Programmed Search in a Timetabling Problem over Finite Domains[1]

R. González-del-Campo[2] F. Sáenz-Pérez[3]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

Labeling is crucial in the performance of solving timetabling problems with constraint programming. Traditionally, labeling strategies are based on static and dynamic information about variables and their domains, and selecting variables and values to assign. However, the size of combinatorial problems tractable by these techniques is limited. In this paper, we present a real problem solved with constraint programming using programmed search based on the knowledge about the solution structure as a starting point for classical propagation and labeling techniques to find a feasible solution. For those problems in which solutions are close to the seed because of its structure, propagation and labeling can reach a first solution within a small response time. We apply our approach to a real timetabling problem, and we tackle its implementation with two different languages, OPL and $\mathcal{TOY}$, using the constraint programming paradigm over finite domains. While OPL is a commercial, algebraic, and specific-purpose constraint programming language, $\mathcal{TOY}$ is a prototype of a general-purpose constraint functional logic programming language. We present the specification of the problem, its implementation with both languages, and a comparative performance analysis.

Keywords: Finite Domains, Search, Applications, Timetabling

1 Introduction

In the last years, the number of applications of timetabling has grown spectacularly. Timetabling [6] refers to the allocation, subject to constraints, of given resources to objects being placed in space-time, in such a way as to satisfy as nearly as possible a set of desirable objectives (also known as the cost function). Timetabling problems are NP-complete and, therefore, these problems have been usually tackled with four techniques: evolutionary computing [25,15], integer programming [13], constraint programming [18], and constraint logic programming [19]. Evolutionary computing is based on rules simulating natural evolution and solutions are stochastically looked for, reaching *reasonable* solutions but, in general, not optimal w.r.t. a cost function [28]. In addition, problem formulation lacks of a clear reading. See, for instance,

[1] This work has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.

[2] Email: rgonzale@estad.ucm.es

[3] Email: fernan@sip.ucm.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the works [27,26,9,10] that apply this technique to timetabling. Both integer [33] and constraint programming [23] applied to timetabling problems in particular can reach optimal solutions w.r.t a cost function, and problem formulation becomes algebraic, which is a very abstract programming paradigm, but they lack the benefits of a general purpose programming language. OPL [30] is an outstanding example of a (commercial) constraint programming language with a quite effective state-of-the-art constraint solver. These timetabling problems have also been formulated under the constraint logic programming paradigm [1,21], and the advantages coming out from both its declarative nature and general purpose approach makes them more amenable for problem solving. In addition, optimal solutions can be found in this paradigm (as well as in constraint programming) because, besides they enjoy efficient solution space cutting due to concurrent propagators, a complete enumeration procedure can be applied.

Another step beyond declarative languages has been raised with the integration of higher order functions, lazy evaluation, types and partial applications into constraint logic programming, giving as a result the constraint functional logic programming paradigm [22]. Examples of this paradigm are $\mathcal{TOY}$ [12] and Curry [16], which are equipped, in particular, with finite domain solvers adequated for the formulation of timetabling problems. We can see an example of applications of the constraint functional logic programming paradigm in timetabling problems in [2].

Finite domain constraint solvers usually found in constraint languages (both algebraic or general purpose languages) include labeling strategies for enumerating the search space. These strategies are based on static and dynamic information about variables and their domains, and the selection of variables and values to assign. Because of their enumeration nature, the size of the combinatorial problems which can be tackled with such an approach is limited. In order to overcome this situation, incomplete enumeration strategies are proposed [31,32], trying to find feasible (but no necessarily optimal) solutions. In many cases, this is a suitable approach since one is interested in quickly finding a reasonable solution. However, one can also be interested on being able to find all the feasible solutions, although insisting on a quick approximation to the optimum. An example of this application is the finding of timetables for a company staff.

In this paper, we perform a programmed search based on the knowledge about the solution structure. It amounts to generate a seed [24], as a case of local search space pruning [5,29,4,7]. This search consists of a fast generation of a seed which will be used as a starting point for the classical propagation and labeling techniques present in constraint solving (see also [20,8]). In contrast to other approaches, as stochastical (such as evolutionary computing and simulated annealing), we will be able to quickly find a first solution but retaining the ability of searching the whole solution space by using the efficient constraint solving classical techniques (i.e., propagation and labeling). We can apply this technique for those problems in which solutions are close to the seed because of the problem structure itself. Incidentally, this is the case of real problems with a regular solution structure, as we have found in a particular company staff timetabling. In this problem, solutions occur close in the search space in the sense that, in general, few variables need to be assigned to distinct values for subsequent solutions, and we conjecture that our

approach could be applied to other problems showing the same property.

In addition, we apply our approach to this real case using two constraint systems: OPL, a commercial, algebraic, and specific-purpose constraint programming language, and $\mathcal{TOY}$, a prototype of a general-purpose declarative constraint programming language. We test and compare the performance of both programming systems solving this problem over finite domain constraints [4].

This paper is organised as follows. Section 2 presents the specification of the timetabling problem in the concrete real case we faced. In Section 3, we describe our approach to the programmed search. Section 4 and 5 introduce, respectively, an outline of the implementation with both OPL 3.7 and $\mathcal{TOY}$ 2.1.0 systems. Section 6 resumes the performance results during a calendar up to one year, comparing and analysing the results from both systems. Finally, in Section 7, we summarise some conclusions and propose future work.

2 A Real-Case Problem

We were faced to this problem during our professional service in the data processing department of a big national company (for which we omit its name and concrete data for the problem because of confidential issues), in which the problem of finding feasible assignments for workers in the working calendar revealed as a complex task. This company offers a continuous service in a given department to fulfill an annual agenda. There are thirteen workers which are organised by four teams of three workers with skills enough to provide the service. There is also an extra worker (a joker) for coping with incidents, which may be because of holidays or other absences (sick leaves, maternity leaves, and so on). These workers present different qualification levels and, therefore, an adequate assignment is needed for a team or part of a team in order to have enough abilities to fulfill their duties. In addition, there are some possible time slots that workers can be assigned to, that tightly depend on the needs of the company. In particular, we can find the following time slots:

- T1: 8:00 to 8:00 (24 working hours)
- T2: 8:00 to 22:00 (14 working hours)
- T3: 18:00 to 8:00 (14 working hours)
- T4: 15:00 to 8:00 (17 working hours)
- T5: 8:00 to 21:00 (13 working hours)
- T6: 8:00 to 14:00 (6 working hours)

Each worker works during a time slot. With respect to worker qualification, we find two levels: workers with level 1 are experienced and can be workers in charge on all time slots. Workers with level 2 are apprentices and cannot be in charge of a team. In every team there is a worker with qualification level 2 and two workers with qualification level 1. A worker which has been working during a night must rest during the next two working days, at the least. The number of working hours

[4] For a comparative analysis of several finite domain constraint programming systems, see [12].

during a year is established by a working calendar, and there are maximum and minimum limits, both monthly and annual, over working hours which cannot be violated. In every team, only one worker can simultaneously enjoy holidays.

Workers have to be assigned to time slots during the working calendar, which usually extends to one year, although plannings can observe shorter time intervals. Usually, a team works every four days. In a working day, there should be three workers available. Every worker has to be assigned to a different time slot (T1, T2 or T3). The joker has assigned the time slot T6 in absence of incidents. Saturdays and holidays feature two workers available with time slot T1 and the extra worker does not work. Time slots rotate for the workers in a team each time they complete a time slot. December 24^{th} and 31^{st} are special days without continuous service in which there must be two workers available, every one gets the time slot T5, and at least one worker must have a qualification level 1. When an incident happens because a worker is absent, and if the joker is available, then the joker replaces the absent. Otherwise, only two workers will cover the absence with time slots T1 or T4.

3 Seed Generation and Programmed Search

As stated in the introduction, in order to gain performance in the search-for-solutions process, we quickly generate a seed which is not expected to fulfill all the constraints imposed by the problem, and then we apply classical propagation and labeling techniques. The idea is to generate an assignment for the decision variables present in the implementation of the problem such that the solution structure is observed. This means for our particular problem that we assign rotating time slots to each member of all the teams each four days in a consistent way with the working calendar, and ignoring some other constraints. Although this seed may not meet all the constraints, such as the limits imposed on the maximum number of working hours during the planning, it behaves as a good starting point for the classical constraint solving techniques to find a first solution.

The procedure to develop the seed is to assign the time slots T1, T2 and T3 to workers for working days, and T1 to two workers of the same team if either the day is Saturday or there is a known incident. Time slots rotate next days. Then, we assign incidents to workers. In such a way, the number of working hours of each worker is uniform along the planned calendar. If there are few incidents, we have found that the seed is close to a feasible solution because the labeling strategy finds a solution by processing a few nodes in the search tree.

Once this first assignment is done, a feasible solution will be hopefully close to a solution. If so, the process of labeling will have few failures and the first solution will be met within a small running time, which is the case of our problem, as we will see in Section 6. After the initial assignment of the seed has been applied and the first solution found, we develop a search based on the remaining variables in the corresponding domains in an ordered way, i.e., observing the problem structure and imposing disequality constraints on values found to be a solution so far.

4 Implementing with OPL

OPL [17] is a commercial and specific-purpose programming language, which was motivated by modeling languages such as AMPL [14] and GAMS [3] that provide computer equivalents to the traditional algebraic notation. It provides similar support for modeling linear, integer, and constrained programs. OPL adds support for constraint programming and complex structures of data: arrays, records, variables over enumerated sets, and variables and expressions as indices. In addition, OPL provides predefined search procedures as well as constructs for defining user-defined search strategies.

OPL programs must conform with a sectioned arrangement in which several sections are included: data initialisation, decision variable and constraint declarations, and search procedures. In the section intended for data initialisation, all data parameters needed for posing the constraints are declared and initialised. For example, the declaration `int+ totalHours` below is the number of hours in a timetable with the exact working hours. `int+ variation= ...;` is the variation allowed in the number of hours of each worker, where dots (`...`) indicate that variables are assingned from a data file. `int+` stands for the type of positive integers. `timeSlotDuration[timeSlots]` is an array with duration of time slots. The enumeration `workersRange` represents all the workers, and `daysRange`, the working calendar. `timeSlots` is an integer range that represents time slots. Time slots and teams are represented by subranges of integers.

```
int+ totalHours = ...;
int+ variation = ...;
range durationRange 0..24;
durationRange timeSlotDuration[timeSlots] = ...;

timeSlots new [workersRange, daysRange];
```

In the declaration section, we specify decision variables and constraints. For instance, the timetable is represented by the two-dimensional array `timetable` with elements of type `timeSlots`, which ranges over the subrange of integers 1 to 6, denoting the possible time slots. The prefix `var` indicates that `timetable` is a decision variable.

```
var timeSlots timetable [workersRange, daysRange];
```

We then impose constraints over this two-dimensional decision array, and show, as an example, how the constraint about the variation limit on working hours is posed.

```
forall (t in workersRange) {
 abs (sum (d in daysRange)
           timeSlotDuration[timetable[t,d]] - totalHours)
 <=
 variation;
};
```

which is intended to be equivalent to the following algebraic expression:

$$\forall_{t \in workersRange} \\ |\Sigma_{d \in daysRange}(timeSlotDuration[timetable[t,d]]) - totalHours| \\ \leq variation$$

Observe that, in this OPL formulation for this constraint, we have used decision variables as array indexes.

The seed is generated in an auxiliary two-dimensional array `new` of the same type and size as `timetable`, which will hold all the assigments for decision variables.

The next code fragment shows how the searching is implemented in the OPL section devoted to user-defined search procedures. It features some (reflection) functions as `dsize(v)`, which returns the size of the domain of `v`. `bound(v)` is true if the domain of `v` is a singleton. `dmin` returns the minimum value in the domain of `v`. `let m = expression` assigns to `m` the value computed for `expression`. `try v = value 1 | v = value 2` assigns to `v` the value `1` and adds this assignment to the constraint store together with a choice point. On backtracking, `v` is assigned to the value 2 and the choice point is removed.

The search procedure listed below includes these sentences and reflection functions, and it implements the building of the whole search tree.

```
search {
 forall (t in workersRange)
  forall (d in daysRange)
   if dsize(timetable[t,d]) > 1 then
    try timetable[t,d] =  new[t,d] |
       {timetable[t,d] <> new[t,d];
        while not bound(timetable[t,d]) do
         let m = dmin(timetable[t,d]) in
          try timetable[t,d] =  m |
              timetable[t,d] <> m
          endtry;}
    endtry
   endif;
};
```

The OPL program implementing the problem specification has 615 code lines.

5 Implementing with TOY

$\mathcal{TOY}$ is an implementation of a constraint functional logic language which enjoys, in particular, finite domain constraints. This language adds the power of constraint programming over finite domains to the characteristics of functional logic programming. $\mathcal{TOY}$ increases the expressiveness and power of constraint logic programming over finite domains (CLP($\mathcal{FD}$)) by combining functional and relational notation, curried expressions, higher order functions, patterns, partial applications, non-determinism, constraint composition, lazy evaluation, logical variables, types, domain variables, constraints, and constraint propagators. $\mathcal{TOY}$ combines both the efficiency and expressiveness of CLP($\mathcal{FD}$) with new features not existing in

CLP($\mathcal{FD}$) that contribute to increase the expressiveness of constraint declarative languages. Its basic data structure is the list, which is specified as in Prolog, and its elements can only be sequentially accessed. $\mathcal{TOY}$ programs include data type declarations and defined functions, but do not present a sectioned arrangements as OPL programs.

A timetable is represented by a list of lists of decision variables, each one of type `t_timeSlot`. The type for the timetable (`t_planificacion`) is then declared as:

```
type t_timetable = [t_worker]
type t_worker = [t_timeSlot]
type t_timeSlot = int
```

`[T]` denotes a list with elements of type `T`, and each decision variable (a cell in the timetable that represents an assignment for a given worker and day) is of type integer, as the above declaration states, instead of a proper subrange. This subrange is limited by constraining each domain variable, a task that needs to be performed by sequential access to the list of lists, in contrast to OPL, which allows a direct access to each decision variable. However, as constraints are posted sequentially in our problem, the timetable does not need a random access.

The seed is generated again in an auxiliary list of lists of parameters, instead of an array as before, with the same type and size as timetable (a list of lists of decision variables), which will hold all the assigments. Note that neither a timetable nor its seed is declared as a data structure for the decision variables, but it is created at run-time during narrowing as an argument of the main function which implements the timetabling procedure. Again, this is in contrast to OPL, in which a static memory assignment is performed.

As an example of implementing constraints over the list of lists of decision variables, we return to the limitation about working hours during the calendar. As the example in the previous section, we want this number of hours to be inside a given interval. In the following code fragment, `workerHours D` is applied over a vector which is the list of the time slots for all the days in the working calendar, and returns an integer which is the total sum of hours worked by the worker. This function uses another one, `duration D`, which returns the hours corresponding to a given time slot `D`. The function `yearHours` posts constraints about the limits of exceeded working hours. It takes the timetable as a first argument, `M` as an input parameter representing the number of working hours in the calendar, and `R` as also an input parameter representing the allowed variation in the number of worked hours along the calendar.

```
workerHours :: t_worker -> int workerHours W = foldl (#+) 0 (map
duration W)

yearHours :: t_timetable -> int -> int -> bool yearHours [] M R =
true yearHours [T|Ts] M R = true
      <== workerHours T #> (M-R), workerHours T #< (M+R),
          yearHours Ts M R
```

Durations for each time slot are represented by the function `duration`, instead of the array `timeSlotDuration[timeSlots]` indexed by constraint variables as used

in OPL. In addition, we implement two versions of this function for comparing its readability and performance in order to analyse the trade-off between such factors. The first implementation is shown below and uses arithmetical constraint operators:

```
duration:: int -> int duration T = m0_0 T #+ m1_24 T #+ m2_14 T #+
m3_14 T#+
            m4_17 T #+ m5_13 T #+ m6_6 T #+ m7_6 T #+ m8_0 T
```

We show a case of the functions involved in `duration`, which are intended to compute the duration of a given time slot:

```
m4_17:: int -> int m4_17 T = 17#*T#*(T #- 1)#*(T #- 2)#*(T #-
3)#*(5#- T)#*
                (6 #- T)#*(7 #- T)#*(8 #- T)#/576
```

The second implementation involves two non-existing propositional constraint operators in $\mathcal{TOY}$ version 2.1.0, namely implication (`#=>`) and disjunction (`#\/`), so that we have implemented them into the system.

```
duration:: int -> int duration T = D <== (((((T #= 1) #=> (D #= 24))
#\/
                        ((T #= 2) #=> (D#= 14)))   #\/
                       (((T #= 3) #=> (D #= 14))   #\/
                        ((T #= 4) #=> (D #= 17)))) #\/
                      ((((T #= 5) #=> (D #= 13))   #\/
                        ((T #= 6) #=> (D #= 6)))   #\/
                       (((T #= 7) #=> (D #= 6))    #\/
                        ((T #= 8) #=> (D #= 0))))) #\/
                        ((T #= 0) #=> (D #= 0))
```

The generation of the seed in $\mathcal{TOY}$ is similar to OPL, but we make the elements of the list `[V|Vs]` to be assigned to suitable values. The list `[X|Xs]` of lists of finite domains variables is assigned to the seed list.

In the following code fragment, `remove V List` removes `V` from its second argument (`List`). `fromXuntilY V W` generates a list with all values between `V` and `W`. The reflection function `fd_min V` returns the minimum value in the domain of `V`, whereas `fd_max V` returns the maximum. `rest V W` removes the value `W` from the domain of the decision variable `V`. `generate_list X V` generates a list of values including the value `V` and all the values in the domain variable `X`, assumed that maybe `V` is not a feasible assignment for `X`. The first element of the generated list is the value of the seed for `X`. `try V [W|Ws]` tries, by backtracking, to label the decision variable `V` with every value `W` of its second argument. `my_search [X|Xs] [V|Vs]` tries to assign each value `V` in the list, which is in its first argument, to each corresponding decision variable `X`, which is in its second argument. `++` is the list concatenation operator.

```
rest :: int -> int -> [int] rest X V = remove V (fromXuntilY
(fd_minX) (fd_max X))
```

```
generate_list :: int -> int -> [int] generate_list X V = [V] ++ rest
X V

try :: int -> [int] -> bool try X [V|Vs] = true <== X==V try X
[V|Vs] = true <== try X Vs

my_search :: [int] -> [int] -> bool my_search [] [] = true my_search
[X|Xs] [V|Vs] = true
  <== try X (generate_list X V),
      my_search Xs Vs
```

The $\mathcal{TOY}$ program implementing the problem specification has 1,010 code lines, which represents an excess of about a forty percent compared to the OPL program that implements the same functionality. We find that OPL is in particular more suitable to express algebraic expressions implementing constraints than $\mathcal{TOY}$ since it allows a more compact formulation of the problem.

6 Performance Analysis

In this section, we show the performance results we have obtained for finding the first solution of the stated real problem as implemented in the systems $\mathcal{TOY}$ 2.1.0 and OPL 3.7, both running on a Pentium III at 1 GHz with 256 Mb of RAM and Windows 2000 Professional. We have considered several calendar sizes, ranging from a week to a year, and also we consider built-in search strategies of these languages in order to compare with our programmed search based on the generation of a seed. We have obtained running times for these parameters as the average of four runs.

Table 1 shows these results and has several columns: The column Size represent the size of the problem in terms of the number of months of the timetable. The column TO stands for the labeling strategy equally specified in both systems that assigns values to variables using their textual (static) order, and its possible values in ascending order. The column FF stands for the first-fail strategy. The column FFC stands for the first-fail strategy considering suspended constraints. The column S-B stands for the Slice-Based strategy. The column D-B stands for the Depth-Bounded strategy. The last column, ES, stands for our proposal, a programmed search strategy. Each cell in the table shows up to three values separated by a slash (/): The first value indicates the execution time in seconds for the test run under OPL, the second value indicates the same for $\mathcal{TOY}$, and the third value, if present, the speed-up of OPL w.r.t. $\mathcal{TOY}$. A dash (–) instead of a value represents that the test could not be done because the corresponding strategy is not implemented in the system. An infinite symbol (∞) means that the elapsed time for finding a solution is greater than one day. The execution time for generating the seed is included in ES.

From the numbers above we check that our proposal performs better than the rest of labeling strategies in all cases. Classical search procedure do not even find a solution within a reasonable time (one day of computing). Although not shown in the tables, alternative solutions are found without a noticeable delay, which

Size	TO	FF	FFC	S-B	D-B	ES
1/4	0.04/0.54/12.2	–/0.52	–/0.55	0.04/–	0.04/–	0.05/0.76/15.2
1/2	0.07/1.09/14.7	–/1.11	–/1.06	0.07/–	0.07/–	0.09/1.41/15.6
1	0.87/13.77/15.9	–/18.85	–/3.84	0.91/–	0.90 /–	0.19 /2.78/14.6
2	12.15 / 209.99/17.2	–/309.62	–/27.23	12.68/–	12.52/–	0.31/5.86/18.9
3	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	0.41/10.00/24.4
4	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	0.53 /14.89/28.1
5	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	0.65 /20.84/32.1
6	∞ / ∞	–/∞	–/∞	∞/–	∞ /–	0.77 /27.43/35.6
8	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	0.98 /43.52/44.4
10	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	1.25 /63.56/50.9
12	∞ / ∞	–/∞	–/∞	∞ /–	∞ /–	1.69 /86.58/51.2

Table 1
Programmed Search vs. Classical Constraint Solving for OPL and $\mathcal{TOY}$

indicates the locality of solutions. However, for small problem sizes (less than one month), classical strategies are slightly better than explicit search.

If, on the other hand, we compare the execution times for OPL and $\mathcal{TOY}$, we find that programmed search in OPL behaves better than in $\mathcal{TOY}$ because OPL has several features which are not present is $\mathcal{TOY}$ and makes it more appropriate for performance. For instance, OPL has conditional and disjunction constraint operators, static decision and data variables with direct memory access, and arrays. To overcome the drawbacks derived from the absence of such features, the $\mathcal{TOY}$ program has to rely on building structures (lists of lists, in particular) with sequential access by means of recursive functions.

In order to identify in more detail the factors intervening in the total goal solving time for both systems, we have accounted for the ones shown in Table 2. The column "Size" stands for the size of the problem, as before. Next column, "Data Structure", stands for the time involved in computing the data structures. The column labeled with "Seed" stands for the time employed in building the seed. The columns "Posting and Propagation" and "Labeling" show the time for these processes, whereas the last column shows the total time. The cells in the table follow the same data format as Table 1. Note that there are times shown as 0.00, which means that the time measure is less than 0.01 seconds.

Size	Data Structure	Seed	Posting and Propagation	Labeling	Total
1/4	0.00/0.13/–	0.00/0.16/–	0.05/0.47/9.44	0.00/0.00/–	0.05/0.76/15.2
1/2	0.00/0.22/–	0.00/0.23/–	0.09/0.89/9.89	0.00/0.07/–	0.09/1.41/15.6
1	0.00/0.30/–	0.01/0.48/48.4	0.18/1.87/10.4	0.00/0.13/–	0.19 /2.78/14.6
2	0.00/0.46/–	0.02/1.11/55.5	0.29/3.93/13.6	0.00/0.36/–	0.31/5.86/18.9
3	0.00/0.63/–	0.05/2.07/41.3	0.37/6.85/19.0	0.00 /0.46/–	0.41/10.00/24.4
4	0.00/0.77/–	0.06/3.23/53.8	0.46/10.20/22.7	0.02/0.69/34.5	0.53 /14.89/28.1
5	0.00/0.86/–	0.08/4.89/61.1	0.53/14.40/27.2	0.04/0.70/19.3	0.65 /20.84/32.1
6	0.00/1.06/–	0.11/6.57/59.7	0.62/18.77/30.3	0.04/1.05/26.3	0.77 /27.43/35.6
8	0.00/1.37/–	0.16/10.98/68.6	0.81/29.81/39.7	0.07/1.36/19.4	0.98 /43.52/44.4
10	0.00/1.66/–	0.22/16.38/74.4	1.11/43.03/46.8	0.11/2.50/22.7	1.25 /63.56/50.9
12	0.00/1.93/–	0.28/22.98/82.1	1.27/58.98/46.4	0.14/2.70/19.3	1.69 /86.58/51.2

Table 2
Factors involved in Goal Solving for OPL and $\mathcal{TOY}$

Building structures in OPL is negligible, whereas $\mathcal{TOY}$ takes as much as almost 2 seconds. In this last system, seed generation quickly grows with problem size, more than the former, which means that the specific-purpose algorithms in OPL to build structures behave better than the general-purpose computation performed in $\mathcal{TOY}$. Posting and propagating constraints also grow, but the gain of OPL w.r.t. $\mathcal{TOY}$ is less than before. The gain of the labeling is, in the average, about 23.5, with small deviations. The last column shows the same data as Table 1 and is kept for reference.

Next, Table 3 shows the impact of propagation alone over the total computation time for both systems. This table highlights the power of the underlying constraint solver. In particular, the second column shows a maintained gain of about 10.3 in the average, showing that the growing gain of OPL w.r.t. $\mathcal{TOY}$ noticed in former tables is not due to propagation, but for the nature of the declarative language involving less efficient data structures and the lazy narrowing mechanism inherent to the system. This table also contains the number of constraints (which is the same for the former tables) and this number is about 22 in the average.

Finally, in Table 4 we compare the two implementations of the function `duration` in $\mathcal{TOY}$, measuring the timings for several factors: posting and propagation, propagation, and propagation for the function vs. total propagation. In this table, the format of timing cells changes as follows. There are three values separated by a slash (/): The first value indicates the execution time in seconds for the first implementation of `duration` with arithmetical operators, the second value indicates the same for its implementation with propositional constraint operators, and the third value, the speed-up of the first implementation w.r.t. the second one.

We note that the first implementation behaves better than the second in about 19 percent of total time. Also, posting and propagating constraints is about 28 percent better. Propagation time grows up to almost 5 percent. The ratio of propagation time of the constraints due to `duration` w.r.t. total propagation time is constant. In the first case it is 0.38, whereas in the second case is 0.67. There is about a 10 percent more constraints in the second case. The ratio of the number of constraints due to `duration` w.r.t. total number of constraints is constant. In the first case is 0.93, whereas in the second case is 0.94. Although the second implementation behaves worse than the first one, the additional costs may be accepted in favour of

Size	Propagation	Rest of Program	Number of Constraints	Propagation/Total
1/4	0.03/0.25/8.33	0.02/0.51/25.5	585/10,638/18.18	0.60/0.34
1/2	0.06/0.56/9.33	0.03/0.85/28.33	1,047/22,741/21.72	0.67/0.41
1	0.14/1.02/7.29	0.05/1.77/35.40	2,163/46,948/21.71	0.73/0.38
2	0.22/1.95/8.86	0.09/3.91/43.44	4,011/89,278/22.26	0.71/0.37
3	0.25/2.80/11.2	0.16/7.20/45.00	6,057/136,138/22.48	0.61/0.33
4	0.31/3.81/12.29	0.22/11.09/50.41	8,037/181,428/22.57	0.58/0.30
5	0.37/4.90/13.24	0.28/ 15.95/56.96	10,083/228,360/22.65	0.57/0.28
6	0.46/5.16/11.21	0.31/22.27/71.84	12,063/273,686/22.69	0.60/0.25
8	0.57/6.35/11.14	0.41/37.17/90.65	16,155/367,424/22.74	0.58/0.22
10	0.74/8.41/11.36	0.51/55.15/108.14	20,181/459,646/22.78	0.59/0.21
12	1.05/9.12/8.69	0.64/77.47/121.05	24,207/551,850/22.80	0.62/0.19

Table 3
Propagation vs. Goal Solving for OPL and $\mathcal{TOY}$

Size	Total	Posting and Propagation	Propagation	Propagation for `duration` / Total Propagation	Number of Constraints
1/4	0.76/0.88/1.16	0.47/0.59/1.25	0.25/0.52/2.08	0.51/0.78	10,638/11,824/1.11
1/2	1.41/1.68/1.19	0.89/1.16/1.30	0.56/1.08/1.95	0.48/0.73	22,741/25,488/1.12
1	2.78/3.44/1.24	1.87/2.53/1.35	1.02/2.21/2.17	0.31/0.75	46,948/51,656/1.10
2	5.86/7.15/1.22	3.93/5.23/1.32	1.95/4.42/2.26	0.31/0.74	89,278/98,322/1.10
3	9.99/11.96/1.20	6.85/8.81/1.28	2.80/6.92/2.47	0.35/0.69	136,138/150,046/1.10
4	14.89/17.52/1.18	10.20/12.83/1.26	3.81/9.76/2.56	0.32/0.67	181,428/200,064/1.10
5	20.84/24.24/1.16	14.40/17.80/1.24	4.89/13.08/2.67	0.34/0.62	228,360/251,560/1.10
6	27.43/31.67/1.15	18.77/23.01/1.23	5.16/16.09/3.12	0.36/0.63	273,686/301,878/1.10
8	43.52/54.53/1.25	29.81/40.82/1.37	6.35/28.51/4.49	0.37/0.65	367,424/407,841/1.11
10	63.56/76.01/1.20	43.03/54.47/1.27	8.41/37.40/4.45	0.42/0.59	459,646/505,611/1.10
12	86.58/100.77/1.16	58.98/73.17/1.24	9.12/43.90/4.82	0.45/0.54	551,850/612,554/1.11

Table 4
Comparing both Implementations of the Function `duration` in $\mathcal{TOY}$

a more readable implementation.

7 Conclusions and Future Work

Traditionally, labeling strategies are based on static and dynamic information about variables and their domains, and selecting variables and values to assign. However, this information is not sufficient for many hard problems to be tractable. Labeling produces many fails during searching for solutions and the response time grows exponentially with problem size. With a programmed search based on the knowledge about the program structure, a seed close to a solution can be found in a reasonable time, which means than the enumeration strategy produces few fails. The key question is whether one can find close solutions in the problem, which strongly depends on the solution structure, a point that should be eventually addressed.

In this work, two of the best state-of-the-art constraint programming systems (in their corresponding settings) have been taken into account for implementing the specification of a real problem. From the performance results we have found that the average time for finding the first solution is low compared to classical techniques in the field of constraint solving, even if the seed is not a solution. It is therefore not necessary to specify a first solution to depart from in our searching proposal. The execution time becomes moderate with few different values in variable domains. OPL gives responses faster than $\mathcal{TOY}$ because $\mathcal{TOY}$ version 2.1.0 did not enjoy key features present in OPL as arrays indexed by decision variables. Implication and disjunction constraints were also not included, and we have implemented them, showing that their use augments program readability and introducing a reasonable burden.

Although OPL behaves clearly better than $\mathcal{TOY}$, this system enjoys a more homogeneous syntax for solving problems in the sense that the same program constructs are used to generate the seed, post constraints, and specify the search strategy. That is, there is no the impedance mismatch that can be found in OPL when

used from a host language. OPL, in turn, has three sections in a program with isolated syntaxes: initialisation of data, decision variable and constraint declarations, and search procedures section (among others such as database handling). We have found that the implementation of the problem is easier in a language as $\mathcal{TOY}$ since it seamlessly embodies constraints into a very expressive general purpose language because of its declarative nature. In addition, the propagation solver for the $\mathcal{TOY}$ underlying system behaves reasonable fine w.r.t. the solver of OPL. Finally, while OPL is a commercial system, $\mathcal{TOY}$ is for free.

Some lines we emphasise as being amenable to explore as future work are: First, the inclusion of the array data structure with direct access on its elements, along the possibility to index such an array by means of decision variables. Second, a memory usage analysis (including garbage collection) in the context of a complex operating system. Finally, an algebraic component should be added to the language in order to be able to compactly declare constraints and decision variables. The algebraic notation would allow more compact programs, whereas (static) decision variable declarations would do for faster memory allocations.

References

[1] Azevedo, F. and P. Barahona, *Timetabling in Constraint Logic Programming*, in: *Proceedings of 2nd World Congress on Expert Systems*, Estoril, Portugal, 1994.

[2] Brauner, N., R. Echahed, G. Finke, H. Gregor and F. Prost, *Specializing narrowing for timetable generation: A case study.*, in: *PADL*, 2005, pp. 22–36.

[3] Brooke, A., D. Kendrick and A. Meeraus, *GAMS: A User's Guide* (1992).

[4] Burke, E. and J. Landa Silva, *The design of memetic algorithms for scheduling and timetabling problems*, in: S. J. Krasnogor N., Hart W., editor, *Recent Advances in Memetic Algorithms*, 2004, pp. 289–312.

[5] Burke, E. and S. Petrovic, *Recent Research Directions in Automated Timetabling*, European Journal of Operational Research **140** (2002), pp. 266–280.

[6] Burke, E. K., K. Jackson, J. H. Kingston and R. F. Weare, *Automated University Timetabling: The State of the Art*, Comput. J. **40** (1997), pp. 565–571.

[7] Burke, E. K., J. P. Newall and R. F. Weare, *A Memetic Algorithm for University Exam Timetabling*, in: *Practice and Theory of Automated Timetabling. Volume 1153 of Lecture Notes in Computer Science*, Lecture Notes in Computer Science **1153** (1995), pp. 241–250.

[8] Castro, C., M. Moossen and M. Riff, *A Cooperative Framework Based on Local Search and Constraint Programming for Solving Discrete Global Optimisation*, in: *Advances in Artificial Intelligence SBIA 2004* (2004), pp. 93–102.

[9] Corne, D., P. Ross and H.-L. Fang, *Evolutionary Timetabling: Practice, Prospects and work in Progress*, in: *Proceedings of the UK Planning and Scheduling SIG Workshop Strathclyde*, 1994.

[10] Corne, D., P. Ross and H.-L. Fang, *Fast Practical Evolutionary Timetabling*, in: *Lecture Notes in Computer Science, vol 865 (Evolutionary Computing AISB Workshop, Leeds, UK, April 1994)* (1994), pp. 251–263.

[11] Díaz, A., "Optimización Heurística y Redes Neuronales," Editorial Paraninfo, 1996.

[12] Fernández, A. J., T. Hortalá-González, F. Sáenz-Pérez and R. del Vado, *Constraint functional logic programming over finite domains*, Theory and Practice of Logic Programming (2006), in Press.

[13] Garfinkel, R. and G. Nemhauser, "Integer Programming," John Wiley & Sons, New York, 1972.

[14] Gay, D. M., *Symbolic-Algebraic Computations in a Modeling Language for Mathematical Programming.*

[15] Goldberg, D. E., "Genetic Algorithms in Search, Optimization and Machine Learning," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[16] Hanus, M., *Curry: a Truly Integrated Functional Logic Language* (1999), http://www.informatik.uni-kiel.de/∼curry/.

[17] Hentenryck, P. V., L. Michel, L. Perron and J.-C. Régin, *Constraint Programming in OPL*, in: G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Lecture Notes in Computer Science **1702**, 1999, pp. 98–116.

[18] Hentenryck, P. V. and V. Saraswat, *Strategic Directions in Constraint Programming*, ACM Comput. Surv. **28** (1996), pp. 701–726.

[19] Jaffar, J. and J. Lassez, *Constraint Logic Programming*, in: *14th ACM Symposium on Principles of Programming Languages (POPL'87)* (1987), pp. 111–119.

[20] Khemmoudj, M., M. Porcheron and H. Bennceur, *Using Constraint Programming and Local Search for Scheduling of Electricité de France Nuclear Power Plant Outages* (1998).

[21] Lajos, G., *Complete University Modular Timetabling using Constraint Logic Programming*, in: *Selected papers from the First International Conference on Practice and Theory of Automated Timetabling* (1996), pp. 146–161.

[22] López-Fraguas, F. J., *A General Scheme for Constraint Functional Logic Programming*, in: *Proceedings of the Third International Conference on Algebraic and Logic Programming* (1992), pp. 213–227.

[23] Marte, M., "Models and Algorithms for School Timetabling - A Constraint-Programming Approach," Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich (2003).

[24] Merlot, L. T. G., N. Boland, B. D. Hughes and P. J. Stuckey, *A Hybrid Algorithm for the Examination Timetabling Problem*, in: *Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling. Volume 2740 of Lecture Notes in Computer Science*, Lecture Notes in Computer Science **2740** (2002), pp. 207–231.

[25] Michalewicz, Z., "Genetic Algorithms + Data Structures = Evolution Programs (3rd ed.)," Springer-Verlag, London, UK, 1996.

[26] Ross, P., D. Corne and H.-L. Fang, *Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation*, in: Y. Davidor, H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN III* (1994), pp. 556–565.

[27] Ross, P., D. Corne and H.-L. Fang, *Successful Lecture Timetabling with Evolutionary Algorithms*, in: A. E. Eiben, B. Manderick and Z. Ruttkay, editors, *Applied Genetic and other Evolutionary Algorithms: Proceedings of the ECAI'94 Workshop*, Springer, Berlin, 1995 .

[28] Ross, P., E. Hart and D. Corne, *Some Observations about GA-Based Exam Timetabling*, in: *PATAT '97: Selected papers from the Second International Conference on Practice and Theory of Automated Timetabling II* (1998), pp. 115–129.

[29] Rossi-Doria O., P. B., *A memetic algorithm for university course timetabling*, in: *Combinatorial Optimisation 2004 Book of Abstracts*, Lancaster, UK, Lancaster University, 2004.

[30] Van Hentenryck, P., "The OPL Optimization Programming Language," The MIT Press, Cambridge, MA, 1999.

[31] Walsh, T., *Depth-bounded Discrepancy Search*, in: *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI*, 1997, pp. 1388–1395.

[32] William D. Harvey, M. L. G., *Limited Discrepancy Search*, in: C. S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1* (1995), pp. 607–615.

[33] Winston, W., "Operations Research: Applications and Algorithms," International Thomson Publishing, Boston, Massachusetts, 1991.