Interactive Timetabling: Concepts, Techniques, and Practical Results

Tomáš Müller[*], Roman Barták

Charles University
Department of Theoretical Computer Science
Malostranské náměstí 2/25, Praha 1, Czech Republic
{muller,bartak}@kti.mff.cuni.cz

Abstract. Timetabling is a typical application of constraint programming whose task is to allocate activities to slots in available resources respecting various constraints like precedence and capacity. In this paper we present a basic concept, a constraint model, and a solution algorithm for interactive timetabling. Interactive timetabling combines automated timetabling (the computer allocates the activities) with user interaction (the user can interfere with the process of timetabling). Because the user can see how the timetabling proceeds and he or she can intervene this process, we believe that such approach is more convenient than full automated timetabling, which behaves like a black-box. We also present some results achieved when applying our techniques to solving a real-life timetabling problem at the Faculty of Mathematics and Physics at Charles University.

Keywords: Educational Timetabling, Interactive Timetabling, Constraint Based Methods, Local Search, Tabu Search.

1 Introduction

Timetabling can be seen as a form of scheduling where the task is to allocate activities to available slots in resources respecting some constraints. Typically, the activity is described by its duration and by resources required to process the activity (conjunction and disjunction of resources can be used). There are also precedence constraints between the activities stating which activity must proceed before another activity and resource constraints stating when and how many activities can be processed by a given resource. In addition to the above hard constraints, that cannot be violated, there exist many preferences - soft constraints - describing the users' wishes about the timetable. Constraint programming (CP) is a natural tool for describing and solving such problems and there exist many timetabling systems based on CP [1, 2, 5, 11, 14].

In the current personal computing environment, an interactive behaviour of software is a feature requested by many users. Interactivity manifests itself in two directions: the user can observe what is doing inside the system (this is important for

[*] Supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942.

debugging but the final users like the animation of the problem solving process as well) and he or she can immediately influence the system. Interactive timetabling is a concept where the system presents to the user how the timetable is being built and the user can interfere with this process by changing the task during runtime. Naturally, this requires a special interactive solution algorithm that is building the timetable step by step and that can present a feasible (sub-) result at anytime (probably incomplete, but with no conflict). Moreover, such algorithm can be exposed to a task change, so it should not start re-scheduling from scratch after such a change but the new solution should be built on top of the former solution. By the task change we mean that the user can schedule the activity manually (and fix it in the slot) and vice versa (remove the activity from the slot to which it is currently allocated). Moreover, he/she can alter some other parameters like activity duration, precedence relation etc.

In this paper we define a generic timetabling problem motivated by school timetabling and we describe an interactive algorithm to solve this problem. We also show how the basic concept can be extended to solve a real-life university timetabling problem. It may seem that local search algorithms are best suited for this type of problems but note that typically local search is exploring the space of complete schedules which are not feasible and it reduces the number of violations. However, we require a feasible (perhaps incomplete) schedule to be presented to the user every time. On the other hand, backtracking based search, which is a method of extending feasible but incomplete solutions to a complete solution, does not support interactivity in the sense of changing the task during search. Therefore we propose a mixture of both approaches that borrows techniques both from backtracking based search (extending a partial solution to a complete solution and value and variable selection criteria) as well as from local search (tabu list to prevent cycling). In some sense, the presented algorithm mimics the behaviour of a human scheduler and therefore the animation of the scheduling/timetabling process looks naturally. We have implemented the proposed algorithm in Java as part of a complete timetabling system consisting of independent modules with a generic scheduling engine and with a graphical user interface (see Figure 1).

2 Motivation and Problem Description

A traditional scheduling problem is defined by a set of activities and by a set of resources to which the activities are allocated. Moreover, the activities must be positioned in time (otherwise, the problem is called resource allocation). Timetabling can be seen as a special form of the scheduling problem with slightly simplified view of time. In the timetabling problem, the time slots are defined uniformly for all the resources so the activities are being allocated to these slots (rather than to a particular time).

We took a school timetabling problem as the basic motivation of our research and we extracted the basic features to model timetabling problems in general. The basic object to be scheduled in a typical school timetabling problem is a **lecture**. The scheduled time interval, typically a week, is divided into the **time slots** with the same duration. For example, we have fifteen slots per day where every slot takes 45 minutes. Every lecture has assigned its duration, expressed as a number of the time

slots; this number is usually rather small - from one to three. Next, we have a set of resources, e.g. **classrooms** where the lectures take place, **teachers** who teach the lectures, **classes** for which the lectures are given, and other resources like overhead projector etc. Capacity of every resource is limited to one lecture. Every lecture has assigned a teacher, a class, a set of possible classrooms (one of them is selected), and a set of special resources (overhead projector etc.).

Finally, the relation between the lectures should be explicit, for example to express that two alternative lectures are taught in parallel or that the lesson is before the exercises. We call this relation a **dependency**.

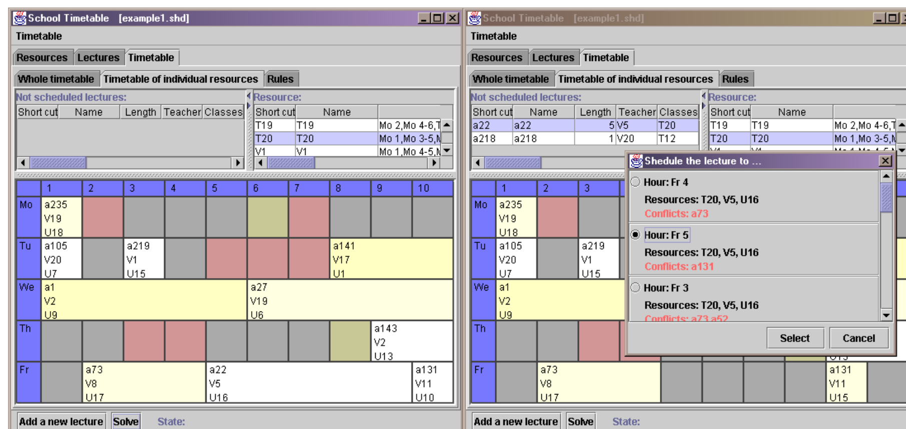

Fig. 1. In interactive timetabling, the user see continuous improvements of the schedule (left) and he or she can interfere with this process, e.g. by manual allocation of the activity (right).

In the following section, we propose a general model motivated by the above school timetabling problem. However, we believe that this model can be applied to other timetabling problems different from this particular school timetabling problem. Still, there is a space for other extensions of this model, like introduction of new preferences or dependencies.

3 The Model

We propose a generic model for timetabling problems consisting of a set of resources, a set of activities, and a set of dependencies between the activities. Time is divided into the time slots with the same duration. Every slot may have assigned a constraint, either hard or soft one: a hard constraint indicates that the slot is forbidden for any activity, a soft constraint indicates that the slot is not preferred. We call these constraints "time preferences". Every activity and every resource may have assigned a set of time preferences, which indicate forbidden and not preferred time slots.

Activity (which can be directly mapped to a lecture) is identified by its name. Every activity is described by its **duration** (expressed as a number of time slots), by **time preferences**, and by a **set of resources**. This set of resources determines which

resources are required by the activity. To model alternative as well as required resources, we divide the set of resources into several subsets – **resource groups**. Each group is either conjunctive or disjunctive: the conjunctive group of resources means that the activity needs all the resources from the group, the disjunctive group means that the activity needs exactly one of the resources (we can select among several alternatives). An example can be a lecture, which will take place in one of the possible classrooms and it will be taught for all of the selected classes. Note that we need not model conjunctive groups explicitly because we can use a set of disjunctive groups containing exactly one resource instead (the set of required resources can be described in a conjunctive normal form). However, usage of both conjunctive and disjunctive groups simplifies modelling for the users.

Resource is also identified by its **name** and it is fully described by **time preferences**. There is a hard restriction, that only one activity can use the resource at the same time. As we mentioned in the previous section, the resource represents a teacher, a class, a classroom, or another special resource at the school timetabling problem.

Finally, we need a mechanism for defining and handling direct **dependencies** between the activities. It seems sufficient to use binary dependencies only that define relationship between two activities. Currently, we use a selection of three temporal constraints: the activity finishes before another activity, the activity finishes exactly at the time when the second activity starts, and two activities run concurrently (they have the same start time). The scheduling engine provides an interface for introduction of other binary dependencies.

The solution of the problem defined by the above model is a timetable, where every scheduled activity has assigned its start time and a set of reserved resources, which are needed for its execution (the activity is allocated to respective slots of the reserved resources). This timetable must satisfy all the hard constraints, namely:

- every scheduled activity has all the required resources reserved, i.e., all resources from the conjunctive groups and one resource from each disjunctive group of resources,
- two scheduled activities cannot use the same resource at the same time,
- no activity is scheduled into a time slot where the activity or some of its reserved resources has a hard constraint in the time preferences,
- all dependencies between the scheduled activities must be satisfied.

Furthermore, we want to minimise the number of violated soft constraints in the time preferences of resources and activities. We do not formally express this objective function; these preferences will be used as a guide during search for the solution.

Last but not least, in interactive timetabling we want to present some solution to the user at each time. Therefore, we will work with partial solutions where some of the activities are not scheduled yet. Still, these partial solutions must satisfy the above constraints. Note that using a partial solution allows us to provide some reasonable result even in case of over-constrained problems.

3.1 A model extension for a real-life university timetabling problem

For solving a real-life timetable problem at the Faculty of Mathematics and Physics at Charles University, Prague we need to extend the set of constraints in the generic scheduling engine.

First, look at the traditional classroom capacity constraint. We are scheduling lectures and we know in advance how many students will attend the lectures (with the exception of alternative lectures, see below). Thus, the capacity restriction is naturally modelled by assigning only the classrooms with enough capacity to the lecture. Note that there are groups of alternative resources attached to each lecture, so the group describing the classrooms contains only the classrooms with enough capacity.

Second, we should be able to model alternative activities per resources (not only alternative resources per activity). In particular, we have several alternative lectures and several classes (groups of students) that should attend the lecture independently of which particular one. These alternative lectures can be taught simultaneously by different teachers in different classrooms or one teacher has these lectures at different time etc. We want to maximise the number of available alternatives for the students under the hard constraint that at least one alternative must be available for each class.

To describe alternative activities, we introduced a new entity into our model, **a group of alternative activities**. This group is assigned to some resources with the following constraints:

- each resource to which the group of alternative activities is assigned must have at least one free slot to which some activity from the group of alternative activities can be allocated (a hard constraint),
- the number of available alternative activities for all resources that has assigned a group of alternative activities should be maximised (modelled as a soft constraint).

Note that we are working with partial schedules as well, so the above hard constraint is relevant only to schedules where all the activities from the group of alternative activities are allocated (no look ahead there). Also notice that we are encoding the objective function using soft constraints, in particular this objective function is encoded in location selection heuristic that minimise the number of violations of soft constraints (see bellow).

The next group of hard constraints is derived from the organisation structure of the faculty. The lectures are taught in three different buildings at Prague so if there are two lectures taught in different buildings and sharing either a teacher or a class then there must be enough time to move between the buildings. In particular, there must be at least one time slot between the lectures or the lectures are taught in the same building. Moreover, we prefer the lectures to be taught in the same building where the teacher has his office. Finally, the number of crossovers between the buildings during the day should be minimised both for classes and for teachers.

The last group of constraints concerns the time preferences. Some subjects have more lectures per week. In such a case, the lectures must be scheduled to different days. Note that such a constraint can be easily described using the direct dependency between the lectures. Each class should not have more than ten teaching hours per day and more than six hours without a break. Similarly, each teacher should have neither more than eight hours per day nor more than six hours without a break. Finally, there

is preference to daytime. For example, the early morning hours or late evening hours are less preferred. Also Friday afternoon is not preferred. These preferences are described as a number (from least preferred -3 to most preferred 3) for each time slot in the week. We call this number a global time preference. The last time constraint, which is very weak, says that the number of breaks (free slots between the lectures per day) should be minimal both for teachers and for students.

Let us now summarise all the additional constraints. We can divide them into hard constraints that must be satisfied and soft constraints expressing the preferences. The additional hard constraints are:

- two (not alternative) lectures of the same subject cannot be taught at the same day,
- the capacity of each classroom cannot be exceeded,
- each class (that has to attend some lectures, which are alternative) must have possibility to attend at least one of the alternative lectures,
- there is at least one hour (slot) break between every two lectures which go one after another, and that share either the same teacher or the class and that are taught in different buildings.

The additional soft constraints are:

- one class should not have more than ten hours per day and should not have more than six hours without a break,
- one teacher should not have more than eight hours per day and should not have more than six hours without a break,
- the number of crossovers during a day for teachers and classes should be minimal,
- a lecture should be taught in the same building where the teacher has his or her workplace
- maximise the number of alternatives which students can attend,
- maximise the sum of used time slots multiplied by the global time preference of each slot,
- minimise the number of free slots between the first and the last lecture of the day for all teachers and classes.

4 An Interactive Solver

To satisfy needs of interactive timetabling we designed an ad-hoc algorithm for solving the above described timetabling problem. Nevertheless, we believe that this algorithm can be generalised to solve other interactive problems as well.

Let us first summarise the requirements to such interactive solution algorithm. The algorithm should provide some (partial) solution at each step. This (partial) solution must be feasible (all required constraints are satisfied) and it should be good enough in respect to satisfaction of soft constraints. It is not necessary to find the optimal solution (a minimal number of violated soft constraints), a good enough solution is OK. We insist more on interactivity of the algorithm that means two things: the difference between the solutions provided by the algorithm in two consecutive steps should be minimal and the user can interfere with the solution process. The user may

change parameters of activities and resources, add or remove dependencies, and allocate or detach any activity from the time slot. The algorithm must be able to start from such a modified timetable, make it feasible and continue in extending the partial solution to a complete solution.

There exist two basic approaches how to solve problems defined by means of constraints: backtracking based search with constraint propagation that extends a partial feasible solution to a complete solution and local search that decreases the number of violations in a complete solution. The idea of local search follows our requirement that there is some solution provided at each step and the difference between the solutions in consecutive steps is rather small. However, the provided solutions are not feasible. On the other side backtracking based search can provide a partial feasible solution at each step but it is not easy to add interactive behaviour to it. In particular, it does not support external changes of the partial solution and the difference between two consecutive solutions can be large after long backtrack.

To satisfy needs of an interactive solution algorithm we propose to mix features of above two approaches. Our algorithm uses forward based search that extends a partial feasible solution to a complete solution. The next solution is derived from the previous solution by allocating some activity and removing conflicting activities from the partial schedule. Every step is guided by a heuristic that combines minimal perturbation and minimal violation of soft constraints. If the user changes somehow the partial solution, the algorithm first removes all the violated activities and then it continues from a given feasible solution.

4.1 A basic concept of the interactive solver

We propose an interactive scheduling algorithm that works in iterations. The algorithm uses two basic data structures: a set of activities that are not scheduled and a partial feasible schedule. At each iteration step, the algorithm tries to improve the current partial schedule towards a complete schedule. The scheduling starts with an empty partial schedule (which is a feasible schedule), i.e. all the activities are in the list of non-scheduled activities. Then it goes repeatedly from one partial feasible solution to another feasible solution until all the activities are scheduled or the maximum number of iterations is reached.

The user may interrupt the algorithm after arbitrary iteration and he or she can acquire a solution (the latest one or the best solution ever found) where perhaps not all the activities are scheduled but all the constraints on the scheduled activities are satisfied. During this interruption, the user may allocate manually some of the non-scheduled activities, weaken some constraints or make other changes (add new activities etc.) and then let the automated scheduling continue.

Look now what is going on in the iteration step. First, the algorithm selects one non-scheduled activity and computes the locations where the activity can be allocated (locations with the hard constraint are not assumed). Every location is described by a start time (the number of the first slot for the activity) and by a set of required resources. Then every location is evaluated using a heuristic function over the current partial schedule. Finally, the activity is placed to the best location that may cause

some conflicts with already scheduled activities. Such conflicting activities are removed from the schedule and they are put back to the list of non-scheduled activities (see Figure 2).

```
procedure solve(unscheduled, schedule, max_iter)
  // unscheduled is a list of non-scheduled activities
  // schedule is a partial schedule (empty at the beginning)
      iterations=0;
      while unscheduled non empty & iterations<max_iter
            & non user interruption do
              iterations ++;
              activity = selectActivityToSchedule(unscheduled);
              unscheduled -= activity;
              location = selectPlaceToSchedule(schedule, activity);
              unscheduled += place(schedule, activity, location)
                 // place the activity and return violated activities
      end while;
      return schedule;
end solve
```

Fig. 2. A kernel of the interactive scheduling algorithm

The above algorithm schema is parameterised by two functions: activity selection and location selection. We can see these functions in a broader view of constraint programming as variable selection and value selection functions. Note that value/variable selection functions can be found both in backtracking based search as well as in local search algorithms and there exist some general techniques used to define these functions. Our view of these functions is somewhere in between backtracking and local search. We are selecting a non-scheduled activity (i.e., non-instantiated variables) and during this selection we use information about the previous locations of the activity. Remind that the activity might be removed from the schedule during some previous iteration step. Moreover, because of removing activities from the partial schedule we need some mechanism to prevent cycles. Techniques used for activity and location selection and for escaping from a cycle are described in next sections.

4.2 Activity selection (a variable selection criterion)

As mentioned above, the proposed algorithm requires a function that selects a non-scheduled activity to be placed in the partial schedule. This problem is equivalent to what is known as a variable selection criterion in constraint programming. There exist several general guidelines how to select a variable. In local search, usually a variable participating in the largest number of violations is selected first. In backtracking based algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is the most complicated is selected first. This could be the variable involved in the largest number of constraints (a static criterion) or the variable with the smallest domain (a dynamic criterion) etc. [3, 4, 7, 8, 13]

We follow the general variable selection rules and we recommend selecting the hardest to schedule activity first. For simplicity reasons, we call it the worst activity. The question is how to decide between two activities which one is the worse activity.

We can use both static and dynamic information when deciding about the worst activity. A number of dependencies in which the activity participates is an example of static information. More dependencies mean a worse activity. Dynamic information arises from the partial schedule, for example a number of locations in the timetable that don't conflict with any already scheduled activity. It is more expensive to acquire dynamic information but this information is usually more accurate than static information. Moreover, if static information is used alone then the solution algorithm can be easily trapped in a cycle.

In our algorithm we currently use the following criteria when selecting the activity:

- How many times was the activity removed from the timetable? ($N_{\#Rm}$)
- In how many dependencies does the activity participate? ($N_{\#Dep}$)
- In how many places can this activity be placed? ($N_{\#Plc}$)
  Note: Places, where another activity lays and can't be removed (the activity was fixed by the user), are not included.
- In how many places this activity can be placed with no conflict? ($N_{\#PlcNC}$)

A weighted-sum of above criteria is used as follows:

$$val_{activity} = -w_1 N_{\#Rm} - w_2 N_{\#Dep} + w_3 N_{\#Plc} + w_4 N_{\#PlcNC}$$

Activity with the minimal heuristic value is selected. Notice that the first two criteria are used with a negative weight, this is because a larger value means a worse activity. Using such formula gives us more flexibility when tuning the system for a particular problem. Moreover, it allows studying influence of a particular criterion to efficiency of the scheduling.

It is possible to select the worst activity among all non-scheduled activities but due to complexity of computing the heuristic value, this could be rather expensive. Therefore we propose to select a subset of non-scheduled activities randomly (we use a probability of selection 0.2) and then to choose the worst activity from this subset. The results will not be much worse and we can select the activity approximately five times faster [9].

4.3 Location selection (a value selection criterion)

After selecting the activity we need to find a location where to place it. This problem is usually called a value selection in constraint programming. Typically, the most useful advice is to select the best-fit place [3, 4, 7, 8, 13]. So, we are looking for a place, which is the most preferred for the activity and also where the activity causes less trouble. It means that we need to find a place with minimal potential future conflicts with other activities. Note that we are not using constraint propagation explicitly in our algorithm, the power of constraint propagation is hidden in the location selection (it roughly corresponds to a forward checking method).

To find the best place for the activity we explore the space of all possible start times and for each start time we explore the possible sets of resources that the activity needs for its execution. Recall that for a single start time we may have several sets of resources to which the activity can be allocated (there can be alternatives among the required resources). We evaluate each such location using the following criteria:

- How many scheduled activities will be in conflict with the current activity if it is placed to the selected location? ($N_{\#CnfAct}$)
- How many activities, that has been removed in any previous step by the current activity, are removed again? ($N_{\#rep}$)
  Note: For every activity we can register the activity, which caused its removal from the schedule in any previous iteration. By including these activities in the next decision we can avoid some short cycles.
- How many scheduled activities, which are in conflict with the location, can't be rescheduled without another conflict elsewhere? ($N_{\#ConflNoRsh}$)
- How many soft constraints are violated (both in activity and in selected resources time preferences)? ($N_{\#soft}$)
- How far is the location from the location where the activity was placed before? (N_{diffPl})
- How good is the location from the user point of view? (N_{user})
  Note: This is a user-defined preference for location used to model real-life problems (for example, preferring morning slots to evening slots).

Again, we use a weighted sum of the above criteria to evaluate the location:

$$val_{place} = w'_1 N_{\#CnfAct} + w'_2 N_{\#rep} + w'_3 N_{\#ConfNoRsh} + w'_4 N_{\#soft} + w'_5 N_{diffPl} + w'_6 N_{user}$$

Location with the minimal heuristic value is selected. To remove cycling, it is possible to randomise this location selection procedure. For example, it is possible to select five best locations for the activity and then choose one randomly. Or, it is possible to select a set of locations such that the heuristic value for the worst location in this group is maximally twice as large as the heuristic value of the best location. Again, the location is selected randomly from this group. This second rule inhibits randomness if there is a single very good location.

The above formula describes a generic heuristic for location selection. The additional constraints from our real-life timetabling problem (Section 3.1) appear in two roles. The new hard constraints are projected to the value $N_{\#CnfAct}$ expressing the number of conflicting activities. Also, a violation of the hard constraint may cause removal of some activity from the current partial schedule. Typically, the violated hard constraint determines the activity to be removed, e.g. two activities cannot share a single resource or the time dependency between the activities is violated. In case of groups of alternative activities, we require that for each resource to which this group is assigned, some alternative activity can be allocated. This hard constraint is tested when all the activities from the group of alternative activities are allocated. If the constraint is violated then one of the activities in the group is removed from the partial schedule, i.e. this activity must be re-scheduled. There is no direct look-ahead technique for this constraint; nevertheless, some form of a look-ahead technique is hidden in the soft constraint that requires maximisation of the number of available alternative activities for resource. So, let us now speak about the additional soft constraints that arose from our real-life problem.

The generic location selection heuristic contains a mechanism for user defined constraints, namely the value N_{user} expresses user's priorities. This value can be

computed from the new soft constraints using the combination of the following criteria:

- If the activity belongs to a set of alternative activities what is the number of resources, for which the activity is alternative and which can be used by this activity? ($N_{\#alt}$)
- What is the number of violations of the "maximum hours per day and the maximum hours without a break for teachers and classes" constraint? ($N_{\#mxH}$)
- What is the number of crossovers for teachers and classes (used by the activity)? ($N_{\#cross}$)
- How many free slots are between the first and the last lecture of the day to which we are allocating the current activity? ($N_{\#freeH}$)
- How many teachers (used by the activity) have his or her workplace in a building different from the building to which the activity is being allocated? ($N_{\#difB}$)
- What is the sum of global time preferences in the timeslots used by the activity? (W_{timePf})

As we mentioned above, these extended criteria are combined via a weighted sum to get the value of N_{user}:

$$N_{user} = -w'_7 N_{\#alt} + w'_8 N_{\#mxH} + w'_9 N_{\#cross} + w'_{10} N_{\#freeH} + w'_{11} N_{\#difB} - w'_{12} W_{timePf}$$

4.4 How to escape a cycle?

In the previous two sections about value and variable selection we proposed some mechanisms how to avoid cycling, but we can do more. In the current implementation of the scheduling engine we use a technique based on a tabu list [3, 6, 12].

Tabu list is a FIFO (first in first out) structure of pairs (variable, value) with a fixed length. When a value V is assigned to the variable X, a new pair (X,V) is added to the end of the tabu list and the first pair is removed from the tabu list. Now, we can avoid a repeated selection of the same value by applying a tabu rule which says: if the pair (X,V) is already in the tabu list then do not select the value V for X again (until the pair disappears from the tabu list). The tabu rule prevents short cycles (the cycle length corresponds to the length of the tabu list) and it can be broken only via so called aspiration criterion. If the aspiration criterion is satisfied for a pair (X,V) then V can be assigned to X even if the pair (X,V) is in the tabu list.

In our scheduling algorithm we adapted a slightly modified version of the tabu list. We save pairs (activity, location) in the tabu list. When we choose an activity A and before we compute a heuristic value for the location L we explore the tabu list. If the pair (A,L) appears twice in the tabu list then we do not use the location L for the activity A in this iteration. If the pair (A,L) appears exactly once in the tabu list then the location L is assumed only if this location has the minimal heuristic value (it is the best location). This is also the case when the pair (A,L) could be introduced to the tabu list for the second time (if the location L is selected). Finally, if the pair (A,L) is not in the tabu list then the location is processed as described in the previous section.

To summarise it, our aspiration criterion allows us to select some location for the activity maximally two times in the period defined by the length of the tabu list. The second assignment is allowed only if the location is the best for the activity.

5 Results

We have implemented the above described scheduling algorithm in Java and we have extended it by adding new constraints to solve a real-life timetabling problem in the Faculty of Mathematics and Physics. The paper [9] describes the results for the generic engine, we present here just the relation between the number of activities and time to allocate them. All tests were run under JDK 1.3.1 on a single processor PC with AMD Duron 750MHz, 256 MB RAM.

We have tested the generic engine using randomly generated timetable problems with the same number of classes, classrooms, and teachers and 15 slots per day (5 days). These benchmarks are available in [10]. In these tests, we used fix weights for all the criteria described in the sections about activity and location selection. The problems were generated in order to have the constant fullness of the timetable, so the number of classes, classrooms, and teachers changed according to the number of activities. The mean duration of activities was two timeslots. Extended constraints (used for solving the timetable at the Faculty of Mathematics and Physics) were disabled. There were 50 dependencies between the activities and 35% of all timeslots (for resources and activities) had a soft constraint (should not be used) and 5% of the timeslots had a hard constraint (cannot be used).

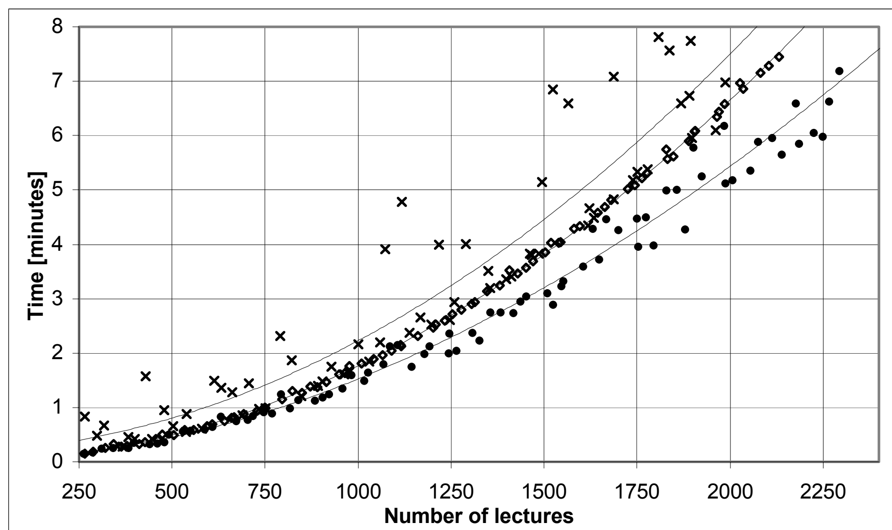

Fig. 3. Comparison of the time spent on solving the problem and the size of the problem expressed as a number of input lectures. Problems are randomly generated so that the mean fullness of each resource stays constant (× 85% fullness, ◊ 80% fullness, • 85 % fullness no hard constraint). There are also 50 time dependencies between the activities, 35% of the timeslots of each resource and activity have a soft constraint, 5% have a hard constraint.

Figure 3 shows a comparison between the time spent by the scheduler on solving these problems (100% activities scheduled) and the number of input lectures. The scheduler can solve fully the problems with 80% mean fullness of all resources, 85% fullness started to be critical (note, that we have some hard constraints here – 5% of all slots can not be used and there are also 50 time dependencies between the activities). When we used an input timetable without any such hard constraint we can get stable results up to 90% fullness (10% more). The scheduler seems not to be very limited by the size of the input problem. Nevertheless, the dependence on the number of activities is not linear because of the growing number of unscheduled activities and possible locations considered in heuristics.

The extended version of the engine was tested using real data from an autumn semester 2001 at the Faculty of Mathematics and Physics, Charles University (see Figure 4). The problem size and structure was as follows:

- 5 days per week, 15 time slots per day,
- 746 lectures, which have to be centrally scheduled (with mean duration 2.03 time slots, teaching hour = 45 minutes), in total 1512 timeslots,
- 349 classes or sub-classes (454 groups of classes),
- 479 teachers,
- 41 classrooms (but only 30 can be used),
- 3 different locations (buildings).

It takes approximately 8 to 10 minutes to solve the problem without any user intervention. Moreover, the system provides interactive capabilities, so the user can easily adjust the timetable, e.g. via the drag and drop technique. This way the user can guide the system or he or she can express some preferences that can be hardly encoded in the soft constraints. The following list shows some features of the timetable found by the system (with no user intervention):

- all activities were scheduled (and all hard constraints were satisfied),
- there were 76 crossovers for the classes and 7 crossovers for the teachers (crossover means change of building during a day),
- there were only 21 cases when a class has more then 10 hours a day and one case when a class had more than 6 hours without a break,
- there was no case when a teacher had more than 8 hours a day or more than 6 hours without a break,
- a class could attend on average 84% of alternative lectures announced for it,
- on average 84% of lectures were scheduled to the same building where the teacher has his office,
- 74% of lectures were scheduled between 3rd and 11th slot (from 9:00 till 16:25), on Fridays between 3rd and 6td slot (from 9:00 till 12:15);
  87% of lectures were scheduled between 2nd – 12th slot (from 8:10 till 17:15), on Fridays between 2nd – 7th slot (from 8:10 till 13:05);
  only 3% of lectures were scheduled on or after 14th slot (from 18:10) and on or after 10th slot (from 14:50) on Fridays

Fig. 4 The system generates compact timetables: a timetable for a class (left), a timetable for a classroom (right)

6 Conclusions and Future Work

We presented a promising algorithm for solving timetabling problems, which combines principles of the local search with other techniques for solving constraint satisfaction problems. The basic motivation was to design a generic algorithm with interactive features for solving school timetabling problems. However, we believe that the proposed principles can be applied to other constraint satisfaction problems especially when interactive behaviour is required. We have also showed that the generic algorithm can be easily extended to cover additional soft constraints. A practical application of the proposed algorithm was presented as well.

Currently, we are working on further empirical studies of this algorithm with a particular emphasis on studies how the weights influence efficiency. Further research is oriented both theoretically, to formalise the techniques and to put them in a wider context of constraint programming, and practically, to implement the engine as a system of co-operating parallel threads.

References

[1] S. Abdennadher and M. Marte. *University timetabling using constraint handling rules*. Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules, 1999.

[2] R. Barták. *Dynamic Constraint Models for Planning and Scheduling Problems*. In New Trends in Constraints, LNAI 1865, pp. 237-255, Springer, 2000.

[3] R. Barták. *On-line Guide to Constraint Programming*, http://kti.mff.cuni.cz/~bartak/constraints, 1998.

[4] D. Clements, J. Crawford, D. Joslin, G. Nemhauser, M. Puttlitz, M. Savelsbergh. *Heuristic optimization: A hybrid AI/OR approach.* In Workshop on Industrial Constraint-Directed Scheduling, 1997.

[5] J. M. Crawford. *An approach to resource constrained project scheduling*. In Artificial Intelligence and Manufacturing Research Workshop, 1996.

[6] P. Galinier and J.K. Hao. *Tabu search for maximal constraint satisfaction problems*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 196-208, Schloss Hagenberg, Austria, Springer, 1997.

[7] J.-M. Labat, L. Mynard. *Oscillation, Heuristic Ordering and Pruning in Neighborhood Search*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 506-518, Schloss Hagenberg, Austria, Springer, 1997.

[8] Z. Michalewicz, D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[9] T. Müller and R. Barták. *Interactive Timetabling*. In Proceedings of the ERCIM workshop on constraints, Prague, 2001.

[10] T. Müller. *Interactive Timetabling – Benchmarks*, http://kti.mff.cuni.cz/~muller/ttbench, 2002

[11] A. Schaerf. *A survey of automated timetabling.* Technical Report CS-R9567, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, 1996.

[12] A. Schaerf. *Tabu search techniques for large high-school timetabling problems*. In Proceedings of the Fourteenth National Conference on Artificial Intelligence, pp. 363-368, Portland, Oregon, USA,1996.

[13] E. Tsang and C. Voudouris. *Fast Local Search and Guided Local Search and Their Application to British Telecom's Workforce Scheduling Problem*. Technical Report CSM-246, Department of Computer Science, University of Essex, 1995.

[14] M. Wallace. *Applying constraints for scheduling*. In Constraint Programming, volume 131 of NATO ASI Series Advanced Science Institute Series, Springer, 1994.