Minimal Perturbation Problem in Course Timetabling

Tomáš Müller[1] and Hana Rudová[2]

[1] Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, Prague, Czech Republic
muller@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. The 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 solution of an initial problem. 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. The methods proposed 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 [8, 14, 13, 10], however, are subject to change. Additional input requirements produce a new problem derived from the original. The dynamics of such a problem may require changes during the solution process, or even after a solution is generated. In many real situations, it is necessary to change the solution process such that 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 a 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 formulation. 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. This may create an avalanche reaction.

Our work is motivated by the class timetabling problem at Purdue University [12]. 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 [3] which are frequently applied to solve timetabling problems [6, 12, 10]. Dynamic constraint satisfaction [8, 14] is able to cover dynamic aspects in the problem. The minimal perturbation problem as defined in [1, 13], allows us to express our desire to keep changes to the solution (perturbations) as small as possible.

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" [13]. Dynamic changes in context of timetabling problems has started to be studied at [5]. A survey of existing approaches to dynamic scheduling can be found in [8]. In the annotated bibliography on dynamic constraint solving [14], it is notable that only four papers were devoted to the problem of minimal changes. The minimal perturbation problem was described formally in [13] 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 was proposed in [11].

In this paper, we introduce a new iterative forward search algorithm to solve the minimal perturbation problem. It is based on earlier work on solving methods for the static (initial) problem [10]. The generality of the method allows solving the initial problem. The basic difference in application is that optimization of the number of changes (perturbations) is not included while solving the initial problem. Our algorithm is close to local search methods [9]; 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 were successfully applied in earlier works [4, 7]. 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 heuristics proposed also do not limit the number of memorized conflicts and assignments. We have extended our search algorithm using these conflict-based statistics, but this is a general strategy that could be applied in other problem solvers.

The following section of this paper presents a description of the timetabling problem that motivates our work. Section 3 describes the iterative forward search algorithm. The subsequent section concentrates on conflict-based heuristics and their inclusion in the search algorithm is defined. The solution of our class timetabling problem is discussed in Sec. 5. Short summary of implementation together with experimental results for the minimal perturbation and initial problem 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 a half 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 750 classes having a high density of interaction that must fit within 41 lecture rooms with capacities up to 474 students. Course demands of almost 29,000 students out of a total enrollment of 38,000 must also be considered. Based on course demands, we must consider about 20,000 constraints between two classes to be taught at different times.

The timetable maps classes (students, instructors) to meeting locations and times. Information from initial enrollment to courses can be expected before any timetable is generated. Final enrollment is processed once the locations and times are published. 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 x 3 day per week classes meet on Monday, Wednesday, and Friday at the half hour. 1.5 hour x 2 day per week classes meet on Tuesday and Thursday during set time blocks. 2 or 3 hours x 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 interesting in the problem solution as it allows easier changes between classes having same or similar meeting patterns.

Another important constraint on the problem solution is instructor availability and instructor time preferences. Room availability is a also a major constraint for Purdue. In addition to room capacity, it was necessary to consider specific equipment needs and the suitability of the room's location.

Another aspect of the timetabling problem that must be considered is the need to perform student sectioning. Most of the classes in the large lecture problem (about 75 %) correspond to single-section courses. Here we have exact information about all students who wish to attend a specific class. The remaining courses are divided into multiple sections. In this case, it is necessary to divide the students enrolled to each course into sections that will constitute the classes.

Currently the timetable for Purdue University is constructed by a manual process. We have proposed an automated timetabling system to solve the initial problem [12]. This solution was based on constraint logic programming (CLP) with soft constraints. The CLP solver is currently under comparison with a new solver described in this paper.

3 Iterative Forward Search Algorithm

In this section, an iterative forward search algorithm is presented. It is based on local search methods [9]. 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 particular 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 algorithm). During each step, an unassigned or assigned variable is initially 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 (whatever 'best' means), 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 selected variable.

The algorithm attempts to move from one (partial) feasible solution to another via repetitive assignment of a selected value to a selected variable. During this search, the feasibility of all hard constraints in each iteration step is en-

```
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);
        if variable ∈ current.Unassigned then
            current.Assigned = current.Assigned ∪ {variable};
            current.Unassigned = current.Unassigned\{variable};
        Conflicting = FIND_CONFLICTING_VARIABLES(current);
        current.Unassigned = current.Unassigned ∪ Conflicting;
        current.Assigned = current.Unassigned\Conflicting;
        ASSIGN(variable,value);
        if better(current, best) then
            best = current
    return best
```

Fig. 1. Pseudo-code of the search algorithm.

forced by removing the conflicting variables. The search is terminated when the requested 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. Assignment of a value to a variable can cause conflicts with other variables however. For example, if there is a hard *all different* constraint for variables A, B and C, and variable A is assigned the value 3 while variable B, together with the value 3, is selected during the following step. The value of A becomes unassigned during the assignment $B = 3$. In our algorithm, the function FIND_CONFLICTING_VARIABLES computes the set of conflicting variables. These variables are unassigned in the next 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 maximal number of iterations or some other 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. For example, the solver can stop when all variables are assigned and less than 10% of soft constraints are violated. Termination of the process by the user can also be a part of the termination condition.

Solution Comparator The solution comparator compares two solutions: the current solution and the best solution found. We are looking for a solution with smaller number of unassigned variables, and in case when these numbers are equal, the solution with less violated soft constraints should be selected. This comparison can be based on several criteria. For example, it can lexicographically order solutions according to the criteria: number of unassigned variables (smaller number is better) or 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 for the second criteria. So, we are looking for a solution with smaller number of unassigned variables together with smaller number of violated soft constraints.

Variable Selection As mentioned above, the algorithm presented 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 [3]. 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.

We split variable selection criterion into two cases. If some variables remain unassigned, the worst variable among them is selected, i.e., first-fail principle is applied. This may, for example, be the variable with the smallest domain or with the highest number of hard and/or soft constraints.

The second case occurs when all variables are assigned. Because the algorithm does not need to stop when a complete feasible solution is found, the variable selection criterion for such case has to be considered as well. Here all variables are assigned but the solution is not good enough, e.g., in the sense of violated soft constraints. We choose a variable whose change of a value can introduce the best improvement of the solution. It may, for example, be a variable whose value violates the highest number of soft constraints.

It is possible for the solution to become incomplete again after such an iteration because a value which is not consistent with all hard constraints can be selected in the value selection criterion. This can also be taken into account in the variable selection heuristics.

To avoid cycling and to improve the search, this variable selection criterion can be randomized. There are several methods [9] which can be applied, e.g.:

- a random walk technique (with the given probability p a random variable is selected),
- not the worst variable, but a random selection of worse enough variable (e.g., from the top N worst variables), or

– a selection of a variable according to a probability based on the above mentioned criteria (e.g., roulette wheel selection).

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 [3]. Typically, the most useful advice is to select the best-fit value. So, we are looking for a value which is most preferred for the variable and also which causes the least trouble. 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 is hidden in the value selection (it roughly corresponds to a forward checking method [3]).

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

To avoid cycling, there are several methods [9] how to randomize the value selection procedure. For example, it is possible to select the N best values for the variable and choose one of them randomly. Or, it is possible to select a set of values so that the heuristic evaluation for the worst value in this group is maximally p percent higher than the heuristic evaluation of the best value (when smaller value means better value). Again, the value is selected randomly from this group. This second rule inhibits randomness if there is a single very good value.

3.1 Minimal Perturbation Problem

Let us first describe the meaning of perturbation in our approach. The changed problem differs from the initial problem by input perturbations. An input perturbation means that a variable must have different values in the initial and changed problem because of some input changes (e.g., course must be scheduled at different time in the changed problem). The solution to the minimal perturbation problem (MPP) [1, 13] can be evaluated by the number of additional perturbations. They are given by subtraction of the final number of perturbations and the number of input perturbations. An alternative approach is to consider variables in initial and new problem which were assigned differently [11, 1]. As before, we need to minimize the number of such differently assigned variables.

Despite the local search nature of the algorithm, there are some adjustments needed to be able to effectively solve the MPP. The task 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 which prefer the previous assignments (but not all the time, to avoid cycling).

For example, value selection heuristics can be adopted to select an initial value (if it exists) randomly with a probability P (it can be rather high, e.g., between 50-90%). If the initial value is not selected, original value selection can be executed. 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 as well. Otherwise, a value can be selected randomly from the constructed set of best-fit values. A disadvantage of such selection is that the probability P 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 amount of additional perturbations it will be hard to find a feasible solution.

Another approach is to limit the number of additional perturbations during the search. Furthermore, like in branch and bound, such a limit can be decreased when a feasible complete solution with the given number of perturbations is found. For example, if the number of additional perturbations is equal to or greater than the limit, the initial value has to be selected. Otherwise, if the number of additional perturbations is below the limit, the original value selection strategy is followed. The number of additional perturbations can also include variables that are not assigned yet whose initial values create a hard conflict with the current assignments.

The above approaches can also be combined together, which can help to divide their influence during the search.

Variable selection heuristics can also be adopted to help find a solution with a small number of perturbations. For example, when all variables are assigned, a variable that has an initial value but is not assigned should be selected, e.g., randomly among all variables that have no initial value assigned, and that participates in the highest number of violated soft constraints.

4 Conflict-based Statistics

In this section, a very promising extension of the search algorithm is presented. The idea behind it is to memorize conflicts and prohibit their potential repetition. When a value, v_0, is assigned to a variable, V_0, hard conflicts with previously assigned variables (e.g., $V_1 = v_1$, $V_2 = v_2$, ... $V_m = v_m$) can occur. These variables V_1,...V_m have to be unassigned before value v_0 is assigned to variable V_0. These unassignments, together with the reason for their unassignment (e.g., 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 an assignment again, the stored information about repetition of past hard conflicts can be taken into account, e.g., in the value selection heuristics. For example, if the variable V_0 is selected for an assignment again (e.g., because it became unassigned as a result of later assignments), we can weight the number of hard conflicts created in the past for each possible value of the 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 $V_1 = v_1$.

Moreover, such statistics can also be used in the opposite direction. If variable V_2 is selected for assignment, besides informing us how many times each value of this variable conflicted with previously assigned variables in the past, it also tells how many times values of other variables caused the unassignment of the value being considered for the selected variable. For example, if for a particular value of the variable V_2, we already know that a later assignment of $V_0 = v_0$ will cause V_2 to be unassigned, we can try to minimize such future conflicts by selecting a different value of the variable V_2 while V_0 is unassigned.

Conflict-based statistics is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency (e.g., that assignment $V_0 = v_0$ caused c_1 times an unassignment of $V_1 = v_1$, c_2 times of $V_2 = v_2$ ... and c_3 times of $V_m = v_m$).

In our implementation, there is a structure for each assignment stating how many times an assignment came in conflict with subsequent assignments and was therefore unassigned.

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

The above example of this structure expresses that variable A lost its assignment 3 times because of later assignments of value a to variable B, 4 times because B is later assigned a value c, etc.

This structure is being used in the value selection heuristics to evaluate existing and potential conflicts with the assigned and unassigned variables respectively. For example, if there is a variable B selected and if the value a is in conflict with an assignment $A = a$, we know that a similar problem has already occurred $3\times$ in the past, and the conflict $A = a$ is weighted with this number (the weight is actually incremented by one, not to ignore new, not yet rated conflicts). Similarly, if there is a variable A selected and if the variable B is not yet assigned, we know that its potential assignment $B = a$ already caused $3\times$ an unassignment $A = a$ and the assignment $B = c$ caused such problem $4\times$. So, there is a chance that if a value a or c is selected to be assigned to the variable B later on, it can cause (e.g., together with some other assignments) an unassignment of $A = a$. We weight these potential conflicts also with negative weight expressing how many other conflicts will cause these potential assignments ($B = a$ and $B = c$ in our example). So, for example if $B = c$ is already bad enough (it is in many other hard conflicts with assigned variables), value c will probably not be selected for variable B later on and we do not need bother with it too much.

Stated in another way, this approach helps the value selection heuristics to select a value that might cause more conflicts than another value, but these conflicts occurred less often, and therefore they have a lower weighted sum. This can help the search a lot to get out of a local minimum. It also tries to minimize future conflicts.

Moreover, these conflict-based statistics can be easily extended to allow some aging of the older conflicts. There is an *aging coefficient* introduced which multiplies all counters after each iteration step (for example, it is approximately 0.9993 for a weight of a single conflict to drop to 1/2 after 1000 iterations). The conflict-based statistics' weighted counters are then

$$\begin{aligned} \Delta_{\text{iteration}}(\text{conflict}) &= \text{iteration}_{\text{current}} - \text{iteration}_{\text{when the conflict occurred}} \\ \text{counter}_{\text{weighted}} &= \textstyle\sum_{\text{conflict occurrences}} w \cdot \text{aging coef}^{\Delta_{\text{iteration}}(\text{conflict})} \end{aligned}$$

where w is a weight of a single conflict and the sum goes over all conflicts in the counter multiplying each conflict weight with the aging coefficient to the power of how many iterations have passed since that conflict.

The implementation of the aging mechanism is rather simple. The weighted counter needs to be updated only when a new conflict occurs. Besides a counter weight, we need to memorize the iteration number, when the latest conflict occurred (and the weighted counter was updated). The current conflict weight is then the memorized weighted counter aged by the number of iterations which have passed since its last update

$$\begin{aligned} \Delta_{\text{iteration}} &= \text{iteration}_{\text{current}} - \text{iteration}_{\text{latest counter update}} \\ \text{counter}_{\text{weighted}} &= \text{aging coef}^{\Delta_{\text{iteration}}} \end{aligned}$$

and a weighted counter incrementent is then, similarly

$$\begin{aligned} \Delta_{\text{iteration}} &= \text{iteration}_{\text{current}} - \text{iteration}_{\text{latest counter update}} \\ \text{counter}_{\text{weighted}} &= \text{counter}_{\text{weighted}} \cdot \text{aging coef}^{\Delta_{\text{iteration}}} + w \\ \text{iteration}_{\text{latest counter update}} &= \text{iteration}_{\text{current}} \end{aligned}$$

5 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 modelling part will be described first, followed by a description of the algorithm.

5.1 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).

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 preference for time, room, building and the room's available equipment. Only placements with valid times and rooms are present in a class's 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 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 can not 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 between -2 (strongly preferred) and 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 courses 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).

5.2 Search Algorithm

In Sec. 3, we have described four functions which parameterize the algorithm proposed. Here we will describe their exact settings in our timetabling solver.

The termination condition stops the search when the solution is complete and good enough (expressed as number of perturbations and the sum of violated soft time and room preferences and the total number of student conflicts). 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, total number of student conflicts, etc., are visible to the user during the search.

The solution comparator prefers a more complete solution (with 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 with fewer violations of time and classroom soft preferences combined with student conflicts is selected.

The variable selection is based on a weighted sum (weights can be defined by the user) of several criteria:

- variable domain size (number of possible placements),
- number of previous assignments,
- number of group constraints in which the variable participates,
- if the variable has initial placement, number of hard conflicts on its initial placement (i.e., number of variables which become unassigned when the initial value is selected).

It also allows for definition of the random walk probability and the choice between roulette wheel and worst variable selection. When all variables are assigned, the variable with the worst evaluation is selected. Such evaluation is given for each variable by the weighted sum of value ordering criteria for optimization (see below). This variable promises the best improvement in optimization.

We have implemented hierarchical handling of value section criteria. There are three levels of comparison. In each level a weighted sum of the criteria described below are computed. Only solutions with the smallest sum are considered in the next level. The weights express how quickly a complete solutions should be found (e.g., 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. In the third level, some minor criteria are considered. These include not using rooms that are too large (having more than 50% excess seats) and minimizing the number of unscheduled half-hours in rooms between classes. Such half-hours cannot be used since all events require at least one hour.

These 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 between the importance of individual criteria. Furthermore, the value selection heuristics also support some limits, e.g., that all values with a first level sum smaller than a given percentage above the best value (typically 20%) will go to the second level comparison and so on. This allows for the continued feasibility of a value which is not the best, but is only slightly worse then the best, yet may be much better in the next level of comparison. If there is more than one solution after these three levels of comparison, one is selected randomly.

The value selection heuristics also allow for random selection of a value with a given probability (typically 2%, random walk) and for MPP to select the initial value (if it exists) with a given probability (e.g., 70%) and to limit the number of additional perturbations.

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

1. number of hard conflicts, weighted by their previous occurrences (see conflict-based statistics section);
2. number of potential conflicts, weighted by their previous occurrences (see conflict-based statistics section);
3. number of assignments (how many times the value has already been assigned to the variable).

Other criteria allows to achieve better results during optimization:

4. time preference delta: time preference of the value decreased by time preferences of the values assigned to variables, which have hard conflicts with the value (they must be unassigned, if the value is selected);
5. initial value delta: 1 if the value is initial, 0 otherwise, decreased by the number of initial values assigned to variables with hard conflicts with the value (they must be unassigned, if the value is selected);
6. number of student conflicts caused by the value if it is assigned to the variable;
7. number of hard student conflicts – same as the number of student conflicts. However, only conflicts between single section courses are counted (for details see Sec. 5.3);
8. preferences of satisfied or violated soft group constraints caused by the value if it is assigned to the variable;
9. 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);
10. soft time preference caused by a value if it is assigned to the variable;
11. delta of unused half-hours: number of empty half-hour time segments between classes that arise, minus those which disappear if the value is selected;
12. classroom is too big: 1 if the selected classroom has more than 50% more seats than the number of the students enrolled in the class.

Let us emphasize that these criteria are needed for optimization only, i.e., they are not needed to find a feasible[3] solution. Even more, assigning different weight to particular criteria allows to influence value of the corresponding objective function (see Fig. 3 with comparison for criteria 4 and 7).

5.3 Student scheduling

Many courses at Purdue University consist of several sections, with students enrolled in the course divided among them. Sections are often associated together

[3] Feasible solution must satisfy hard constraints.

by some constraints. For example, sections of the same course should not overlap. Each such section forms one class which has its own preferences. Therefore each is treated separately – there is a variable for each section.

An initial sectioning of students into course sections is processed. This student sectioning is based on Carter's [2] homogeneous sectioning and it is intended to minimize future student conflicts. However, there is still a possibility of improving the solution with respect to the number of student conflicts. This can be achieved via section changes during the search.

In the current implementation, sectioning is altered only by switching student enrollments between two different sections of the same course. Each student enrollment in a course with more than one section is processed. An attempt is made to switch it with a student enrollment from a different section. If this switch decreases the total number of student conflicts, it is applied.

We have compared two possibilities for switching these student enrollments. The first possibility is during the search, after a course is placed in the timetable. If a class is part of a course with multiple sections, an attempt is made to switch students with other sections of the course. Also, when a course has only one section, the system tries to move some students in multi-section courses who have a conflict with this class.

The second possibility, which appears to be much faster, but with similar results, is to switch students only when the best solution is found. In this case, the students are switched in the current solution, before it is stored as the best solution. All classes are processed and attempted switches are made between students in the same course. Note that a switch of a student enrollment can be followed with subsequent switches, so that classes can be processed more than once.

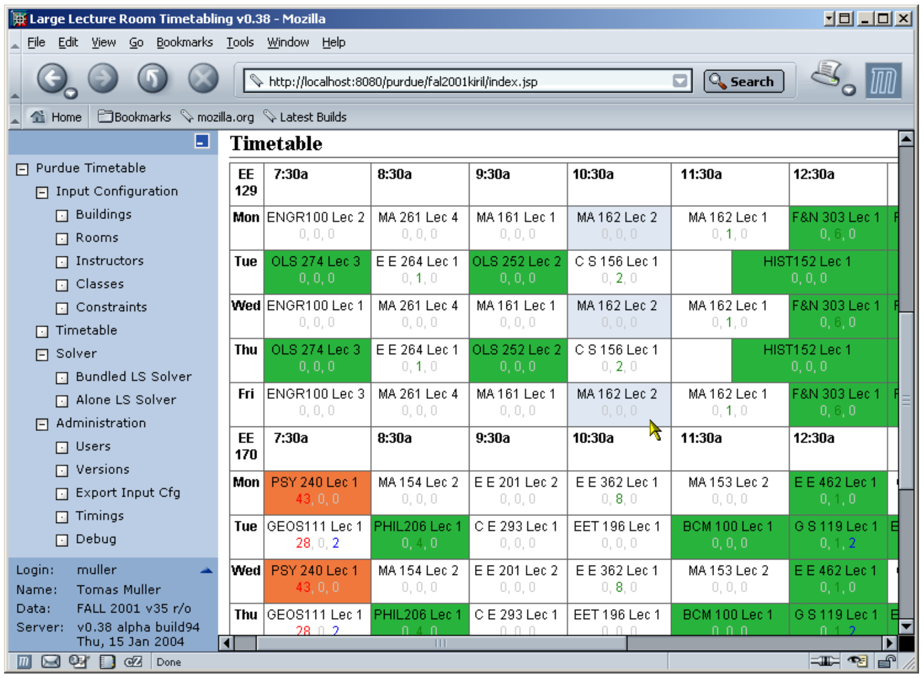
In the case where student enrollments are not switched after each iteration, counting student enrollments in the solution comparator and value selection criterion can be misleading (student conflicts, which can be eliminated by switching student enrollments later on do not matter). Therefore, "hard" student conflicts are counted as well. These consist only of student conflicts that occur between two single-section courses.

In the following experiments, the results using the second presented approach are discussed and compared with a case when no switching of student enrollments is used. Concerning comparison of the presented approaches in this section, the resultant numbers of violated student enrollments are the very same, but the first approach is more than ten times slower.

6 Implementation and Experiments

The timetabling system is implemented in Java. It contains a general implementation of the iterative 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.



Login: muller
Name: Tomas Muller
Data: FALL 2001 v35 r/o
Server: v0.38 alpha build94
Thu, 15 Jan 2004

Timetable

EE 129	7:30a	8:30a	9:30a	10:30a	11:30a	12:30a
Mon	ENGR100 Lec 2 0, 0, 0	MA 261 Lec 4 0, 0, 0	MA 161 Lec 1 0, 0, 0	MA 162 Lec 2 0, 0, 0	MA 162 Lec 1 0, 1, 0	F&N 303 Lec 1 0, 6, 0
Tue	OLS 274 Lec 3 0, 0, 0	E E 264 Lec 1 0, 1, 0	OLS 252 Lec 2 0, 0, 0	C S 156 Lec 1 0, 2, 0		HIST152 Lec 1 0, 0, 0
Wed	ENGR100 Lec 1 0, 0, 0	MA 261 Lec 4 0, 0, 0	MA 161 Lec 1 0, 0, 0	MA 162 Lec 2 0, 0, 0	MA 162 Lec 1 0, 1, 0	F&N 303 Lec 1 0, 6, 0
Thu	OLS 274 Lec 3 0, 0, 0	E E 264 Lec 1 0, 1, 0	OLS 252 Lec 2 0, 0, 0	C S 156 Lec 1 0, 2, 0		HIST152 Lec 1 0, 0, 0
Fri	ENGR100 Lec 3 0, 0, 0	MA 261 Lec 4 0, 0, 0	MA 161 Lec 1 0, 0, 0	MA 162 Lec 2 0, 0, 0	MA 162 Lec 1 0, 1, 0	F&N 303 Lec 1 0, 6, 0
EE 170	**7:30a**	**8:30a**	**9:30a**	**10:30a**	**11:30a**	**12:30a**
Mon	PSY 240 Lec 1 43, 0, 0	MA 154 Lec 2 0, 0, 0	E E 201 Lec 2 0, 0, 0	E E 362 Lec 1 0, 8, 0	MA 153 Lec 2 0, 0, 0	E E 462 Lec 1 0, 1, 0
Tue	GEOS111 Lec 1 28, 0, 2	PHIL206 Lec 1 0, 4, 0	C E 293 Lec 1 0, 0, 0	EET 196 Lec 1 0, 0, 0	BCM 100 Lec 1 0, 0, 0	G S 119 Lec 1 0, 1, 2
Wed	PSY 240 Lec 1 43, 0, 0	MA 154 Lec 2 0, 0, 0	E E 201 Lec 2 0, 0, 0	E E 362 Lec 1 0, 8, 0	MA 153 Lec 2 0, 0, 0	E E 462 Lec 1 0, 1, 0
Thu	GEOS111 Lec 1 28, 0, 2	PHIL206 Lec 1 0, 4, 0	C E 293 Lec 1 0, 0, 0	EET 196 Lec 1 0, 0, 0	BCM 100 Lec 1 0, 0, 0	G S 119 Lec 1 0, 1, 2

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

The following tests were performed on the complete Fall 2001 data set[4], including 747 classes to be placed in 41 classrooms. The classes included represent 81,328 course requirements for 28,994 students. The results presented here were computed on 1GHz Pentium III PC running Windows 2000, with 512 MB RAM and J2SDK 1.4.2.

Below, we present two types of experiments: experiments finding an initial solution (e.g., when all requirements are placed in the system), followed by experiments on the minimal perturbation problem (e.g., where there is an existing solution plus a set of changes to be applied to it). Solving an initial problem can

[4] Since the system has recently started to be used, we are expecting to have some results on the Fall 2004 data set soon.

be seen as a special case of MPP where all variables are new and therefore have no initial values.

6.1 Initial Problem

Figure 3 shows the computational results from 6 independent tests. *Time* refers

Test case	I.	II.	III.	IV.	V.	VI.
Time [min]	53.80	17.96	25.11	56.06	23.23	160.19
Student conflicts [%]	0.50	0.44	0.35	2.29	1.71	0.51
Preferred time [%]	95.67	88.72	44.61	90.50	46.31	48.09
Preferred room [%]	55.63	42.38	52.98	52.32	62.25	49.01
Useless half-hours [%]	7.25	7.89	8.53	7.57	6.12	6.44
Too big room [%]	23.16	18.47	22.89	22.89	22.76	23.96

(a) Conflict-based statistics (CBS) on (b) CBS off

Fig. 3. Solutions of the initial problem

to the amount of time required by the solver to find the presented solution. *Student conflicts* gives the percentage of unsatisfied requirements for courses chosen by students. *Preferred time* and *preferred room* estimate the satisfaction of time and room preferences respectively. The presented percentages are stating the ratios between sums of achieved and the best potential preferences over all the classes. *Useless half-hours* gives the percentage of unused half-hour segments that do not adjoin other segments (and are therefore unschedulable) to the total number of unused half-hour segments in the solution. Finally, *too big room* estimates the percentage of classes placed in rooms whose capacity exceeds the number of enrolled students by more than 50 percent.

A complete solution was found on every attempt for solutions in Fig. 3(a) using conflict-based statistics. The best solution found in each run within 60 minutes is presented. The very first complete solution is typically found in between 5 and 20 minutes.

The first three solutions (marked I., II. and III.) use move students between sections, different emphasis is placed on time preferences (value selection criterion 4 in Sec. 5.2) and student conflicts (value selection criterion 7 in Sec. 5.2) in each. Solution I. places heavier weight on faculty time preferences while solution III. focuses on minimizing student conflicts. The fourth and fifth solutions (marked IV. and V.) present results computed without moving students between sections. Different weightings of faculty time preferences and student conflicts were used in solutions IV. and V. as well. Solution V. attempts to minimize the number of student conflicts without shifts between sections.

A result achieved without using the conflict-based statistics approach is presented in Figure 3(b). The solver was not able to find a complete solution within 180 minutes in this case. The column marked VI. presents the best solution found within this amount of time. Four variables remained unassigned.

In all test cases, room preferences were considered much less important than time preferences and student conflicts. Unused half-hours (i.e., empty half hours in rooms with classes placed both before and after) and placements in rooms that are too large (i.e., rooms with 50% more seats than required for a particular class) are supplementary criteria.

In our second experiment, we would like to show that the general ideas of iterative forward search algorithm together with conflict-based statistics suffice to solve our timetabling problem. The role of heuristics can be minimized, special variable ordering is not needed, and value ordering heuristics needs to be defined for optimization only. Results from this experiment are presented in Fig. 4. All

Test cases	A	B
Time [min]	36.72 ± 12.68	3.51 ± 0.89
Student conflicts [%]	0.60 ± 0.07	2.31 ± 0.44
Preferred time [%]	91.72 ± 0.93	3.90 ± 3.32
Preferred room [%]	56.69 ± 5.14	23.84 ± 5.71
Useless half-hours [%]	6.67 ± 1.4	6.02 ± 1.31
Too big room [%]	23.29 ± 0.50	23.41 ± 0.49

Fig. 4. Solution of the initial problem with random variable ordering.

test cases use only random variable ordering (unassigned variable was randomly selected; if it does not exist any variable is randomly selected). Each column includes the best achieved solutions of 10 independent runs found within 60 minute time limit. The average value together with the root mean-square values are presented. Test cases in the column A uses the same value ordering as above (see Sec. 5.2). Since any variable can be selected several times while preserving partial results, the importance of variable ordering seems to decrease in our algorithm and random variable ordering is sufficient. Test cases in B uses value ordering criteria 1 only. The average results for the first complete found solution is presented here. Value ordering criteria needed for optimization were not processed which results in poor optimization results. However, a complete solution was found in all test cases much faster.

6.2 Minimal Perturbation Problem

The following experiments were conducted on one of the complete initial solutions computed in the previous set of experiments (column II. of Figure 3). 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 can not 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

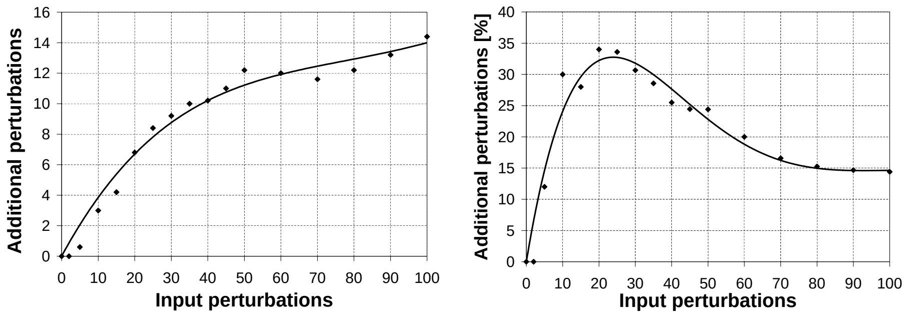


Fig. 5. Absolute number of average additional perturbations (left) and average additional perturbations in terms of percentage of the number of input perturbations (right).

value of input perturbations tested, five 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 30 minutes.

Figure 5 presents the average number of additional perturbations (variables that were: not assigned to their initial values, i.e., either room or starting time is different, though their initial values are not prohibited as for initial perturbations). 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 30 minutes from each test is taken into account. The number of additional perturbations grows with the number of input perturbations. The highest proportion of additional perturbations occurs between 10 and 35 input perturbations.

The graph in the upper left of Figure 6 shows the average quality of the resulting solutions in the same manner as presented in Figures 3. The lower average quality of solutions, as a percentage of the initial solution results, is illustrated by the graphs in the upper right and lower left (detail for the percentage between -6 and 10) of Figure 6. Because the initial solution is (at least locally) optimal, and because the number of additional perturbations is the pri-

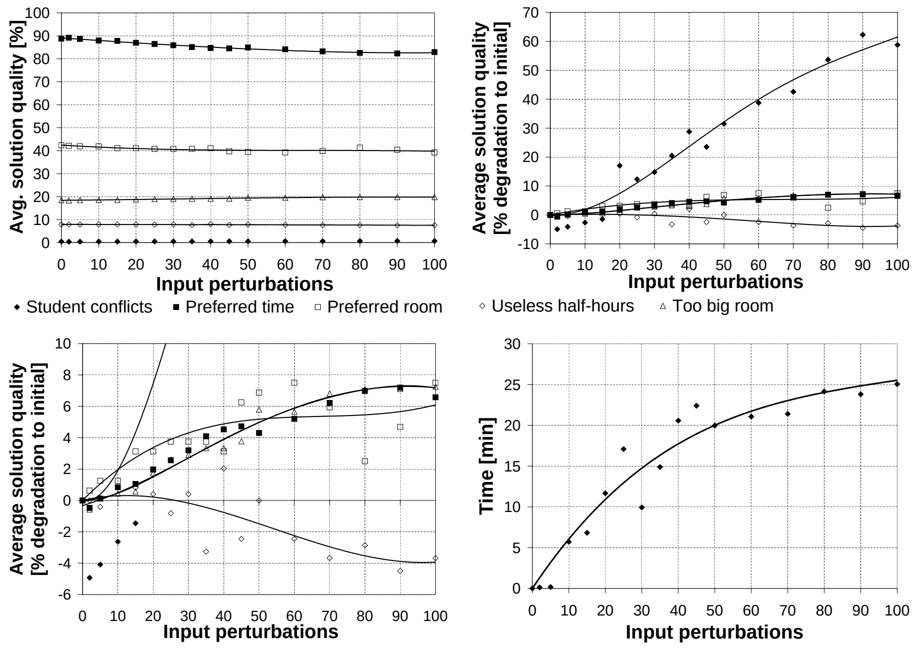


Fig. 6. Average solution quality (top left), resulting in lower average quality of the solution in terms of percentage of the initial solution (top right and bottom left in detail), average time (bottom right).

mary minimization criteria, it is not surprising that the quality of the solution declines with an increasing number of input perturbations. The parameter exhibiting the largest increase is the number of student conflicts. Note, however, that there are only 357 out of 81,328 course requirements in conflict in the input solution. 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 in the lower right of Figure 6 presents the average time needed to find the best solution. Note that a 30 minutes time limit for finding a best solution was set. The influence of his limit is seen mostly the right portion of the chart, where the number of input perturbations exceed 50.

7 Conclusions

We have proposed and implemented a solution to a large scale university timetabling problem. Our proposal includes a new iterative forward search algorithm. It 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. Our initial solution is able to satisfy the course requests of more than 99% of students together with about 90% of time requirements. The automated search was able to find suitable times and classrooms for all classes. The dynamic experiments give us very promising results as well. Within 30 minutes, the solver was able to find a complete, high quality solution with a slightly increased 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-based problems. Also, our approach must be validated using data sets from other semesters. We are also planning to compare our results with the former CLP solver [12] we have implemented. We are currently extending the CLP solver with some of the features included here (e.g., moving students between sections) to present a fair comparison.

Currently we are 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 of the 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.

The most interesting future direction in the development of the algorithm lies in its extension to constraint propagation. When there is a value assigned to a variable, such assignment can be propagated to unassigned variables to prohibit all values which come into conflict with the current assignments. The information about such prohibited values can be propagated as well. Since there is no backtracking, we need to memorize the reason why the value is prohibited (e.g. a no-good set containing an assignment with the prohibited value). When a variable becomes unassigned, the algorithm must be able to refresh all values which become prohibited as a result of the canceled assignment. Currently, we have implemented such a propagation corresponding to the forward checking method (i.e., there is no propagation over prohibited values). So far, it has not brought any improvements to the current algorithm, the resultant program was even slower. This is because the same work is already done in the value selection heuristics (but only for values of the desired variable). Extension of the implemented propagation technique and its possible conjunction with the conflict-based heuristics is planned for future investigation.

Acknowledgements This work is supported by Purdue University. We would like to thank our students for their assistance solving this problem and Purdue staff who have helped in many ways. Our thanks also to Keith Murray for his careful review and editing of the final drafts of this paper.

References

[1] Roman Barták, Tomáš Muller, 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] Michael W. Carter. A comprihensive course timetabling and student scheduling system at the University of Waterloo. In Edmund Burke and Wilhelm Erben, editors, *PATAT 2000 — Proceedings of the 3rd international conference on the Practice And Theory of Automated Timetabling*, pages 64–82, 2000.

[3] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[4] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[5] 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.

[6] 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.

[7] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[8] Waldemar Kocjan. Dynamic scheduling: State of the art report. Technical Report T2002:28, SICS, 2002.

[9] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[10] Tomáš Muller 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.

[11] 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.

[12] 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.

[13] Hani El Sakkout and Mark Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *CONSTRAINTS*, 4(5):359–388, 2000.

[14] 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/`.