NORTH-HOLLAND

CONSTRAINT LOGIC PROGRAMMING FOR EXAMINATION TIMETABLING

P. BOIZUMAULT, Y. DELON, AND L. PERIDY

▷ In this paper, we present an application of constraint logic programming to the examination timetabling problem of our university. Each year, in June, 4000 students in various programs must attend examinations during a couple of weeks for academic reasons. A set of examinations must be planned on specific half-days over a collection of rooms of different capacities. Various kinds of constraints must be taken into account. In particular, several examinations can be assigned to the same room if they respect the capacity constraint. This problem has been identified by operations researchers as a scheduling problem with disjunctive and cumulative conjunctive constraints and is classified as NP-complete. However no classical operations research (OR) approach is directly applicable. Our application has been developed using constraint logic programming over finite domains. First, we give a brief overview of OR approaches for solving examination timetabling problems. Then we describe the examination timetabling problem for our university and show how constraint logic programming over finite domains can be used to solve it efficiently. Finally, we illustrate the important potentialities of constraint logic programming for the prototyping and implementation of real-life applications. ◁

1. INTRODUCTION

Most educational institutions must schedule a set of examinations at the end of each session or year. In its simplest form, the problem can be defined as assigning

Address correspondence to Patrice Boizumault, Ecole des Mines de Nantes, 4, rue Alfred Kastler, La Chantrerie, 44070 Nantes Cedex 03, France or Yan Delon and Laurent Péridy, Institut de Mathématiques Appliquées, Université Catholique de l'Ouest, 3 Place André Leroy, B.P. 808, 49008, ANGERS Cedex 01, France.

Received July 1994; accepted June 1995.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00
SSDI 0743-1066(95)00100-X

a set of examinations to a fixed number of time periods so that no student is required to take more than one examination at any time. However, additional and specific constraints must be taken into account: room capacities, consecutive and nonconsecutive examinations, preassignments, exclusions and time preferences, etc.

Over the last 30 years, various systems have been developed in different high schools and universities to solve examination timetabling [7, 8, 10, 16, 17, 23, 41, 43]. All these proposals are related to particular examination timetablings, and are implemented by mixing various operations research (OR) techniques (graph coloring, integer programming, heuristics for the knapsack problem, heuristics for the traveling salesman problem, tabu search, ...) [7, 9, 19]. In fact, classical OR approaches cannot be directly applied to this kind of problem, so dedicated algorithms must be conceived and implemented generally in procedural languages. From a software engineering point of view, the time spent in developing such programs is very great. Also, adding new constraints frequently forces the program to be entirely redesigned.

As stated in [32], constraint logic programming (CLP) began as a merger of two declarative paradigms: logic programming and constraint solving. This combination helps make CLP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. CLP languages have already proven to be successful in tackling many combinatorial problems such as planning, scheduling, resource allocation, assignment problems, placement, and configuration.

In this paper, we show an application of constraint logic programming to the examination timetabling problem of our university. Each year, in June, 4000 students in various programs must attend examinations during a couple of weeks for academic reasons. The problem (for June 1993) consists of planning 308 different examinations on 33 half-days using 7 rooms of different capacities. A set of various constraints must be satisfied. In particular several examinations can be assigned to a same room if they respect the capacity constraint. This problem has been identified by operations researchers as a scheduling problem with "disjunctive" and cumulative conjunctive constraints and is classified as NP-complete. This problem does not fit with any classical OR approach due to the variety of the constraints that must be taken into account.

Our application has been built using constraint logic programming over finite domains, which provided an excellent framework for our development. First, we give a brief overview of OR approaches for solving examination timetabling problems. Then, after having described the specific problem of our university, we present our implementations using the CLP(FD) language CHIP [12, 14, 28]. Finally, we illustrate the important potentialities of constraint logic programming for the prototyping and implementation of real-life applications [37, 38]. In fact, CLP is very useful for building applications where no general algorithm is available and where changes may frequently arise.

2. AN OVERVIEW OF OPERATIONS RESEARCH APPROACHES

In this section, we give an overview of OR approaches for solving the examination timetabling problem. First we describe a solution to the simplified problem using graph coloring. Then we describe various applications that have been realized

at different universities. Finally, we discuss the adequacy of OR for solving such problems.

2.1. Simplified Problem

In its simplest form, the problem can be defined as assigning a set of examinations to a fixed number of time periods so that no student is required to take more than one examination at any time. The problem of finding a conflict-free assignment is structurally similar to the vertex coloring problem studied extensively in the literature on graph theory [20]: each course is represented by a vertex, and an edge connects two vertices if the corresponding courses have at least one student in common and, hence, cannot be scheduled at the same time period. This problem (although NP-complete) is solved quite efficiently by operations research [20]. So many OR approaches for solving examination timetabling problems are inspired from graph coloring heuristics.

However, practical examination timetabling problems differ from pure vertex coloring problems [9] because they must take into account multiple kinds of constraints such as:

- A limit on the number of students and/or examinations in any one period.
- Room capacity constraints (each examination is assigned to a particular room).
- Consecutive examination constraints (certain exams must occur in adjacent time periods).
- Nonconsecutive conflict constraints (no examinations in succession for any student).
- Preassignments (certain examinations are preassigned to specific periods).
- Exclusions and time preferences (certain examinations are excluded from particular periods).
- Each student's examinations should be spread over the examination period.

These constraints are not required in all examination timetabling problems and are specific to particular academic institutions. So dedicated algorithms must be conceived in order to solve each type of examination timetabling problem. Moreover, graph coloring heuristics must be adapted in order to take into account such constraints.

2.2. Various Applications

In this section, we give an overview of three systems that have been implemented at different universities. For more details, see [9], which gives a survey of examination timetabling problems.

In 1968, Wood devised an examination scheduling algorithm that was implemented at the University of Manchester (England). His primary concern [43] was that examinations had to be scheduled into a set of designated rooms. Moreover, he tried to minimize the number of conflicts (no consecutive examinations on the same day for any student). For this, Wood implemented a look-ahead algorithm and assigned the selected room with the "closest fit," namely, the least acceptable number of places. When his algorithm failed, Wood claimed that inspection of the conflict pattern related to the unscheduled courses "clearly" reveal the subjects that

cause the difficulty. These subjects are preassigned manually and the algorithm is repeated.

In 1978, Carter developed an algorithm for final examination scheduling at the University of Waterloo (Canada). This system was extended in 1985 for scheduling all area high school examinations. The basic algorithm uses graph coloring techniques and integrates the ability to take into account preferred constraints such as that several courses must be preassigned to fixed time periods or that no student should be required to sit for three or more consecutive exams.

In 1978, Desroches, Laporte and Rousseau presented the system HOREX, a name derived from the French "Horaire" for timetable. Their method consisted of the following general steps: First, they find p sets of nonconflicting examinations, where p is the number of periods (graph coloring algorithm). Then they combine these sets in order to have a minimum number of students having two examinations the same day (branch and bound code for integer linear programming). In order to maximize the number of examinations scheduled in the morning, they use a heuristic for the knapsack problem. Finally, the days are ordered using a traveling salesman problem heuristic in order to minimize the number of "successions" in which a student must take examinations on consecutive days.

2.3. Tabu Search

Tabu search [24, 25] is an effective local search method that moves step by step from an initial solution of a combinatorial optimization problem toward a solution that is expected to be optimal or near-optimal. For each solution s, such a method requires the definition of a neighborhood $V(s)$, consisting of solutions reachable in one step from s. The basic step is to move from the current solution s to the best solution $s*$ of $V(s)$, even if it is not better than s. A tabu list T is used to avoid cycling. It acts as a short-term memory by storing a description of the NT last moves or solutions. When exploring $V(S)$ to find $s*$, T is scanned to avoid the so-called tabu moves, which could bring the search back to a previous iteration. The procedure stops after a maximum number of iterations and outputs the best solution found.

Boufflet and Negre [7] implemented such a method for examination planning. They transformed the problem into a graph coloring problem in which the vertices are the examinations and the edges the constraints. A weight is assigned to each constraint. The problem is to find a p coloring that minimizes a multiobjective function. Boufflet used a tabu search on a graph coloring problem inspired by Hertz and de Werra's techniques [26]: each vertex is given one of the p allowed colors. This assignment may not be a coloring (if some adjacent vertices are assigned the same color). The idea is to minimize the conflicts (two adjacent vertices with the same color) by exploring the neighborhood of the initial solution (modification of the color of a node).

2.4. Adequacy for Examination Timetabling Problems

As previously seen, an examination timetabling problem is closely tied to a particular academic institution. In each case, very specific constraints must be taken into account. Moreover, no classical OR model can be directly applied for taking into account the various kinds of constraints, so either dedicated algorithms must be conceived and implemented, or local search procedures including the extra constraints in the objective function must be used. From a software engineering point

of view, the time spent in developing such programs is very great. Also, adding new constraints frequently forces the program to be entirely redesigned.

3. PROBLEM STATEMENT

At the West Catholic University in Angers, each year in June, 4000 students in various programs must attend examinations during a couple of weeks. The problem for June 1993 consisted of planning 308 different examinations on 33 half-days. For this, seven rooms of various capacity could be used. Some rooms were not available every half-day and, moreover, several examinations could be assigned to the same room. This latter constraint differentiates our problem from that reviewed in Section 2, for which each examination had its own room.

A set of various constraints of different types must be satisfied. We have classified these constraints into 12 categories:

- C1, fixed date: The date of the examination is imposed.
- C2, three other universities depend on the university located in Angers. This creates some time constraints concerning the examination organization.
- C3, special orals: Some examinations have to be planned very quickly because of the possibility of special orals. Notice that orals are managed independently by each department.
- C4, release date, due date: Some examinations have to start after a release date and finish before a due date.
- C5, examination incompatibilities: A student cannot attend several examinations at a time.
- C6, examination and half-day incompatibilities: Some examinations cannot be planned on some particular half-days.
- C7, coupling: Some examinations have to be planned on two consecutive half-days.
- C8, decoupling: Some examinations have to be planned on different days. A time lag of k half-days must be left between two given examinations. Moreover, we know the precedence between the two examinations.
- C9, precedence constraints: A given examination must occur before another one.
- C10, time constraints concerning rooms, i.e., a four hour examination cannot take place in a room available for only three hours.
- C11, room availability: Several examinations can be assigned to the same room if they satisfy the capacity constraint.
- C12, priority is given to high capacity rooms, so examinations should preferably be assigned to the rooms with the highest capacities.

Notice that the first 11 kinds of constraints are imperative, but the twelfth constraint can be considered as a preference.

4. A BRIEF OVERVIEW OF CLP(FD)

Constraint logic programming languages over finite domains such as CHIP [12, 14, 28] or clp(FD) [15] use constraints solving and local consistency techniques

inherited from CSP (constraint satisfaction problems [42]). CLP(FD) languages have already proven to be successful in tackling many combinatorial problems such planning, scheduling, resource allocation, assignment problems, placement, and configuration.

Our application has been built using the finite domains of CHIP. The next two sections describe the numerical and symbolic constraints of CLP(FD) languages. The last one presents the *cumulative* constraint of CHIP.

4.1. Numerical Constraint Handling

Each constrained variable has a domain (finite set of scalar values) that must be declared a priori. Three kinds of linear constraints are provided: equality, disequalities, and inequality. Each can be applied on linear terms built upon domain variables and constants.

4.2. Symbolic Constraint Handling

Symbolic constraints are also handled. A very useful one is `element(N, List, Value)`. It specifies, as an internal system constraint, that the `Nth` element of the list `List` must have the value `Value`. `List` is a nonempty list of natural numbers and `Value` is either a natural number, a domain variable or a free variable.

The most interesting use of the constraint `element/3` is when `N` or `Value` are uninstanciated variables. Therefore, as soon as the domains of `N` or `Value` change, inconsistent values are removed from domains. The `atmost/3` `(atleast/3)` constraints impose that at most (at least) `N` elements of a list `List` have the value `Value`.

4.3. The Cumulative Constraint

The cumulative constraint has been introduced in CHIP in order to solve scheduling and placement problems [1]. The `cumulative` constraint has eight arguments:

`cumulative(`*LStarts, LDurations, LResources,* `?, ?,` *High,* `?, ?)`

We have only mentioned the four parameters that have been used to solve our problem.

This constraint is usually introduced by taking a scheduling problem as an example. Let us consider the scheduling of n tasks (ti) of known duration where each task consumes a certain amount of an available resource. The cumulative constraint states that, at any instant t of the schedule, the total of the amount of resource for the tasks that overlap t does not exceed the maximal available amount of the resource.

- Let $LStarts = [S1, \ldots, Sn]$ be the list of the starting dates of each task ti, i in $[1, \ldots, n]$.
- Let $LDurations = [D1, \ldots, Dn]$ be the list of the duration of the tasks ti, i in $[1, \ldots, n]$.
- Let $LResources = [R1, \ldots, Rn]$ be the list of the amount of resource required by the tasks ti, i in $[1, \ldots, n]$.
- Let $a = \min(\min(S1), \ldots, \min(Sn))$ and $b = \max(\max(S1) + \max(D1), \ldots, \max(Sn) + \max(Dn))$; a is the smallest release date and b is the largest due date of the schedule.
- Let *High* be the upper limit of the amount of available resource.

The following constraints are then enforced:

1. $\forall i \in [1, \ldots, n],\ Di > 0,$
2. $\forall i \in [1, \ldots, n],\ Ri > 0,$
3. $\forall i\ a \leq i \leq b,$

$$\max \sum_{j/Sj<=i<Sj+Dj} Rj \leq High.$$

The last constraint specifies that, at any instant i of the schedule, the summation of the required amounts of resource of the tasks that are active (tj) does not exceed *High*, the maximal quantity of available resource.

Let us consider three major uses of the cumulative constraint [1]; other applications are presented in [2, 39]:

1. Considering Figure 1, there are three tasks to schedule: The first task uses one unit of the resource during four consecutive periods; tasks 2 and 3 use two units during, respectively, two and three periods. At any time the total amount of resource used by the different tasks is always less than or equal to 3.
2. For Figure 2, all the task durations are equal to 1. This particular case corresponds to the bin packing problem [40]: m bins of fixed capacities and n objects of fixed size to put in these bins.
3. The third example (see Figure 3) forbids having a cumulative amount of resource greater than 1. This corresponds in scheduling to the problem of two tasks that cannot be executed at the same time because they share the same resource (so-called disjunctive tasks).

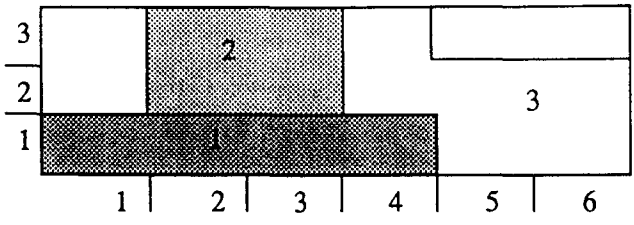

FIGURE 1. `cumulative`([1,2,4], [4,2,3], [1,2,2], 3).

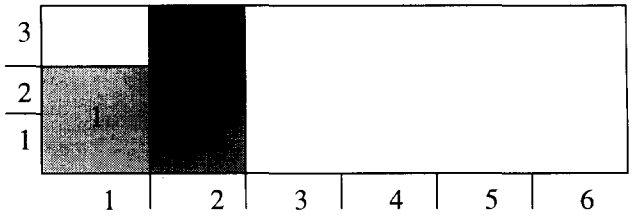

FIGURE 2. `cumulative`([1,2,2], [1,1,1], [2,1,2], 3).

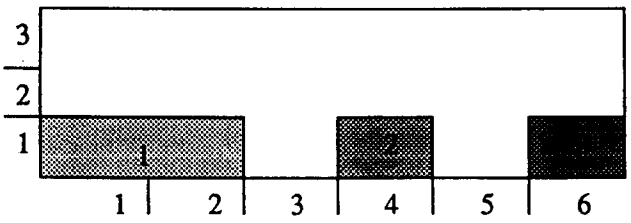

FIGURE 3. `cumulative`([1,4,6], [2,1,1], [1,1,1], 1).

5. IMPLEMENTATIONS

Our problem fits very well with the constraint logic programming over finite domains paradigm. In fact, every examination's date can be represented by a domain variable ranging over the 33 half-days. Each room can be identified by a scalar value ranging from 1 to 7. Moreover, we can naturally express all the constraints of our problem using the numerical and the symbolic constraints over finite domains of CHIP.

The usual development of an application over finite domains consists of modeling the data and the domain variables, thus imposing the constraints, and finally defining the labeling strategy [4]. We follow this pattern in our presentation. We describe the two implementations realized with or without the *cumulative* constraint.

5.1. Modeling the Data

We introduce 308 domain variables representing the 308 examinations to plan: $Exams = [E1, \ldots, E308]$. The domain of each variable is determined by its release date and due date (data given by the university). The aim is to find, for each domain variable Ei, one half-day that satisfies all the constraints. Moreover, for each examination, we must assign it to a room, respecting the capacity constraints. So we consider 308 domain variables `Rooms` $= [R1, \ldots, R308]$ ranging over 1–7.

Since several examinations can be planned on the same half-day, it is important to know the cumulative number of students for each room and each half-day. So we use some Prolog terms that represent this information (list of cumulative numbers for each room and each half-day, ...).

Finally, for each examination the number of students is known (data given by the academic services). The calendar of room availability and the list of all the durations of the half-days are also given by the academic services.

5.2. Stating the Constraints

We have classified the various constraints into five categories according to their implementation:

- Constraints directly applied on the domains:
  1. C1 is translated by an equality constraint (40 constraints).
  2. C2, C3, and C4 are translated by a restrictive definition of the domains (80 constraints).

- Incompatibility constraints:
  1. C5 is translated by a disequality constraint between two domain variables (notice that there are 1930 such constraints).
  2. C6 is translated by a disequality constraint between a domain variable and a constant (50 constraints).
- Precedence constraints:
  1. C7, coupling: equality `Ei = Ej + 1`.
  2. C8, decoupling: `Ei = Ej + k` (a lag of k half days must be left).
  3. C9, classical precedence: `Ei + k <= Ej` (there are 50 constraints of type C7, C8, and C9).
- Time constraints, C10: For the `Ei` examination (duration `DEi`), let `D = [D1,...,D33]` be the list of the maximum durations of all the half-days. The conjunction of constraints, `element(Ei, D, Durat), DEi <= Durat`, defines the time constraints concerning room availability (there are 308 such constraints).
- The capacity constraints C11 and C12 are managed differently in the two implementations.

5.3. First Implementation

In the first implementation, the room capacities (C11) are verified a posteriori after labeling. The labeling stage consists of enumerating the various potential solutions. Obviously, the search space is a priori pruned by the active constraints. We developed various labeling strategies in order to have an efficient resolution of the problem.

The labeling stage is implemented as follows:

```
top(Exams,Rooms) :-
      Exams = [E1,...,E308],
      Exams::1..33,
      Rooms = [R1,...,R308],
      Rooms::1..7,
      impose_constraints_c1_to_c10(Exams, Rooms, Env),
      merge(Exams, Rooms, ExamsRooms),
      labeling(ExamsRooms, Env).

labeling([], _).
labeling([E|Es], Env) :-
       select_exam(Exam@Room, [E|Es], ExamsRooms, Env),
       select_halfday(Exam, Env),
       select_room(Exam, Room, Env),
       verify_c11(Room, Env, NewEnv),
       labeling(ExamsRooms, NewEnv).
```

where `Env` mainly describes the list of cumulative numbers of students by half-day and by room.

Various labeling strategies have been tested:

- *First-fail:* Examinations are selected with the first-fail heuristic [27]. Half-days are labeled using the built-in predicate `indomain/2`. At the end of the labeling stage, we could have a set of examinations mutually incompatible and not enough half-days to plan them. This leads to inefficiency with many useless backtracks.

- *Best fit:* We share out the examinations in order to have a uniform distribution of the cumulative numbers of students on all half-days. We keep the first-fail strategy to select the examinations, but we assign each examination to the half-day that has the smallest cumulative number of students (best-fit strategy). At the end of the labeling stage, some examinations with a large number of students have to be planned and it is not possible to find a room with a sufficient capacity for them.
- *Best-fit decreasing:* We select the examinations in the decreasing order of their number of students: We keep the best-fit strategy for choosing a half-day.

The last two strategies are inherited from the so-called "bin packing algorithms" [21].

5.4. Second Implementation

With the *cumulative* constraint, we can state the capacity constraint C11 before the labeling stage, and so get a better pruning of the search tree.

Our problem can be modeled as a bin packing problem (see Figure 2). In fact, we have m (=$7 * 33$) effective rooms of fixed capacity and n (=308) examinations to affect to these rooms. Each examination corresponds to a task of duration 1 consuming an amount of resource equal to its number of students.

Let:

- $[E1, \ldots, E308]$ be the list of the examinations.
- $[R1, \ldots, R308]$ be the list of the rooms (ranging over 1–7).
- $[S1, \ldots, S308]$ be the second list of the rooms. Each Si is a domain variable ranging over 1–$(7 * 33)$. The Si's are linked to the half-day Ei and the room Ri by the constraint $Si = 7 * (Ei - 1) + Ri$.
- $[e1, \ldots, e308]$ be the list of the numbers of students of each examination.

So the constraint C10 can be stated as follows: *cumulative*$([S1, \ldots, S308], [1, \ldots, 1], [e1, \ldots, e308], ?, ?, 370, ?, ?)$, where 370 is the capacity of the largest room. The capacity constraints are now taken into account a priori.

This simple implementation would be sufficient if all the rooms had the same capacity, but this was not the case. In order to solve this problem, we introduce virtual examinations in order to simulate an identical behavior of the rooms. For each room Si, we introduce a virtual examination with an effective of 370 minus the capacity of Si. With these extra examinations, we can now tackle uniformly the various rooms.

The labeling stage is now implemented as follows:

```
top(Exams, Rooms) :-
     Exams = [E1,...,E308],
     Exams::1..33,
     Rooms = [R1,...,R308],
     Rooms::1..7,
     [S1,...,S308]::1.231,
     impose_constraints_c1_to_c10(Exams, Rooms, Env),
     cumulative([S1,...,S308], [1,...,1], [e1,...,e308],?,?,
                370,?,?),
     link(Exams, Rooms, [S1,...,S308]),
     merge(Exams, Rooms, ExamsRooms),
     labeling(ExamsRooms, Env).
```

```
labeling([], _).
labeling([E|Es], Env) :-
      select_exam(Exam@Room, [E|Es], ExamsRooms, Env),
      select_halfday(Exam, Env),
      select_room(Exam, Room, Env, NewEnv),
      labeling(ExamsRooms, NewEnv).
```

We tried various labeling strategies. The first one corresponds to the standard first-fail strategy. The second one consists of sorting the examinations by their decreasing number of students and then to affect each examination to the half-day with the smallest cumulative number of students (best-fit decreasing strategy). Then, we choose for the examination the largest available room with respect to the constraint C12.

The main difference compared to the first implementation is that the capacity constraints are managed a priori thanks to the `cumulative` constraint. Let us consider three examples describing the behavior of the `cumulative` constraint for our problem. Six examinations must be planned over two half-days using two rooms, each with a capacity of 5 (if the two rooms did not have the same capacity, a virtual exam would be assigned to the smaller room).

In the first example, after having labeled `[S1,S2,S3]`, the `cumulative` constraint detects that there is only one remaining room for `E6`, namely, `S6=4`. So trying to assign the examination `E4` to room `4` leads to a failure:

```
example1(Exams,Rooms) :-
      Exams=[E1,E2,E3,E4,E5,E6],
      Exams::1..2,
      Rooms=[R1,R2,R3,R4,R5,R6],
      Rooms::1..2,
      S=[S1,S2,S3,S4,S5,S6],
      S::1..4,
      S1+2#=2*E1+R1, S2+2#=2*E2+R2, S3+2#=2*E3+R3,
      S4+2#=2*E4+R4, S5+2#=2*E5+R5, S6+2#=2*E6+R6,
      cumulative(S,[1,1,1,1,1,1],[2,2,2,2,3,4],?,?,5,?,?),
      S1#=1, S2#=2, S3#=3, S4#=4.
```

In the second example, after having labeled `[S1,S2,S3]`, there is still only one place for `E5` in room `4` (`S5=4`), so there is no remaining place for `E6`. The `cumulative` constraint still leads to failure:

```
example2(Exams,Rooms) :-
     Exams=[E1,E2,E3,E4,E5,E6],
     Exams::1..2,
     Rooms=[R1,R2,R3,R4,R5,R6],
     Rooms::1..2,
     S=[S1,S2,S3,S4,S5,S6],
     S::1..4,
     S1+2 #=2*E1+R1, S2+2#=2*E2+R2,  S3+2#=2*E3+R3,
     S4+2#=2*E4+R4,S5+2#=2*E5+R5,S6+2#=2*E6+R6,
     cumulative(S,[1,1,1,1,1,1],[3,3,3,2,3,3],?,?,5,?,?),
     S1#=1, S2#=2, S3#=3.
```

However, the cumulative constraint only applies local consistency techniques and does not guarantee in every case the existence of a solution. Let us consider the third example: If we commit S1 and S2 to share the same room, there is no solution. However, all of the domains of [S3,S4,S5,S6] have a size greater than or equal to 2 (in fact DS3=DS4=DS5=DS6=[2,3,4]), so the failure will only be detected by further labeling:

```
example3(Exams,Rooms) :-
     Exams=[E1,E2,E3,E4,E5,E6],
     Exams::1..2,
     Rooms=[R1,R2,R3,R4,R5,R6],
     Rooms::1..2,
     S=[S1,S2,S3,S4,S5,S6],
     S::1..4,
     S1+2 #=2*E1+R1, S2+2#=2*E2+R2, S3+2#=2*E3+R3,
     S4+2 #=2*E4+R4, S5+2#=2*E5+R5, S6+2#=2*E6+R6,
     cumulative(S,[1,1,1,1,1,1],[1,2,4,4,4,3],?,?,5,?,?),
     S1#=1, S2#=1.
```

Consequently, the cumulative constraint enables a better pruning for our problem. In fact it always guarantees a "potential" room for each examination by applying local consistency techniques, so a great number of failures are detected sooner. However, this constraint does not ensure the existence of a solution (in the third example, four examinations have the same set of three "potential" rooms and exclude each other).

6. RESULTS

6.1. Efficiency Results

For June 1993, we have about 2600 constraints, and we found a solution with 28 half-days (which is less than the 33 half-days imposed by the academic services). In less than 10 s, we can also prove that there is no solution under 28 half-days. The two implementations have found a first timetabling in less than 1 min of computing time (30 s for stating the constraints and 25 s for labeling).

Next, we increased the number of students for each examination (first 5 and next by 10) in order to test the robustness of our modeling and labeling. With the first implementation, we did not get any solution in 60 h of computing time. With the second, we only get solutions using the best-fit decreasing labeling strategy.

Finally, we generated tests by modifying the number of students for each examination by $\pm 10\%$. Without stating a priori the capacity constraints, we only get solutions with the best-fit decreasing strategy (80%). Using the cumulative constraint, we solved all 100 tests using the best-fit decreasing strategy.

Implementing an "intelligent" labeling strategy is important when using constraint logic programming over finite domains. For our problem, the best suited strategies are those inherited from bin packing heuristics [21].

6.2. Software Engineering Considerations

We spent a couple of weeks to solve this real-life problem using CLP over finite domains. The code (without the constraints, which are stored in separate files)

is about 350 lines. It was relatively easy to adapt the program to the requirements of the cumulative constraint. Moreover, additional constraints and a few changes have been taken into account easily. This illustrates the quality of software engineering development in constraint logic programming languages [3, 5, 11, 37, 38].

As quoted in [34], constraint logic programming has proven to be useful in building applications where there are no general algorithms to solve the particular problems at hand and/or where the requirements are changing rapidly. In such situations, problem-specific strategies are often used. Constraint logic programming enables modelization of such problems and provides a way to define efficient search strategies within a unified framework.

7. ADDING PREFERENCE CONSTRAINTS

The academic secretary has run the program and received the solutions as previously described, but for many groups of students, their examinations are planned very closed to each other. They have, for example, two examinations planned on the same day. So the directors of each department decide to define the notion of "heavy examination," which corresponds to an important course. They then request that two heavy exams occur on two consecutive half-days (this leads to 600 preference constraints).

This led us to tackle the problem using preference constraints of the form ($Ei \geq Ej + 2$ or $Ej \geq Ei + 2$). They are a special case of the so-called "disjunctive constraints" in the CLP community. First, we will describe the implementation of the disjunctions and then discuss the problem of taking into account preference constraints.

7.1. Implementing Disjunctive Constraints

There are several ways to handle, with more or less efficiency, disjunctive constraints in CHIP [5]. Let us consider two tasks `Ti` and `Tj` of duration `Di` and `Dj` that cannot be managed at the same time.

The first way is to introduce a choice point:

```
disjunctive(Ti,Di,Tj,Dj) :-
   Ti #>=Tj+Dj.
disjunctive(Ti,Di,Tj,Dj) :-
   Tj #>=Ti+Di.
```

This leads to a lot of inefficiency because the disjunction is not considered as a constraint. A lot of choice points are introduced and useless backtrackings occur. For our problem, such an implementation would lead to 2^{600} alternatives!

The second way is to use demons that implement conditional propagation:

```
disjunctive(T1, D1, T2, D2) :-
   if T1+D1 > T2 then T2+D2 <= T1,
   if T2+D2 > T1 then T1+D1 <= T2.
```

This implementation is more efficient, but is too weak to wake the constraints and prune the search tree.

The third way, is to use the cumulative constraint:

```
disjunctive(T1, D1, T2, D2) :-
   cumulative([T1,T2], [D1,D2], [1,1], 1).
```

We first tried to implement the preference constraints with the cumulative primitive, but did not get good results: The pruning realized was not efficient enough.

Finally, taking into account that the constraint was symmetrical (`|Ti -Tj| >=2`), we used the constraint `distance/4` to implement the preference constraints.

The constraint `distance(X,Y,Op,K)` states that `|X - Y| Op K`, namely, for us `distance(Ti,Tj,>=,2)`. This led to very good results (a labeling in 25 s for June 1993 taking into account the 600 preference constraints); the `distance/4` constraint realizes, in this case, a better pruning of the search tree.

There are other ways to efficiently tackle disjunctive constraints. In cc(FD) [29], the cardinality operator enables us to impose that "at least n" and "at most m" constraints of the set of constraints must be verified. So a disjunction may be specified as follow:

```
disjunctive(T1, D1, T2, D2) :-
   #(1, _, [T1>=T2+D2, T2>=T1+D1]).
```

Finally, another way to implement disjunctions efficiently is the notion of constructive disjunction proposed by Van Hentenryck [30, 31, 33].

7.2. Taking into Account Preference Constraints

In order to test the robustness of our approach, we ran our software on 100 benchmarks constituted by the data of June 1993, where the number of students for each examination has been modified by $\pm 10\%$.

On 100 tests, we got 55 solutions in less than 20 s of labeling time, taking into account the initial constraints plus the whole preference ones (3200 constraints). For the other 45 tests, we did not get any answer in 5 min of CPU time. In fact, for these tests, experimentation shows that it would suffice to relax two or three constraints in order to solve the problem efficiently.

CHIP does not yet provide the ability to implement this approach. Moreover, we cannot hierarchize these constraints as in HCLP(R) [6] or Preferred CLP(FD) [22], because they are all at the same level. Proposals have been made for timetabling problems. In [35], the system IHCS written in C enables hierarchization of constraints and implements an automatic relaxation using intelligent backtracking. In [13], Cousin also proposed a way to relax constraints and integrated this with CLEF, a CLP system written in Lisp [36].

We are currently developing a heuristic in order to manage such preference constraints. First, we impose the imperative constraints, and then we try to built sets of preference constraints that are interrelated, thus imposing them incrementally and finally exhibiting the problematic ones.

8. EXTENSIONS AND FURTHER WORKS

Our experience shows that the CLP programmer needs high level primitives to express its particular constraints. Such primitives must be efficiently implemented in order to significantly reduce the search space.

Our experience shows the importance of an intelligent labeling strategy. As quoted by Tsang [42], one important issue is the ordering in which the variables are selected and the ordering in which the values are assigned to each variable. Different orderings affect the efficiency of the search strategies significantly. Another way to optimize the system is to add redundant constraints [18, 28], but this was not necessary for our problem.

Our experience points out the importance of introducing the relaxation of constraints in CLP. Most real life problems (timetabling, for example) induce imperative constraints and preference constraints. With the CLP tools available today, however, we cannot efficiently tackle constraint hierachization and constraint relaxation.

Finally our experience illustrates the important potentialities of constraint logic programming for the prototyping and implementation of real-life applications.[1] We can solve problems with more than 3000 constraints. Moreover, the conciseness of the programs and the short development times allow us to rapidly develop alternative versions. Indeed, various heuristics have been developed, tested, and validated in a very short development time.

We thank E. Pinson, M. Rueher, and N. Edgard for their fruitful comments.

REFERENCES

1. Aggoun, A. and Beldiceanu, N., Extending Chip in Order to Solve Complex Scheduling and Placement Problems, in: *Journées Francophones de la Programmation en logique*, Lille, 1992.
2. Beldiceanu, N. and Contejean, E., Introducing Global Constraints in Chip, *Math. Comput. Modelling* 20:97–123 (1994).
3. Bellone, J., Chamard, A., and Pradelles, C., Plane: An Evolutive System for Aircraft Production Written in Chip, in: *First International Conference on Practical Applications of Prolog*, London, 1992.
4. Boizumault, P., Delon, Y., and Péridy, L., Planning Exams Using Constraint Logic Programming, in: *Second International Conference on Practical Applications of Prolog*, London, 1994.
5. Baptiste, P., Legeard, B., and Varnier, C., Hoist Scheduling Problem: An Approach Based on Constraint Programming, in: *IEEE International Conference on Robotics and Automation*, Nice, 1992.
6. Borning, A., Maher, M., Martingale, A., and Wilson, M., Constraint Hierarchies and Logic Programming, in: *Sixth International Logic Programming Conference*, Lisboa, 1989.
7. Boufflet, J. P. and Negre, S., About Planning an Examination Session, in: *ECCO VII*, Milan, 1994.
8. Barham, A. M. and Westwood, J. B., A Simple Heuristic to Facilitate Course Time-Tabling, *J. Oper. Res. Soc.* 29:1055–1060 (1978).
9. Carter, M. W., A Survey of Practical Applications of Examination Timetabling Algorithms, *Oper. Res.* 34:193–202 (1986).
10. Vi Cao, N., du Merle, O., and Vial, J. P., Un Système de Confection Automatisée d'Examens, *Rev. Syst. de Décision* 1(4):377–399 (1992).

[1] It is important to notice that the resolution of this real-life problem does not imply the systematic resolution of the general problem defined in the introduction: The scheduling problem with disjunctive and cumulative conjunctive constraints is NP-complete.

11. Chamard, A. and Fischler, A., A Workshop Scheduler Written in Chip, in: *Second International Conference on Practical Applications of Prolog*, London, 1994.
12. Chip User's Guide, Cosytec SA, Orsay, France, 1993.
13. Cousin, X., Applications de la Programmation en Logique avec Contraintes au Problème d'Emploi du Temps, Ph.D. thesis, Rennes University, France, 1993.
14. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F., The Constraint Logic Programming Language Chip, in: *International Conference on Fifth Generation Computer Systems*, Tokyo, 1988.
15. Diaz, D., `clp(fd)` User's Manual, in: *Inria*, Le Chesney, France, 1994.
16. Desroches, S. and Laporte, G., Examination Timetabling by Computer, *Oper. Res.*, pp. 351–360, 1984.
17. Desroches, S., Laporte, G., and Rousseau, J. M., Horex: A Computer Program for the Construction of Examination Schedules, *INFOR* 16, 1978.
18. Dincbas, M., Simonis, H., and Van Hentenryck, P., Solving Large Combinatorial Problems in Logic Programming, *J. Logic Programming* 8(1–2):74–94 (1990).
19. de Werra, D., An Introduction to Time Tabling, *European J. Oper. Res.* 19:151–162 (1985).
20. de Werra, D., Heuristics for Graph Coloring, *Comput. Suppl.* 7:191–208 (1990).
21. Dyckhoff, H., A Typology of Cutting and Packing Problems, *European J. Oper. Res.* 44:145–159 (1990).
22. Fages, F., Fowler, J., and Sola, T., Handling Preferences in Constraint Logic Programming with Relationnal Optimization, in: *PLIP'94*, 1994.
23. Feldman, R. and Golumbic, M., Optimization Algorithms for Student Scheduling via Constraint Satisfiability, *Comput. J.* 33(4):356–364 (1990).
24. Glover, F., Tabu Search: Part 1, *ORSA J. Comput.* 1(3):190–206 (1989).
25. Glover, F., Tabu Search: Part 2, *ORSA J. Comput.* 2(1):4–32 (1989).
26. Hertz, A. and de Werra, D., Using Tabu Search for Graph Coloring, *Comput.* 39:345–351 (1987).
27. Harralick, R. and Elliot, G., Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence* 14:263–313 (1980).
28. Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.
29. Van Hentenryck, P., The Cardinality Operator: A New Logical Connective for Constraint Logic Programming, in: *Eighth International Logic Programming Conference*, Paris, 1991.
30. Van Hentenryck, P., Scheduling and Packing in the Constraint Language cc(FD), Technical Report, CS Department, Brown University, 1992.
31. Van Hentenryck, P., Saraswat, V., and Deville, Y., Implementation and Evaluation of the Constraint Language cc(FD), Technical Report, CS Department, Brown University, 1992.
32. Jaffar, J. and Maher, M. J., Constraint Logic Programming: A Survey, *J. Logic Programming*, pp. 503–581, 1994.
33. Jourdan, J. and Sola, T., The Versality of Handling Disjunctions as Constraints, in: *PLIP'93*, 1993.
34. Lim, P. and Jourdan, J., Workshop on Constraint Languages and Their use in Problem Modelling, in: *ILPS'95*, Ithaca, New York, 1994.
35. Menez, F., Barahona, P., and Codognet, P., An Incremental Constraint Solver Applied to a Time-Tabling Problem, in: *Thirteenth Conference on Expert Systems*, Avignon France, 1993.
36. Le Pape, J. P. and Ranson, D., Clef or Programming with Constraints, Logic, Equations and Functions, in: *4th International Conference on Software Engineering and its Applications*, Toulouse, France, 1991.

37. Rueher, M. and Legeard, B., Which Role for CLP in Software Engineering? An Investigation on the Basis of First Applications, in: *First International Conference on Practical Applications of Prolog*, London, 1992.
38. Rueher, M., A First Exploration of PrologIII's Capabilities, *Software—Practice and Experience* 23(2):177–200 (1993).
39. Simonis, H. and Cornelissens, T., Modelling Producer/Consumer Constraints, in: *Workshop on Constraint Languages and Their use in Problem Modelling, ILPS'95*, Ithaca, New York, 1994.
40. Rayward-Smith, V. J. and Shing M. T., Bin-Packing, *Bulletin of the IMA* 19:142–146 (1983).
41. Tripathy, A., School Timetabling: A Case in Large Binary Integer Linear Programming, *Management Sci.* 30(12):1473–1489 (1984).
42. Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press, New York, 1993.
43. Wood, D. C., A System for Computing University Examination Timetables, *Comput. J.*, pp. 41–47, 1968.