A Generic Object-Oriented Constraint-Based Model for University Course Timetabling

Kyriakos Zervoudakis and Panagiotis Stamatopoulos

Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 157 84 Athens, Greece
{quasi, takis}@di.uoa.gr

Abstract. The construction of course timetables for academic institutions is a very difficult problem with a lot of constraints that have to be respected and a huge search space to be explored, even if the size of the problem input is not significantly large, due to the exponential number of the possible feasible timetables. On the other hand, the problem itself does not have a widely approved definition, since different variations of it are faced by different departments. However, there exists a set of entities and constraints among them which are common to every possible instantiation of the timetabling problem. In this paper, we present a model of this common core in terms of Ilog Solver, a constraint programming object-oriented C++ library, and we show the way this model may be extended to cover the needs of a specific academic unit.

1 Introduction

The construction of university timetables falls under the class of scheduling problems. Its difficulty [8], along with the fact that an educational institution has to tackle it once or twice a year, has drawn the attention of researchers from various fields, even as long ago as the 1960s [15]. Since then, the problem has been tackled with various approaches including graph coloring [23], network flows [6] and operations research methods [1]. During the last years, various artificial intelligence techniques have also been used against the problem, like tabu search [3], [30], simulated annealing [9], [27], genetic algorithms [24], [10] and constraint programming [16], [22], [19], [11], [26].

It is surprising that, even now, after all these years, not all educational institutions use automated tools to construct timetables. A survey regarding this issue in British universities [5] showed that only 21% of the universities used a computer for constructing examination timetables. 37% used a computer only for assisting the process and 42% did not use computers at all. Although these data concern examination timetabling, it seems that the numbers are similar for course timetabling case as well.

One might wonder about the reasons for this situation. In [2], it is mentioned that many institutions have automated solutions, but they are so tailored to each institution's case that they cannot be used by others. It is true that the differences between institutions are many, especially as far as quality criteria are concerned. That makes the use of one tool that was implemented for one university inappropriate for another. What is more, it is possible for a tool that was developed for one institution to be inapplicable for that same institution if some changes in the curricula occur.

In [2] the basic rules that govern timetables are also mentioned. In spite of the differences between institutions, some basic categories of rules can be recognized. The quality determination criteria can also be easily categorized and many similarities can be identified. For example, in all cases the same hard rules hold, such as that one teacher cannot give more than one lecture simultaneously or that one student cannot attend more than one lecture at a time. Most quality criteria have to do with the distribution of certain groups of lectures in time, like uniform distribution of the lectures in each day of the week, dense scheduling of these lectures for every day, upper bounds in the time that a teacher or a student can be occupied in teaching activities and so on.

Constraint programming is a problem solving methodology that allows the user to describe the data of a problem and the constraints that govern them without explicitly handling these constraints. This methodology fits perfectly to the timetabling problem, as someone might define the involved entities and express, in a declarative way, what is a legal and good timetable. On the other hand, it would be very good, from the software engineering point of view, if the modeling were to follow an object-oriented approach, as the obtained result would then be as general as we would like it to be at the same time as having unbounded opportunities to be specialized as much as we would like, in order to capture any specific timetabling situation. Fortunately, the combination of constraint programming and object-oriented design is provided by an existing commercial C++ library, the ILOG SOLVER [21], [20]. In this paper, a generic model for university course timetabling, which is based on ILOG SOLVER, is presented. The application of this model on the specific course timetabling problem faced by the Department of Informatics and Telecommunications of the University of Athens is also described.

2 University Course Timetabling

Most university timetabling problems are based on the basic student–course model [7]. Let $U = \{u_1, u_2, \ldots, u_{|U|}\}$ be the set of students, $S = \{s_1, s_2, \ldots, s_{|S|}\}$ be the set of courses taught, d_i be the number of teaching periods for course s_i, $S_j = \{s_{j_1}, s_{j_2}, \ldots, s_{j_{|S_j|}}\}$ be the set of courses student j wishes to attend, $T = \{t_1, t_2, \ldots, t_{|T|}\}$ be the set of teachers, $t : S \mapsto T$ a function mapping each course to the teacher that teaches it and $x_{i,k}, 1 \leq k \leq d_i, 1 \leq x_{i,k} \leq p$ the time that the kth teaching period of course s_i is given, with p the number of possible teaching periods of the institution (the maximum length of the timetable). A solution to the problem is any assignment to the variables $x_{i,k}$ such that the following constraints are respected:

– A teacher cannot give more than one course at a time

$$\sum_{i,k}[x_{i,k} = m][t_n = t(s_i)] \le 1, \quad \forall m, n; \tag{1}$$

– A student cannot attend more than one course at a time

$$\sum_{i,k}[x_{i,k} = m][s_i \in S_j] \le 1, \quad \forall j, m. \tag{2}$$

The above model gives complete freedom to the student to select the courses he/she takes and is close to the way most universities operate. If, in addition, a set of classrooms $C = \{c_1, c_2, \ldots, c_{|C|}\}$ is added, as well as a constraint

$$\sum_{i,k}[x_{i,k} = m][cl_{i,k} = c_n] \le 1, \quad \forall m, n \tag{3}$$

where $cl_{i,k}$ is the classroom where the kth teaching period of course s_i is conducted, then the model becomes even more realistic. Within the framework of this basic model, a number of more specialized, but common in practice, constraints can be expressed. For example, the unavailabilities of a teacher can be taken into account by modifying constraint (1):

$$\sum_{i,k}[x_{i,k} = m][t_n = t(s_i)] \le avail(t_n, m), \quad \forall m, n \tag{4}$$

where $avail(t_n, m)$ equals 1 if teacher t_n is available in period m, or 0 if not. Next, a short introduction to the Ilog Solver library is given and, then, an object-oriented constraint programming model of the timetabling problem, based on the student-course model, is presented.

3 Ilog Solver

Ilog Solver [21], [20] is a constraint programming object-oriented C++ library. Some information on Solver is necessary in order for the model that follows to be understood. The type `IlcInt` corresponds to the C++ `long` type. Integer constraint variables are represented by the class `IlcIntVar`. A command like `IlcIntVar x(m,0,10)` creates a constrained variable `x` with the integers from 0 to 10 included in its domain. `m` is an object of the class `IlcManager` to which all constrained variables and constraints between them are connected. The method `IlcIntVar::removeValue(IlcInt a)` removes value `a` from a variable's domain and the method `IlcIntVar::setValue(IlcInt a)` assigns value `a` to a variable. The class `IlcIntVarArray` implements an array of constrained integer variables. The call `IlcIntVarArray t(m,10)` creates an array of 10 constrained integer variables which can be accessed, as usually, with the overloaded operator `[]`. Constraints are represented by the class `IlcConstraint` and can be created with the overloaded C++ relational operators. If `x` and `y` are objects of the class

`IlcIntVar`, the call `IlcConstraint c(x>y)` creates a constraint `c` that ensures all values in `x`'s domain are greater than the minimum of `y`'s domain. In order for a constraint to be posted, the method `IlcManager::add()` has to be used. The above constraint, for example, can be posted with the call `m.add(c)`. Posting a constraint ensures that it will be considered by SOLVER's internal constraint propagation engine and that after any modification of a constrained variable, the constraint network will be modified, in order to be brought to a certain consistency degree. SOLVER uses a version of the AC-5 algorithm [28] to bring the constraint graph to an arc-consistent state after any such modification. Finally, SOLVER supports goal programming and gives the user the ability to define any search algorithm of choice, like depth-first search (DFS), limited-discrepancy search (LDS) and so on. We omit the details of the implementation in SOLVER terms of these algorithms, since they are not necessary for the scope of this paper.

4 Object-Oriented Modeling

The model for the university course timetabling problem we present in this section aims to be as generic as possible. However, some simplifying assumptions are commonplace in many universities. For example, the mathematical model presented in Section 2 considers only the number of teaching periods for each subject with no further constraints. For example, a four period subject can be split in two two-hour lectures, one three-hour lecture and one one-hour lecture and so on. It is usual for many universities for subjects to be partitioned in lectures in a preprocessing step and this model follows this assumption. Time is measured in multiples of a predetermined unit of time, further partitioning of which is of no practical importance. This unit can be an hour, half an hour or whatever. In the following, this unit will be referred to as a "teaching period". Also, there is a maximum number of such units within which the timetable has to be constructed. A timetable is constructed for a week with D days and H time units every day. The model, however, can be easily extended to cases where timetables span within more than one week.

The similarity in format of the equations in the student–course model is obvious. Teachers and classrooms are necessary resources for a lecture to take place and each resource cannot support more than one activity at a time. Thus, it is reasonable for the notion of resource to play a central role in the modeling of the problem. It can also be seen that although a student is not actually a resource for a lecture, the equations that define the constraints on students are of the same form, since lectures that are taken by one student cannot be conducted at the same time. In the original student–course model, every student is completely free to take the course he/she chooses. This is true for many universities where modularity is important. In other universities, a student can select from a set of predetermined course offers. In any case, the net effect is that either because of student preferences or because of predetermined management decisions, some sets of lectures are formed and it is demanded that no two lectures of the same set be conducted at the same time. Thus, these sets are modeled like other resources and will be called "lecture groups" from now on.

One more important issue is the criteria according to which the quality of a timetable is measured, since, in practice, this is an optimization problem. However, it would be very difficult to gather all such criteria and present a way of implementing them with the model which follows. We will try to display by example that the implementation of the most usual criteria is not only possible but, indeed, easy within this framework.

4.1 Subjects

```
class c_Subject {
protected:
  IlcIntVarArray StartVariables; };
```

The class `c_Subject` represents a subject. `StartVariables` is an array of constrained variables, each one representing the time in which a lecture of this subject is scheduled. This class is used for logistic purposes, but also for calculating preferences regarding the distance between lectures of the same subject.

4.2 Lectures

```
class c_Lecture {
protected:
  IlcInt Duration;
  IlcIntVar Start;
  IlcIntVar Classroom; };
```

The class `c_Lecture` represents a lecture. It contains the two decision variables for each lecture: the starting time of the lecture and the classroom in which it is conducted.

4.3 Unary Resources

```
class c_UnaryResource {
protected:
  IlcIntVarArray TimeTable;
  IlcIntVarArray LectureTimeUnits;
public:
  void add(IlcIntVar start, IlcInt duration); };
```

As mentioned before, the notion of a unary resource is of central importance in the modeling. The class `c_UnaryResource` represents a unary resource: that is, a resource that can only support one activity at a time. In order to declare that a lecture uses a certain resource (any activity can of course require more than one resource) the method `add` has to be called with parameter variables `Start` and `Duration` of the corresponding lecture:

```
void c_UnaryResource::add(IlcIntVar start, IlcInt duration) {
  for (IlcInt i=0; i<duration; i++)
    LectureTimeUnits[count++]=start+i; }
```

As can be seen above, for each time unit of a lecture's duration, one variable in array `LectureTimeUnits` is created. For example, if a resource supports three lectures out of which the first has duration 2 time units and starting time `[0..2 4]`, the second duration 1 and starting time `[8..9]` and the third duration 3 and starting time `[12]`, then the array `LectureTimeUnits` will have six elements (the sum of the lectures' durations) which will be `[0..2 4][1..3 5][8..9][12][13][14]`. The array `TimeTable` has `D*H` elements which correspond to the `D*H` time units of a timetable. Each one of them takes values within the interval `[-1..d]`, where `d` is the number of elements in the array `LectureTimeUnits`. When all lectures have been added, then the SOLVER function `inverse` is called. This function's declaration is

```
IlcConstraint IlcInverse(IlcIntVarArray f, IlcIntVarArray invf);
```

according to which if f's length is n and $invf$'s length is m then

- If $f[i] \in [0, m-1]$ then $invf[f[i]] == i$,
- If $invf[j] \in [0, n-1]$ then $f[invf[j]] == j$.

The above constraint guarantees that during each time unit the resource supports at most one activity.

4.4 Multiple Resources

```
class c_MultipleResource : public c_UnaryResource {
protected:
  IlcInt Multiplicity;
public:
  void add(IlcIntVar start, IlcIntVar classroom, IlcInt duration); };
```

In most cases, a unary resource can represent teachers and groups of lectures that cannot be given simultaneously, since an activity will either require such a resource or not. But this is not enough in the case of resources such as classrooms, since a lecture can be given in any one of a certain set of classrooms. This requirement is a disjunctive demand of a resource. In our case, all such resources, let us say classrooms, are modeled as one multiple resource with multiplicity equal to the number of the individual resources. The produced class was created with minor modifications on the existing unary resource class:

- The array `TimeTable` has `mult*D*H` elements instead of `D*H`, where `mult` is the multiplicity of the resource. Each `D*H`-tuple of this array corresponds to one of the `mult` individual resources.
- In order for a lecture to be added to a resource, one more parameter is needed along with the starting variable of the lecture and its duration, namely a constrained variable representing the resource to which the lecture will be eventually assigned. In the case of classrooms, this variable represents the classroom in which the lecture will take place.

– The add method is modified as follows:

```
void c_MultipleResource::add(IlcIntVar start,
                             IlcIntVar classroom, IlcInt duration) {
for (IlcInt i=0; i<duration; i++) {
  LectureTimeUnits[count++]=start+i+classroom*D*H; } }
```

As can be seen, the elements of the array LectureTimeUnits are like the ones in unary resources plus the term classroom*D*H which makes each of these elements point to the D*H-tuple of the multiple resource corresponding to the value of the variable classroom.

5 Application

We believe that the proposed model is general enough to cover most cases. Its representative potential will be exhibited through one real problem, that of constructing a course timetable for the Department of Informatics and Telecommunications of the University of Athens (DIT/UoA).

5.1 Curricula

The set of given courses and the teaching periods for each course are given. In case the course cannot (or is not desired to) be given in one lecture, then it is partitioned into two or more lectures of predefined duration. In this case, lectures of the same course cannot be given in the same day. Each lecture is taught by one or more teachers. If a course is given by another department, then it is scheduled by that department. It can be required for a lecture to be given in a specific classroom or in one of a subset of the available ones. In this case, a degree of preference between classrooms can be given. The undergraduate curriculum is organized over four years. Each course can be either obligatory for a certain year or belong to a certain group of lectures which are called directions. It is demanded that lectures of the same year which are either obligatory or belong to the same group not be given simultaneously. Also, there is the possibility for teachers to express certain personal constraints on the maximum number of teaching hours in a day and the maximum number of days in which they are occupied in teaching activities.

The above are hard constraints that have to be respected fully in order for a timetable to be feasible. Apart from these, there are also several criteria according to which the quality of a timetable is measured. It is desired that obligatory lectures or lectures of the same direction and year have as few gaps as possible between them during each day. It is also desired for such lectures to be as uniformly distributed during the timetabling period as possible, and it is desired for lectures of the same course to have a reasonable distance in days between them.

5.2 Formulation

Let $L = \{l_1, l_2, \ldots, l_{|L|}\}$ be the set of lectures, $T = \{t_1, t_2, \ldots, t_{|T|}\}$ the set of teachers, $C = \{c_1, c_2, \ldots, c_{|C|}\}$ the set of classrooms and $S = \{s_1, s_2, \ldots, s_{|S|}\}$ the set of courses. There exists a function $sub : L \mapsto S$ which maps every lecture to the corresponding course. Let D be the number of teaching days in a week and H the number of periods in every day. Let d_i be the duration of lecture l_i. We define $x_{i,1} = x_i$, $x_{i,j} = x_{i,j-1} + 1, j = 2, \ldots, d_i$. Let $t(l_i) \subseteq T$ the teachers of lecture l_i and $G = \{g_1, g_2, \ldots, g_{|G|}\}$, $g_i \subseteq L$, the set of lecture groups. The problem is to find the time and classroom for each lecture, so a solution is any mapping $sol : L \mapsto \{0, 1, \ldots, D * H - 1\} \times C, sol(l_i) \mapsto (x_i, cl_i)$ such that the following constraints are respected:

– Any two lectures with one or more common teachers cannot be given simultaneously:

$$\sum_{i,j} [x_{i,j} = k][t_m \in t(l_i)] \leq 1, \quad 0 \leq k \leq D * H - 1, \quad 1 \leq m \leq |T|.$$

– Any two lectures cannot be given simultaneously in the same classroom:

$$\sum_{i,j} [x_{i,j} = k][cl(l_i) = c_m] \leq 1, \quad 0 \leq k \leq D * H - 1, \quad 1 \leq m \leq |C|.$$

– Any two lectures of the same group cannot be given simultaneously:

$$\sum_{i,j} [x_{i,j} = k][l_i \in g_m] \leq 1, \quad 0 \leq k \leq D * H - 1, \quad 1 \leq m \leq |G|.$$

– Two lectures of the same course cannot be given in the same day:

$$sub(l_i) = sub(l_j) \Rightarrow x_i/H \neq x_j/H, \quad 1 \leq i, \quad j \leq |L|.$$

– The quality of a timetable is calculated according to three criteria which are linearly combined with certain weights to produce the objective function which is to be minimized:
  - Uniform distribution of the teaching hours for every lecture group during the week. For each lecture group, this is the difference between the maximum and minimum number of teaching hours in a day during the teaching week:

$$\sum_{g \in G} \left(\max_{0 \leq d \leq D-1} \left\{ \sum_{l_i \in g} d_i [x_i/H = d] \right\} - \min_{0 \leq d \leq D-1} \left\{ \sum_{l_i \in g} d_i [x_i/H = d] \right\} \right).$$

  - Gaps between lectures of the same group during each day:

$$\textstyle\sum_{g \in G} \sum_{0 \leq d \leq D-1} \max(\{x_i + d_i + 1 : x_i/H = d, l_i \in g\} \cup \{0\}) \\ - \min(\{x_i : x_i/H = d, l_i \in g\} \cup \{0\}) - \sum_{l_i \in g} d_i$$

- Distances between lectures of the same course. As mentioned before, it is desired for these lectures to have a reasonable distance between them for educational purposes. This is calculated through a user-defined function $pen : [1..D-1] \mapsto \{0, 1, \ldots\}$ as

$$\sum_{s \in S} pen(\max\{x_i/H : sub(l_i) = s\} - \min\{x_i/H : sub(l_i) = s\}).$$

The low-level similarities between this and the student–course model are obvious. The way that the specialities of this case are implemented on top of the core object model is outlined in the next section.

5.3 Subclasses

The basic unary resource class is subclassed to provide classes for the teacher and lecture groups representation. Some extra features are added to provide new characteristics for these classes. In the case of teachers, for example, the number of days with teaching activities and the maximum number of continuous teaching hours in every day are needed. Thus, two new features, which are calculated appropriately, are added to this class:

```
class c_Teacher : public c_UnaryResource {
  IlcIntVar OccupiedDays;
  IlcIntVar MaxContinuous; };
```

In the case of lecture groups, two extra features are needed, namely the measure of uniform distribution of teaching hours through the days of the week and the sum of holes between lectures for all the days of the week:

```
class c_LectureGroup : public c_UnaryResource {
  protected:
  IlcIntVar Difference;
  IlcIntVarArray Holes; };
```

Penalties are associated with distances between lectures of the same subject. These penalties are provided by the user:

```
IlcIntArray Penalties;
if (nbLectures==1)
  Penalty=IlcIntVar(m,0,0);
else {
  IlcIntVar s1=IlcMax(StartVariables)/H;
  IlcIntVar s2=IlcMin(StartVariables)/H;
  Difference=s1-s2;
  IlcIfThen(Difference==k, Penalty==Penalties[k-1]); }
```

Classrooms are just a multiple resource and no extra members are needed for this class.

6 Search

One of the reasons that makes constraint programming attractive is the fact that logical implications stemming from the variables, their possible values and the constraints that are posted upon them are carried out in an implicit way without the programmer's manual intervention. Values that are possible for certain variables are ruled out if they violate any of the constraints. In theory, these implications can be carried out until all inconsistent combinations of values for the variables are ruled out. However, this is a task of exponential complexity, so, in practice, implications are calculated only to a limited extent and the task of finding solutions is accomplished through search. Usually, the search space of a problem is viewed as a tree, where each decision point represents a variable and the edges towards its children are its possible values.

In this case, the variables are the starting times for each lecture. Although there are variables for classrooms too, in our approach, the main decision is the one concerning time. After that decision is made, the lecture is placed on the most appropriate classroom, but that is not regarded as a decision or, in other words, no backtracking occurs for assigning a value to a classroom variable.

Two factors influence the efficiency of the search. The first one is the heuristic rules which involve the selection of a variable to be instantiated next (variable-ordering heuristics) and the selection of a value for that variable (value-ordering heuristics). Heuristics form the actual search tree by labeling each node with a variable and each edge with a value. Usually, the choices of the value-ordering heuristic are depicted by ordering the possible values from left to right, with the leftmost edge being the one selected first by the heuristic. The second factor is the search method which controls the order in which the nodes of the search tree are going to be examined.

6.1 Search Methods

Some of the most popular and efficient search methods were implemented, namely DFS, iterative broadening (IB) [14], LDS [18] and depth-bounded discrepancy search (DDS) [29]. These search methods evolved from the need to exploit the heuristic rules used in the search in the best possible way.

DFS implements the obvious way to explore a search space by examining all the leaves of the search tree from left to right and backtracking if needed. Although constraint propagation prunes hopeless parts of the search space, there is always the danger for such an approach to get trapped in a shallow part of the search tree. One of the reasons for this is that DFS considers each of the decisions of assigning a value to a variable of equal importance. That means that the first choice of a value for a variable made by the value-ordering heuristic is thought of as having the same probability to lead to a solution as the second or any of the latter decisions. IB narrows the search space by searching only the subtree which includes a certain number of the leftmost decisions of the value-ordering heuristic, practically exploring a subtree of limited width. This follows from the assumption that the first choices of the value-ordering heuristic are most probable to lead to a solution. In case this assumption is false, then the width to which the search was limited is incremented and the whole process starts from the beginning.

LDS gives even more importance to the heuristic by assuming that it makes none or just a few errors and using the total number of discrepancies or the deviations from the heuristic's decisions as a guide for the search. In a first iteration, the number of such discrepancies is assumed to be zero, thus only the leftmost decision of the heuristic is considered for each node. If the assumption proves to be false, then the number of discrepancies is incremented assuming that the heuristic might make at most one error and the process is repeated.

DDS is also a discrepancy-based search method. LDS revisits nodes that were examined in earlier iterations and DDS uses an algorithm that examines each node only once.

Both discrepancy-based methods are heavily depending on the heuristic's accuracy. However, it is not always the case that a heuristic making no more than, let us say, 5% erroneous decisions can be implemented. Actually, it is usual for heuristics to get confused deep in the search tree. Usually, this is tackled by adding a lookahead parameter of a certain depth in the method, allowing subtrees of that depth at the bottom of the search tree to be explored fully without taking the discrepancies that occur there into account. The trust of such methods in the heuristic's accuracy is also expressed by the assumption that heuristics usually make errors high in the search tree where decisions are not so informed. In our approach, the opposite was also implemented in LDS's case: that is a version that assumes that the heuristic fails deeply in the search tree.

In our implementation, it was also possible to loosen the confidence on the heuristic. For example, IB increases the width bound by one in every iteration and the same happens with the number of allowed discrepancies in LDS. We implemented versions that allowed bounds to be variable, thus allowing the search to explore wider areas of the search space in every iteration. This seems to contradict the very essence of these methods and that might be true in feasibility problems, but there is a good reason for this; these methods were designed for feasibility problems. In other words, they were designed assuming that the heuristic makes choices towards finding a solution and not necessarily a good one. On the other hand, timetabling problems are usually optimization problems, so the heuristics are targeted towards the quality of the solutions. In the experiments that follow, it will be seen that there is a trade-off between the ease in finding a solution and its quality. Since we are interested in quality, it is reasonable to examine how these search methods can be extended to handle heuristics towards better solution and how that affects their ability to find one.

6.2 Heuristics

The celebrated first-fail [17] and Brelaz [4] variable ordering heuristics are employed in our implementation. Also, some other general-purpose heuristics, the kappa family of heuristics for constraint satisfaction problems, proposed in [13], are also included. The first-fail selection criterion is supposed to follow the rule "in order to succeed, try first where you are most likely to fail". Although it is argued that selecting the variable with the least values in its domain leads to harder subproblems, this heuristic is efficient in many cases. The Brelaz heuristic extends it by tie-breaking on the number of neighboring unbound constraint variables in the constraint graph. The kappa family of heuristics is based on a measure of difficulty of a constraint satisfaction problem (CSP), namely the parameter kappa. The kappa heuristic selects the variable that is supposed to lead to an easier subproblem when instantiated. Since it is costly to compute, two approximations are proposed, the $E(N)$ and the rho heuristics, the former being closer to the original kappa measure. Since these parameters are extensively described in [13], we will describe them only briefly for the sake of completeness. The kappa parameter expresses the constrainedness of a problem, or else the difficulty of finding a solution for it, and is defined as follows; if $\mathcal{V}$ is the set of variables, C the set of constraints on those variables, C_i the set of constraints on variable i and m_v the cardinality of variable's v domain, then the parameter kappa of the problem is equal to

$$\kappa = \frac{-\sum_{c \in C} \log(1 - p_c)}{\sum_{v \in V} \log(m_v)}$$

where p_c is the tightness of a constraint involving two variables, that is the percentage of the combinations of values of the two variables that are ruled out by the constraint. Since this parameter is hard to compute, two approximations are provided; $E(N)$, which approximates the expected number of solutions for a problem

$$E(N) = \prod_{v \in V} m_v \times \prod_{c \in C} (1 - p_c)$$

and ρ, which approximates the solution density

$$\rho = \prod_{c \in C} (1 - p_c).$$

These parameters can be used as heuristic information in the following manner: since a problem with bigger kappa is tougher than one with a lower one, we should branch on the variable which would lead to an easier problem if instantiated. The same idea holds for $E(N)$ and ρ as well. The only practical problem in calculating these parameters in a real problem is that they were expressed for binary CSPs. However, we can use the assumption that the dominating constraint in timetabling problems is that no two activities demanding some common resource can happen simultaneously, which in general holds. Thus, we can easily calculate the parameter p_c for any pair of lectures demanding any common resource. Of course, we will have to ignore disjunctive resources – classrooms in our case – in our calculations, but that is not of great importance, since timetabling problems can also be solved in two phases, taking classrooms into account in the second one.

Although the literature is rich in general-purpose variable ordering heuristics, the same does not hold for value ordering ones. That is to be expected in optimization problems, since such heuristics have to do with the quality evaluation criteria and thus depend on the problem instance. In our case, heuristics making decisions according to the objective function of the DIT/UoA were implemented among which was one choosing the value that caused the minimum increase in the lower bound of the objective. Also, heuristics aiming at finding a feasible solution were implemented. Following the idea behind the kappa heuristics and the effectiveness of first-fail, a heuristic that chose the value for the current variable that maximized the product of the values of the remaining variables, as proposed in [12], was also implemented.

7 Experimental Results

Experiments were carried out on a Sun SPARCserver 1000 under SunOS 5.6 and with 256 MB of main memory. The problem to be solved involved 68 lectures that had to be scheduled in five days of nine teaching periods, each within four classrooms. The total duration of the lectures is 187 hours.

The heuristics used for variable selection were first-fail, Brelaz and the kappa family of heuristics Rho, E(N) and Kappa, denoted FF, BR, R, E and K respectively. Many different value selection heuristics were used in preliminary experiments. The most successful proved to be one that chose the value which would lead to the least increase in the lower bound of the minimization objective. Since the objective is represented by a constrained variable and the problem is a minimization one, an increase in the lower bound of the objective value means a certain decrease in the quality of the partial solution constructed until that point.

In Figure 1, a quality (smaller values represent higher qualities) versus time (measured in seconds) plot for DFS with some variable selection heuristics is shown. A major drawback of DFS can be observed there: there is not much improvement after the first solution is found. It can also be seen that the classic heuristics FF and BR are faster in finding a first solution than E and R (no solution was found with K). Drawing conclusions from just one data set is risky, but a first intuition is that the simple heuristics FF and BR can be more efficient in this specific problem than other more sophisticated and generic heuristics because of the following reason: the dominant type of constraint in course timetabling is a disjunctive constraint between lectures which has, more or less, the same effect in every pair of mutually exclusive lectures. Thus, what remains is the number of values in every variable's domain.

The LDS application on timetabling was not problem free. The first plot in Figure 2 shows the progress of search with LDS. It can be observed that the first solution is found rather late (an order of magnitude later than DFS) and the quality is not that good. The last solution found is quite satisfactory, but is found too late. This is a problem that should be expected; such methods were designed to address feasibility problems assuming that special heuristics would

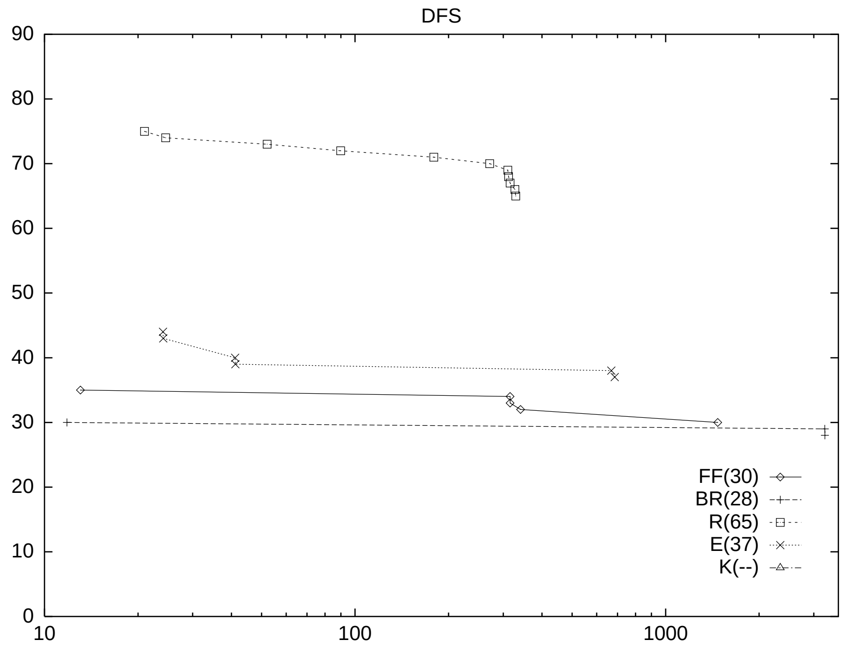

Fig. 1. DFS with varying variable selection heuristics

assist the search for a solution. Truly, the second plot shows the progress of search with LDS and a value selection heuristic targeting on feasibility. It is obvious that such a heuristic leads very quickly to a solution. On the other hand, the solution's quality is not satisfactory at all and that should be expected, since the value selection heuristic focuses on feasibility. In order to overcome these problems, the original method was modified. The third plot shows the progress of search with a modified version of LDS; the method assumes that the heuristic fails lower in the search tree. With that addition alone the search is much faster. The last plot shows one more addition: in every LDS iteration the discrepancy limit is not increased by a step of one, but by a step of three discrepancies. That means that less trust is justified upon the heuristic and its choices and the method is allowed to explore wider areas of the search space. There it can be observed that although LDS is somewhat slower than DFS in the beginning, it soon manages to find solutions of equal quality and, eventually, even better ones having the ability to escape from local minima.

Local search is a search method with great ability to escape from local minima and very popular in optimization problems. In [25], a local search variant for constraint programming is proposed, named large neighborhood search (LNS). The main idea is to iteratively keep a part of a solution stable while leaving the remaining variables unbound and thus performing a full search in a narrowed search space. Results are shown in Figure 3. It can be seen that this method with a good starting solution leads to the best-quality solution of the experiments

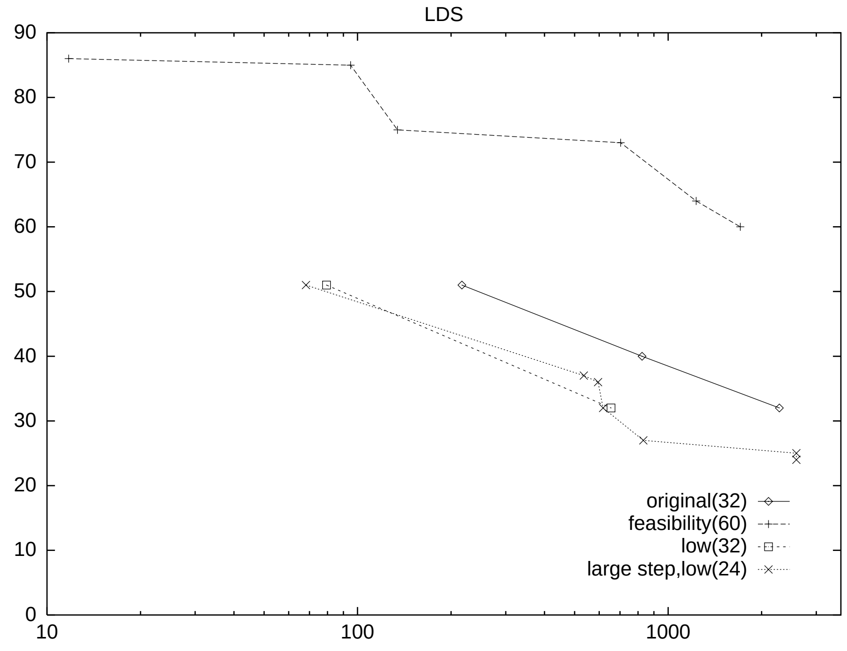

Fig. 2. LDS and variations with FF

presented. However, it should be noted that LNS alone is not able to provide as good solutions as a direct tree search. As Figure 3 shows, LNS needs more than 1000 s to reach a solution of quality slightly worse than the one found by a direct tree search guided by good heuristics in something more than 10 s. That is easily explained: LNS can lead only to minor local improvements. On the other hand, global decisions made by heuristics taking into account the problem as a whole can lead to larger improvements at the beginning of the search. Of course, direct tree search will eventually get stuck in a local optimum and that is the point where post-processing methods like LNS can be effectively used.

Another factor of interest is scalability. In this direction, we decided to use artificial data sets created on the real ones we used. Data sets up to six times bigger than the original data set used were created in the following way: the teachers and classrooms were kept the same while the number of lectures given and the total number of available time slots were multiplied by the corresponding factor. For example, to create a data set twice as large as the original, we created a timetabling problem with 10 instead of 5 days with 136 instead of 68 lectures while keeping all the other elements untouched. Then, of course, it would not be of interest to compare the objective value since the problems would be different. One parameter that could be measured and show how the approach scales with regard to the problem's size is the time needed to reach a first solution. Figure 4 shows how the time needed for reaching a first solution scales for the original problem mentioned above and for problems up to six times larger in size. All runs used plain DFS. The figure shows that time rises no faster than n^3 but faster than n^2. That is to be expected in constraint programming. The complexity of maintaining bound consistency is ed, where e is the number of constraints and d is the cardinality of a variable's domain. Increasing the problem size linearly involves increasing d linearly, but also increasing the number of variables linearly which leads to a quadratic increase of their interrelations or, in other words, constraints between them. This increase in running time is not prohibitive, since algorithms that scale polynomially are generally acceptable. We should also note that since the problem of constructing a timetable for a university occurs twice a year in most cases, there are no real-time constraints on the running time of such an application. Also, the running time can further be reduced by using more sophisticated propagation algorithms for the model's constraints, like network flow algorithms, for example.

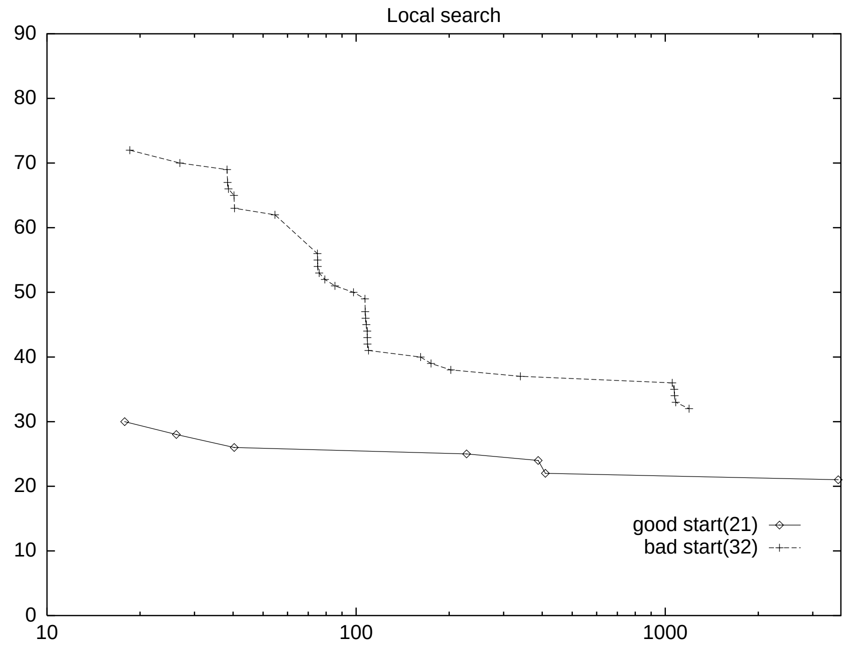

Fig. 3. Local search with varying quality of starting solutions

8 Conclusions

In this paper, we showed that it is possible to define, by exploiting the facilities offered by object-oriented design, a generic model for the university course timetabling problem, which might form the basis for dealing with the timetabling

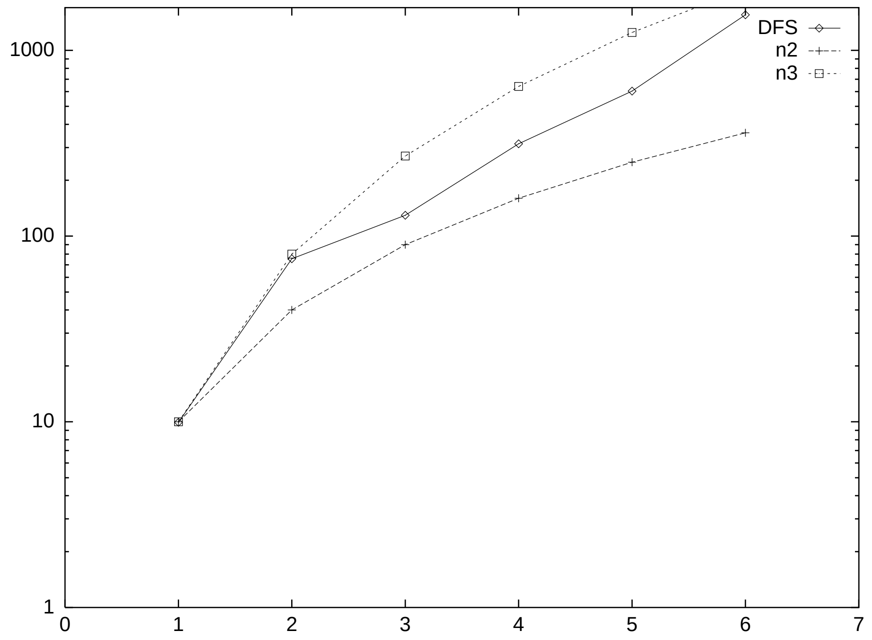

Fig. 4. Scaling of time for the first solution w.r.t. problem size

problems faced by most academic departments. We demonstrated the extensibility of the core model by applying it to the case of the DIT/UoA. The fact that the core model was extended might lead to the conclusion that it is too general, since too much extra needs to be implemented in order to cover any specific case. But, in order to produce the class `c_Teacher` from `c_UnaryResource`, 13 lines of code were needed. In the case of `c_LectureGroup`, 20 lines were needed. The aim of the above model was not to be able to cover any specific case, but to provide the framework on which any case could be covered. From a software engineering point of view, all these little additions for calculating holes between lectures or uniform distribution of lectures during a week or whatever could be stored in libraries and a user could pick from a large collection of such building blocks to construct classes that would represent the entities to model the problem to be solved, minimizing the amount of extra effort.

What is more, the whole idea is based on the constraint programming framework, since the tool employed is the Ilog Solver C++ library. Thus, the user is allowed to express hard and soft constraints with the same ease. For example, if one is concerned with the teaching hours of a teacher in a certain day, it is equally easy to post a hard constraint on the maximum number of these hours or add a penalty in the objective function, if that amount exceeds a certain limit, or associate preferences with the values of that variable.

A variety of search methods and variable ordering heuristics were implemented to be used for the search for near-optimum solutions. All search methods which were implemented behaved as they were expected to in theory. DFS was able to find feasible solutions fast, but could not improve them significantly in the long run. LDS, with some minor modifications aiming at the resolution of small conflicts at the bottom of the search tree, showed better long-term behavior, but was sometimes slow in finding a first solution. We expect LDS performance to scale better as the problem size increases. DDS would not be our method of choice, due to its lack of accepting modifications like LDS. IB did not improve DFS's performance significantly.

Concerning variable ordering heuristics, the celebrated first-fail and Brelaz heuristics performed better than any other variable selection heuristic. The negative result of our experiments was the bad performance of the kappa family of heuristics, which in some cases did not even lead to a feasible solution within a reasonable time limit. One explanation might be that the kappa measure is an approximate measure of a CSP's difficulty, which takes into account several factors, but mainly the number of values in each variable's domain and the constraint tightness between pairs of variables, that is the percentage of the total combinations between values of these two variables that are actually legal. In the timetabling problem, however, the tightness between any two variables sharing the same resources is constant. In other words, if some lectures are sharing the same resources the tougher one to schedule would be the one having the fewer possible slots regardless of the number of lectures, explaining why first-fail and Brelaz apply well on our problem.

As for value ordering heuristics, no general heuristic was applied. Heuristics targeted towards the DIT/UoA were used, that is heuristics that take into account holes between lectures, the distribution of lectures during the days of the week and the distances in days between lectures of the same subject. These heuristics worked fine by guiding the search towards feasible solutions of satisfying quality. At first glance, such heuristics do not have to do with feasibility and so the fact that the search finally reached feasible solutions might be considered coincidental. However, one can notice that the objective contains a factor expressing the number of holes between lectures, so that the search is guided towards more compact schedules. If the objective did not contain such a term, then a heuristic taking into account both the objective and the feasibility factor should be used. The feasibility factor could be measured by counting holes between lectures or, even better, with a heuristic calculating the product of the cardinality of all remaining variables' domains, as mentioned in the previous section concerning heuristics.

The problem of using tree search in optimization is that the search, either naive or sophisticated, will eventually get stuck in a local optimum. Using LNS helped in every case with our experiments. In no case, however, could LNS outperform direct tree search alone, so the best way to use it, in our opinion, is as a post-processing method after a normal tree search has been performed and a reasonable time cutoff limit reached.

References

1. Badri, M.: A Two-Stage Multiobjective Scheduling Model for [Faculty-Course-Time] Assignments. Eur. J. Oper. Res. **94** (1996) 16–28
2. Barbadym, V.A.: Computer-Aided School and University Timetabling: The New Wave. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 22–45
3. Boufflet, J.P., Negre, S.: Three Methods Used to Solve an Examination Timetable Problem. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 327–344
4. Brelaz, D.: New Methods to Color the Vertices of a Graph. JACM **22** (1979) 251–256,
5. Burke, E., Jackson, K., Kingston, J., Weare, R.: Automated University Timetabling: The State of the Art. Comput. J. **40** (1997) 565–571
6. Chahal, N., de Werra, D.: An Interactive System for Constructing Timetables on a PC. Eur. J. Oper. Res. **40** (1989) 32–37
7. de Werra, D.: An Introduction to Timetabling. Eur. J. Oper. Res. **19** (1985) 151–162
8. de Werra, D.: The Combinatorics of Timetabling. Eur. J. Oper. Res. **96** (1997) 504–513
9. Elmohamed, S., Coddington, P., Fox, G.: A Comparison of Annealing Techniques for Academic Course Scheduling. In: Burke, E., Ross, P. (eds.): Proc. 2nd Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1408. Springer-Verlag, Berlin Heidelberg New York (1997) 92–112
10. Erben, W., Keppler, J.: A Genetic Algorithm Solving a Weekly Course-Timetabling Problem. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 198–211
11. Frangouli, H., Harmandas, V., Stamatopoulos, P.: UTSE: Construction of Optimum Timetables for University Courses – A CLP Based Approach. In: Proc. 3rd Int. Conf. on the Practical Applications of Prolog (1995) 225–243
12. Geelen, P. A.: Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In: Proc. 10th Eur. Conf. on Artificial Intelligence (1992) 31–35
13. Gent, I.P., MacIntyre, E., Prosser, P., Smith, B.M., Walsh, T.: An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In: Proc. 2nd Int. Conf. on the Principles and Practice of Constraint Programming (1996) 179–193
14. Ginsberg, M.L., Harvey, W.D.: Iterative Broadening. Artif. Intell. **55** (1992) 367–383
15. Gotlieb, C.: The Construction of Class–Teacher Timetables. In: Proc. IFIP Congress (1962) 73–77
16. Gueret, C., Jussien, N., Boizumault, P., Prins, C.: Building University Timetables Using Constraint Logic Programming. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 130–145

17. Haralick, R.M., Elliott, G.L.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artif. Intell. **14** (1980) 263–313,
18. Harvey, W.D., Ginsberg, M.L.: Limited Discrepancy Search. In: Proc. 14th Int. Joint Conf. on Artificial Intelligence (1995) 607–613
19. Henz, M., Wurtz, J.: Using Oz for College Timetabling. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 162–177
20. ILOG S.A.: ILOG Solver 4.4: Reference Manual (1999)
21. ILOG S.A.: ILOG Solver 4.4: User's Manual (1999)
22. Lajos, G.: Complete University Modular Timetabling Using Constraint Logic Programming. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 146–161
23. Mehta, N.: The Application of a Graph Coloring Method to an Examination Scheduling Problem. Interfaces **11** (1981) 57–64
24. Rich, D.: A Smart Genetic Algorithm for University Timetabling. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 181–197
25. P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Proc. 4th Int. Conf. on the Principles and Practice of Constraint Programming (1998) 417–431
26. Stamatopoulos, P., Viglas, E., Karaboyas, S.: Nearly Optimum Timetable Construction Through CLP and Intelligent Search. Int. J. Artif. Intell. Tools **7** (1998) 415–442
27. Thompson, J., Dowsland, K.: General Cooling Schedules for a Simulated Annealing Based Timetabling System. In: Burke, E., Ross, P. (eds.): Proc. 1st Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1995) 345–363
28. van Hentenryck, P., Deville, Y., Teng, C.M.: A Generic Arc-Consistency Algorithm and its Specializations. Artif. Intell. **57** (1992) 291–321
29. Walsh, T.: Depth-Bounded Discrepancy Search. In: Proc. 15th Int. Joint Conf. on Artificial Intelligence (1997) 1388–1393
30. White, G., Zhang, J.: Generating Complete University Timetables by Combining Tabu Search with Constraint Logic. In: Burke, E., Ross, P. (eds.): Proc. 2nd Int. Conf. on the Practice and Theory of Automated Timetabling. Lecture Notes in Computer Science, Vol. 1408. Springer-Verlag, Berlin Heidelberg New York (1997) 187–198