 Initial Problem

The experiments in Table 1 present the behavior of the solver wrt. various settings of weights for particular criteria (the student conflicts, violated time preferences, and violated room preferences). It is important to see that the weights for particular criteria can be easily adjusted. It allows to emphasize or suppress particular optimization criteria and it results in the corresponding change of the solution quality.

Time refers to the amount of time required by the solver to find the presented solution. *Satisfied enrollments* gives the percentage of satisfied requirements for courses chosen by students. *Preferred time* and *preferred room* correspond to the satisfaction of time and room preferences respectively. 100% corresponds to a case when all classes are placed in their most preferred times or rooms, 0% means a case when the least preferred locations are used. Preferences of soft group constraints are not presented, since there are no such constraints in the Fall 2004 data set (all group constraints are either required or prohibited).

A complete solution was found on every run of all experiments in Table 1 except the experiment marked **No CBS**. Average values together with their RMS (root-mean-square) variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented.

The experiment marked **No Preference** presents average solutions obtained without any preferences on the soft constraints. All solution quality weights W and value selection weights V are set to zero, except of the weight $V_{\text{wconf},1} = 1$ (weight of the weighted hard conflicts in the first level of the value selection).

The following three experiments marked **Students**, **Time** and **Rooms** are minimizing just one of the criteria: the student conflicts, violated time preferences, or violated room preferences. **Students** experiment uses the same weights as **No Preference** experiment, but student weights are the following: $V_{\text{student},1} = 0.5$, $V_{\text{student},2} = W_{\text{student}} = 1$. Similarly, **Time** experiment uses weights $V_{\text{time},1} =$

Table 1. Solutions of the initial problem

Test case	No preference	Students	Time	Rooms
Assigned variables [%]	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
Time [min]	0.16 ± 0.03	9.18 ± 4.47	18.79 ± 7.35	0.17 ± 0.01
Satisfied enrollments [%]	98.26 ± 0.15	99.74 ± 0.02	98.19 ± 0.13	98.18 ± 0.24
Preferred time [%]	62.54 ± 1.19	65.57 ± 1.53	98.75 ± 0.13	62.14 ± 0.94
Preferred room [%]	63.64 ± 2.29	62.96 ± 1.67	63.72 ± 1.64	98.58 ± 0.29
Test case	Students + Time	Students + Time + Rooms	No CBS	
Assigned variables [%]	100.00 ± 0.00	100.00 ± 0.00	98.42 ± 0.20	
Time [min]	19.96 ± 5.34	14.79 ± 4.87	24.08 ± 4.42	
Satisfied enrollments [%]	99.61 ± 0.03	99.79 ± 0.01	99.52 ± 0.06	
Preferred time [%]	95.70 ± 0.32	95.02 ± 0.37	94.62 ± 0.43	
Preferred room [%]	62.68 ± 2.23	75.30 ± 2.30	83.77 ± 1.49	

0.5, $V_{\text{time},2} = W_{\text{time}} = 1$ and **Rooms** experiment weights $V_{\text{room},1} = 0.5$, $V_{\text{room},2} = W_{\text{room}} = 1$.

The experiment marked **Students + Time** equally combines student conflicts with time preferences, weights are $V_{\text{student},1} = V_{\text{time},1} = 0.25$, $V_{\text{student},2} = V_{\text{time},2} = W_{\text{student}} = W_{\text{time}} = 1$.

The next experiment (marked **Students + Time + Rooms**) most closely corresponds to reality. Here all the soft preferences are considered. Student conflicts and time preferences are weighted equally, room preferences are considered much less important. Weights of student conflicts and time preferences are the same as in the previous experiment (marked **Students + Time**). Moreover, the weights on room preferences are $V_{\text{room},2} = W_{\text{room}} = 0.2$. Note that rooms are not considered in the first level of the value selection criteria.

Finally, the last experiment (marked **No CBS**) presents average solutions obtained from the solver without conflict-based statistics. The weights on soft constraints are the same as in the previous experiment. But there is $V_{\text{conf},1} = 1$ (weight of a hard conflict) instead of $V_{\text{wconf},1} = 1$ (weight of a hard conflict weighted by CBS). $V_{\text{wconf},1}$ is set to zero. The solver was not able to find a complete solution within the given 30 minute time limit, not even when 2% random walk selection was used ($P_{\text{rw}} = 0.02$) to avoid cycling. Furthermore, there were at least 5 unassigned classes after 3 hours of running time.

Figure 3 compares several experiments giving different stress on student conflicts and time preferences. Average values from the best solutions of 10 different runs found within 30 minute time limit are presented. Only student conflicts or time preferences are considered in the border experiments marked **students** or **time** respectively. In the middle (experiment marked **1:1**), student conflicts

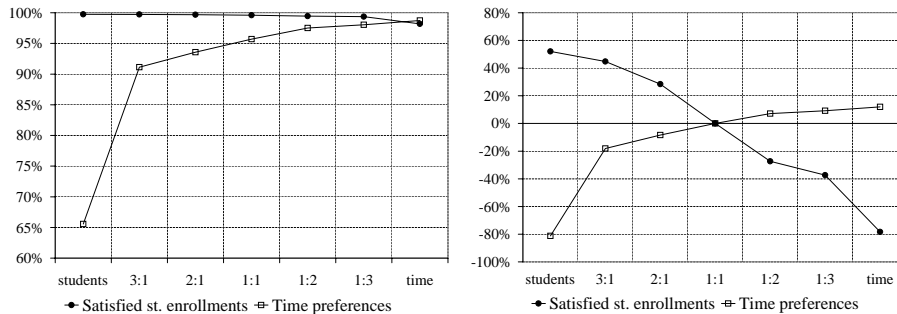


Figure 3. Comparison of satisfied student enrollments and time preferences: average quality of the solution (left), improvement of the solution in terms of percentage of the 1:1 solution (right).

and time preferences are equally weighted. The experiment marked **3:1** prefers student conflicts three times as much as time preferences (i.e., weights of student conflicts are three times higher than weights of time preferences) and vice versa. For instance, the experiment marked **1:2** has the following weights: $V_{wconf,1} = 1$, $V_{student,1} = 0.2$, $V_{time,1} = 0.4$, $V_{student,2} = W_{student} = 1$, $V_{time,2} = W_{time} = 2$.

7.2 Minimal Perturbation Problem

The following experiments were conducted on one of the complete initial solutions computed in the previous set of experiments (column marked **Students + Time + Room** in Table 1). Input perturbations were generated such that a given number of randomly selected variables were not allowed to retain the values they were assigned in the initial solution. Therefore, these classes cannot be scheduled to the same placement as in the initial solution (either room or starting time must be different). Only variables with more than one value in their domains were used. For each number of input perturbations, 10 different sets of input perturbations (i.e., variables with initial values prohibited) were generated. The following figures show the average parameter values of the best solutions found within 10 minutes.

The aim of the first set of experiments is to find a suitable setting for P_{init} (probability of selection of an initial value) and $V_{\Delta_{init},1..3}$ (difference in the number of perturbations in value selection). In each experiment, we have executed 10 tests for each of 10, 20, 30, ... 100 input perturbations respectively (100 runs in total). The average number of assigned variables together with the average number of additional perturbations are presented in Table 2. One or a combination of the criteria is used in each experiment. The second column refers to the set of criteria described in Table 3. Let us look at the explanation of this table. For instance, the expression $0.25_{s=1}, 1.0_{s=2}$ in the column marked $V_{students,s}$ means that $V_{students,1}$ is set to 0.25 and $V_{students,2}$ is set to 1. The first case

Table 2. Comparison of several approaches to MPP.

Test case		Assigned	Number of
P_{init}	Δ_{init}	variables [%]	perturbations
0.5	0	100.00	13.83
0.6	0	99.98	13.48
0.7	0	99.96	13.33
0.8	0	99.95	12.94
0	2	100.00	31.40
0.6	2	99.99	13.26
0	1	100.00	13.70
0.6	1	100.00	11.90

Table 3. Meaning of Δ_{init} .

Δ_{init}	$V_{\Delta_{\text{init}},i}$	$V_{\text{student},s}$	$V_{\text{time},t}$	$V_{\text{room},r}$
0	–	$0.25_{s=1}, 1.0_{s=2}$	$0.25_{t=1}, 1.0_{t=2}$	$0.2_{r=2}$
1	$0.5_{i=1}$	$1.0_{s=2}$	$1.0_{t=2}$	$0.2_{r=2}$
2	$1.0_{i=2}$	$0.25_{s=1}, 1.0_{s=3}$	$0.25_{t=1}, 1.0_{t=3}$	$0.2_{r=3}$

($\Delta_{\text{init}}=0$) corresponds to the settings of the **Students + Time + Room** experiment. In remaining Δ_{init} sets, we tried to decrease the importance of other value selection criteria in comparison with the initial value delta. For $\Delta_{\text{init}}=1$, the first level value selection criterion $V_{\Delta_{\text{init}},1}$ is used and the other optimization criteria which were placed in the first level are disabled ($V_{\text{student},1}, V_{\text{time},1}$ are set to zero). And the third line ($\Delta_{\text{init}}=2$) corresponds to a case when the second level value selection criterion $V_{\Delta_{\text{init}},2}$ is used and the other optimization criteria from the second level ($V_{\text{student},2}, V_{\text{time},2}, V_{\text{room},2}$) are moved to the third level.

Let us discuss particular experiments from Table 2. In the first four experiments (marked $P_{\text{init}} = 0.5, \dots, P_{\text{init}} = 0.8$), the minimal perturbation problem was solved only by changing the value selection criteria so that it selected the initial value with a given probability (50%, 60%, 70% and 80% respectively). Otherwise, it worked exactly as **Students + Time + Room** experiment, since all the other weights were the same. As the P_{init} probability is rising, we can see that the average number of additional perturbations is descending, but the algorithm is losing the ability to find a complete solution in every run (in the given 10 minute time limit).

Similarly, we can see that using just the second level value selection criterion $V_{\Delta_{\text{init}},2}$ is able to find a complete solution all the time, but the average number of additional perturbations is too high. A combination with the 60% probability

of an initial value selection helps to improve the average number of additional perturbations, but again, there were some cases where a complete solution was not found.

Using the first level value selection criterion $V_{\Delta_{init},1}$ seems to be very promising. All the presented experiments with this criterion were able to find a complete solution. Moreover, the experiment marked $P_{init} = 0.6$, $\Delta_{init} = 1$ (combining $V_{\Delta_{init},1}$ with 60% initial value selection probability) gave us the best results from the above experiments, since the average number of additional perturbations was the smallest. The following results (see Figures 4 and 5) were computed using the weights from this experiment.

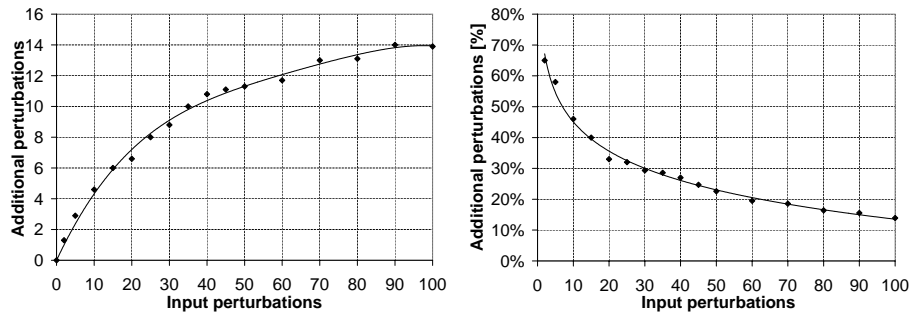


Figure 4. Absolute number of average additional perturbations (left) and average additional perturbations in terms of percentage of the number of input perturbations (right).

Figure 4 presents the average number of additional perturbations (variables that were not assigned their initial values though not prohibited). Additional perturbations are presented wrt. the absolute number of input perturbation (i.e., up to about 13.4% of input perturbations is considered). The best solution found within 10 minutes from each experiment is taken into account. The number of additional perturbations grows with the number of input perturbations.

The graph on the left of Figure 5 shows the average quality of the resulting solutions in the same manner as presented in Table 1. Because the initial solution is (at least locally) optimal, and because the number of additional perturbations is the primary minimization criterion, it is not surprising that the quality of the solution declines with an increasing number of input perturbations. The weighting between time preferences, student conflicts, and other parameters considered in the optimization can have a similar influence as seen in the initial solutions.

Finally, the graph on the right of Figure 5 presents the average time needed to find the best solution. Note that a 10 minute time limit for finding a best solution was set. The influence of this limit is seen mostly on the right portion of the chart, where the number of input perturbations exceed 50.

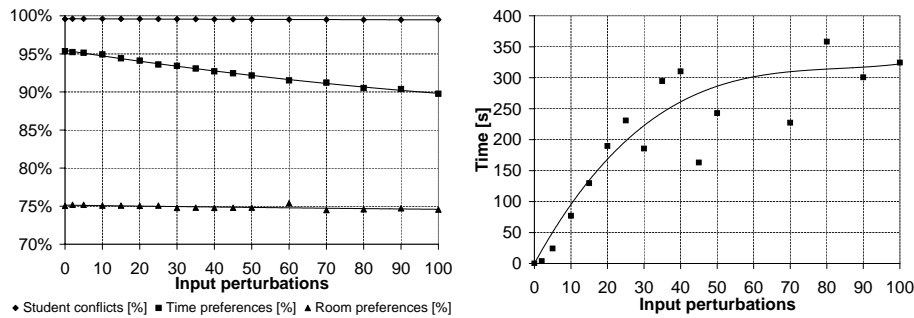


Figure 5. Average solution quality (left), average time (right).

Other Problems. Some MPP results of a preliminary version of the above described iterative forward search algorithm (e.g., without conflict-based statistics) on the *random placement problem* (see <http://www.fi.muni.cz/~hanka/rpp/> for details) are presented in our earlier work [1]. Here, it is compared with an algorithm combining branch-and-bound approach and the limited assignment number (LAN) search algorithm [2]. In this comparison, the iterative forward search algorithm was significantly better in its computational speed and in the number of additional perturbations than the other algorithm.

8 Conclusions

We have proposed and implemented a solution to a large scale university timetabling problem. Our proposal includes a new iterative forward search algorithm that is extended by conflict-based statistics which we believe can be generalized to other search algorithms. Both ideas combined together suffice to solve the problem and the role of additional heuristics can be minimized. Our problem solver is able to construct a demand-driven timetable as well as incorporate dynamic aspects. The initial solution generated by our solver satisfies the course requests of more than 99% of students together with about 95% of time requirements. The automated search was able to find suitable times and classrooms for all classes. The experiments with a MPP give us very promising results as well. Within 10 minutes, the solver was able to find a complete, high quality solution with a small number of additional perturbations.

Our future research will include extensions of the proposed general algorithm together with improvements to the implemented solver. We would like to do an extensive study of the proposed Minimal Perturbation Problem solver and its possible application to other, non timetabling problems. We are also planning to compare our results with the previous CLP solver [15] we have implemented. We are currently extending the CLP solver with some of the features included here to present a fair comparison.

We are also working on extensions to the implemented solver to cover additional requirements and problem features required by Purdue University. The strategy for computing perturbations needs to be extended as well. For example, a change in time is usually much worse than a movement to a different classroom. The number of enrolled/involved students should also be taken into account. Another factor is whether the solution has already been published or not.

Acknowledgments. This work is partially supported by Purdue University, by the Czech Science Foundation under the contract No. 201/04/1102 and by the Ministry of Education of the Czech Republic under the research intent No. 0021622419. Our thanks also go to the Supercomputer Center Brno where experiments with the search algorithms were conducted. We would like to thank our students for their assistance in solving this problem and Purdue staff who have helped in many ways. Our thanks also go to Keith Murray for his careful proofreading of the drafts of this paper.

References

- [1] Roman Barták, Tomáš Müller, and Hana Rudová. A new approach to modeling and solving minimal perturbation problems. In *Recent Advances in Constraints*, pages 233–249. Springer Verlag LNAI 3010, 2004.
- [2] Roman Barták and Hana Rudová. Limited assignments: A new cutoff strategy for incomplete depth-first search. In *Applied Computing*, pages 388–392. ACM, 2005.
- [3] Hadrien Cambazard, Fabien Demazeau, Narendra Jussien, and Philippe David. Interactively solving school timetabling problems using extensions of constraint programming. In Edmund K. Burke and Michael Trick, editors, *PATAT 2004 — Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling*, pages 107–124, 2004.
- [4] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
- [5] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.
- [6] Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Solving dynamic timetabling problems as dynamic resource constrained project scheduling problems using new constraint programming tools. In Edmund Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling, Selected Revised Papers*, pages 39–59. Springer-Verlag LNCS 2740, 2003.
- [7] Christelle Guéret, Narendra Jussien, Patrice Boizumault, and Christian Prins. Building university timetables using constraint logic programming. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, pages 130–145. Springer-Verlag LNCS 1153, 1996.
- [8] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.
- [9] Waldemar Kocjan. Dynamic scheduling: State of the art report. Technical Report T2002:28, SICS, 2002.
- [10] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

- [11] Ian Miguel. *Dynamic Flexible Constraint Satisfaction and its Application to AI Planning*. Springer, 2004.
- [12] Tomáš Müller and Roman Barták. Interactive timetabling: Concepts, techniques, and practical results. In Edmund Burke and Patrick De Causmaecker, editors, *PATAT 2002 — Proceedings of the 4th international conference on the Practice And Theory of Automated Timetabling*, pages 58–72, 2002.
- [13] Sylvain Piechowiak, Jingxua Ma, and René Mandiau. EDT-2004: An open interactive timetabling tool. In Edmund K. Burke and Michael Trick, editors, *PATAT 2004 — Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling*, pages 305–321, 2004.
- [14] Yongping Ran, Nico Roos, and Jaap van den Herik. Approaches to find a near-minimal change solution for dynamic CSPs. In *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 373–387, 2002.
- [15] Hana Rudová and Keith Murray. University course timetabling with soft constraints. In Edmund Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling, Selected Revised Papers*, pages 310–328. Springer-Verlag LNCS 2740, 2003.
- [16] Hani El Sakkout and Mark Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *CONSTRAINTS*, 4(5):359–388, 2000.
- [17] Gérard Verfaillie and Narendra Jussien. Dynamic constraint solving, 2003. A tutorial including commented bibliography presented at CP 2003. See <http://www.emn.fr/x-info/jussien/CP03tutorial/>.
- [18] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, 2005. To appear.