

A local search heuristic for a university timetabling problem

Halvard Arntzen, Arne Løkketangen

halvard.arntzen@himolde.no, arne.lokketangen@himolde.no

Molde College, 6411 Molde, Norway.

1 Introduction

This note describes a heuristic algorithm designed for the 2003 *International Timetabling Competition*¹ organized by the Metaheuristic Network². The algorithm uses a simple adaptive memory search to improve the quality of an initial solution. The search is guided by tabu search mechanisms based on *recency* and *frequency* of certain attributes of previous moves.

2 Problem Formulation

The problem is to find a best possible (to be made precise after the definitions below) timetable within the following framework. There are 45 available *timeslots*,

$$T_1, T_2, \dots, T_{45},$$

nine for each day in a five day week. At any timeslot, there are N_R available rooms

$$R_1, R_2, \dots, R_{N_R}.$$

Each room has a given size, which is the maximum number of students the room can take.

We are given a set of N_E *events*

$$E_1, E_2, \dots, E_{N_E}.$$

Each event E_j has a set of students, S_j .

Finally there are a set of *room features*

$$F_1, F_2, \dots, F_{N_F}.$$

¹<http://www.idsia.ch/Files/ttcomp2002/>

²<http://www.metaheuristics.net/>

Each room *possess* some of these features, whereas each event *requires* some features.

The problem is to find a suitable room and timeslot for each of the given events. We will use the term *place* when we speak of a timeslot/room combination, i.e. a place is a pair

$$P = (T_k, R_j).$$

With this terminology, we seek to assign each event to a suitable place. We have a *feasible solution* to the problem if we have assigned all events to places, such that the following four *hard constraints* are satisfied:

- No student attends more than one event at the same timeslot.
- No event has more students than allowed in the chosen room.
- All features required by an event is possessed by the chosen room.
- No place has more than one event assigned to it.

To measure the quality of a feasible solution there are three *soft constraints*. Each violation of one of these gives a penalty point to the solution. The three possible violations are

- A student has a class in the last timeslot on a day.
- A student has more than two classes in a row.
- A student has a single class in a day.

To be precise about the second soft violation, if we have a solution where a student somewhere has three classes in a row we count this as *one* violation. If a student somewhere has four classes in a row we count this as *two* violations etc. We will refer to the three types of violations as *lastslot violations*, *serial violations*, and *once-in-a-day violations*. The *value* of a feasible solution is the total number of soft constraint violations in the solution. This of course means that a solution with a low value is a good one.

In the contest instances there are typically 10 rooms, hence there are 450 available places. There are typically 350-400 events, 5 -10 features and 200 - 300 students. In some instances all rooms are almost equal in size and in some others the sizes vary much, for example ranging from 3 to 43 in one instance.

The problem may now be precisely formulated: Given rooms, events, etc. as described above, *find the best possible feasible solution within a given time limit*.³

The rest of this note describes a heuristic algorithm designed to solve this problem.

³Approximately 11 minutes on a 2400Mhz Dell Optiplex GX 260

3 Algorithm overview

The algorithm consist basically of two parts. The first part builds a feasible solution from scratch. This part usually takes a very small amount of time to execute, in the order of 0.1% of the total available time. When a feasible solution is found, we employ a local search method to improve the solution quality by moving events around. Some basic mechanisms from the tabu search metaheuristic are employed to guide the search.

We now give a few definitions that are useful when we describe the details of the two parts. For a given event E we say that a place $P = (T, R)$ is a *possible place for E*, if E can be assigned to P without violating any hard constraints. For a given place P we say that an event E is a *liebhaber to P* if E can be assigned to P without violating any of the three first hard constraints. This of course means that P is a possible place for E if and only if E is a liebhaber to P and P is not occupied by another event. If an event E is a liebhaber to P and E is *not* assigned to another place, we call E a *free liebhaber to P*.

4 Constructive solver

The construction of an initial solution is carried out by performing the following steps.

1. Make a list L containing all unassigned events. (At first L contains all events)
2. Pick the event E in L with fewest possible places. If there is not a unique event having fewest possible places, choose randomly among the events with fewest possible places.
3. Find a place for E . This is done as follows. Let K be the list of all possible places for E . for each $P \in K$ let $q = (q_1, q_2, \dots, q_5)$ be defined as follows:

q_1 = The number of free liebhavers to P .

q_2 = The number of taken rooms at the same timeslot as P .

q_3 = The change in the number of lastslot violations created by assigning E to P .

q_4 = The change in the number of serial violations created by assigning E to P .

q_5 = The change in number of once-in-a-day violations created by assigning E to P .

One should note that q_5 may be negative, since assigning E to P may actually remove once-in-a-day violations. The other numbers q_j will always be nonnegative. For a vector $w = (w_1, w_2, \dots, w_5)$ of weights⁴ pick the place

⁴The weights are fixed through the process, and we used the values $w = (1, 1, 10, 20, 1)$ for the contest instances.

$P \in K$ that minimizes

$$wq = w_1q_1 + w_2q_2 + \cdots + w_5q_5.$$

If no unique P minimizes wq , a random selection is made from the minimizing places.

4. Assign E to the chosen place P . Update information about possible places and free liebhavers. Remove E from the list L . If L is not empty, return to step 2. If L is empty, we have a feasible solution.
5. If the process outlined in steps 1-4 fails, in the sense that at some point we have unassigned events E with *no possible place*, the process is restarted entirely from the start, but of course with another seed for the random numbers used.

The scheme outlined above usually finds a feasible solution without restarting the process.

5 Local Search

We now describe the local search used to improve upon the solution found by the constructive solver. We denote by s a feasible solution to the problem, that is, a complete timetable satisfying all the hard constraints. Let $V(s)$ be the value of s . We want to find an s with a low value $V(s)$. The basic plan for our method is as follows:

1. Start with some solution s .
2. Examine solutions s' in a neighborhood $N(s)$ of s .
3. Move to a good solution $s' \in N(s)$.
4. Set $s = s'$ and register if s is the best solution found so far.
5. Return to step 2.
6. If the process runs for a long time without making improvements, do something to get away from the current region of the solution space.

5.1 Moves and Move Neighborhood

Let s be a solution. A *move* is the operation of taking one event E and move it to one of its possible places P . The *move neighborhood* $N(s)$ of s consists of all solutions s' obtainable from s in *one* move. The *move value* is the change in solution value when moving from s to s' . Thus a move value = -1 means that the move improves the solution slightly.

5.2 Tabu status and Frequency measure for events

We implement a tabu criterion for events as follows. After an event E is moved to a place P in day d , ($d = 1, 2 \dots, 5$), we mark the event E as tabu, meaning that no move involving E is now allowed. The event E is tabu until a number T_E of *moves involving events localized in day d are executed*. The *tabu tenure* T_E assigned to an event is computed as follows. We have a basic tenure T_b . This is fixed for a large number of moves. Then we have a dynamic tenure T_d . This ranges between T_b and $2T_b$ with the largest value when the current solution value is far from the best found value so far in the search. We compute T_d as follows. Let v_0 be the value of the initial solution, v_c the current solution and v_b the current best value found. Let

$$T_p = \frac{30(v_c - v_b)}{v_0 - v_b + 1},$$

rounded down to integer value. Then let

$$T_d = \min(2T_b, T_b + T_p).$$

The exact tenure T_E assigned to an event E is $T_E = T_d + r$ where r is a random integer between 0 and 3. The basic tenure T_b can have values between 5 and 9, and it may change during the search.

We have implemented the tabu criterion with *aspiration* in the sense that a move which gives a *new best solution* can be made, even if the involved event is tabu at the time.

This tabu criterion prevents a moved event E from being moved again until a number of moves from the same day are made.

We also use a *frequency measure* to keep track of how frequently an event has been moved. This is done as follows.

Each event is assigned a real number F (frequency penalty). Initially this is 0 for all events. Each time event E is moved, we add a penalty p to F . Initially $p = 0.01$, and it increases with 1% for each move made. We use the numbers F as a secondary criterion for comparing moves, in the sense that if two moves have the same move value, we prefer the move involving the event with the lowest F . The reason for increasing p during the search is to enable the method to put more weight on the recent history of the search.

5.3 Move selection

At each iteration we select a move as follows.

1. For each event E find the place P giving the best move value. If several places give the same move value, chose randomly between the best places.

2. Make a list of all moves found in step 1. (One move for each event).
3. Sort the list using move value as primary criterion, and frequency measure as secondary criterion.
4. If the best move improves on the current best value found, choose this move unconditionally. Otherwise, find the best non-tabu move. Choose this move with probability 0.5. If it is not chosen, find the second best non-tabu move, and choose this with probability 0.5. Proceed in this manner until a move is chosen.

Note that there may be good moves not included in the list in step 2 and 3, since the list only holds one move per event.

5.4 Basic search

By the *basic search* we mean the process of selecting and performing moves iteratively. After each move, we update the tabu information and other relevant information. If the search has gone 4000 moves without improving the best value, we take alternative actions as described below.

5.5 Ejection chains

When a long sequence of moves have been performed without improving solution quality it is a good idea to invoke some alternative actions in order to move the search into other parts of the solution space. We choose to use what we call an *ejection chain* for this purpose. The ejection chains work as follows. Start from the *current solution*.

1. Pick a random place P_0 , that is not taken, and not in the last timeslot of a day.
2. Choose an event E_1 and move E_1 to P_0 . Now the place P_1 occupied by E_1 is free.
3. List all liebhavers of P_1 .
4. Choose an event E_2 from the list and move it to P_1 , leaving another free place P_2 .
5. Continue the process for a given number of steps.

We refer to the number of steps L as the *length* of the ejection chain. In each step we have a fixed place P , and we need to find an event E that is suitable for moving to P . The moves are chosen by the same criteria as in the basic search.

We also use a traditional tabu-mechanism for the ejection chain in the sense that an event is marked tabu for a number of steps when it is moved. In the ejection chains, we do not count iterations within days as we do in the basic search, and we use a fixed value of 10 for the tabu tenure. The main motivation for this is just "to do something else" when the basic search seems stuck in a bad area.

5.6 Control of the search

We now describe how the search process is guided.

1. Find an initial solution s . (Using the constructive solver).
2. Start the basic search from s , using basic tenure $T_b = 8$. Continue until 4000 moves are done without improvement.
3. Let L be a random integer between 50 and 200. Run an ejection chain of length L . Let s be the current solution after the ejection chain.
4. Reset (forget) all tabu information. Pick a random value from 5 to 9 for the tenure T_b . Run the basic search from s until 4000 moves are done without improvement.
5. Return to step 3.
6. The process is stopped according to the overall time limit.

6 A note on parameters of the method

The algorithm we use has many parameters, and time has not permitted any substantial testing for optimal parameter values. Also the parameters are the same for all instances, hence the algorithm does not adjust parameters to special properties of the various instances. On the other hand we change some parameters randomly during a search, partly because we suspect that the best parameter values are not the same for all instances. In particular this is done to the basic tabu tenure T_b , which changes every time a new basic search is started.

7 Performance of the algorithm.

We end with some reports on the results obtained for the 20 instances in the competition. We have tried 25 runs with different random seeds on each instance and submit the best solution found for each instance to the competition. The values mentioned below are for each instance the best value found in 25 runs.

At the point of writing the author has no idea of the overall performance compared to other algorithms, since this is written before any results from the competition are known. Also the author has no previous experience with timetabling, so we can not say whether the results look good or bad. Anyway, the lowest three values was found for instances 2, 18 and 20, where we have values 69, 87, and 53 respectively. The three hardest cases were instances 4, 5 and 12, where our best solutions have values 228, 225 and 290 respectively. Other values are spread evenly between the extremes. In a typical run, the algorithm finds the best value within 20% of the available time, and after that it is not able to improve the value further.

8 Implementation.

The code is written in C++. We have defined abstract data types like `Event`, `Place`, `Timeslot` etc. and we operate on objects of these types. To make the code more efficient, we use bitsets to hold some of the information we need to check up frequently. For instance, each event E has a bitset `AttStudents` flagging attending students, and each Timeslot T has a bitset `CurrStudents` flagging students currently assigned to a place in T . This gives a reasonably fast check of whether any student attending E is already present in the timeslot T . We have used the Standard Library class `bitset` for the specific implementation.