GELATO: a multi-paradigm tool for Large Neighborhood Search

Raffaele Cipriano[1], Luca Di Gaspero[2], and Agostino Dovier[1]

[1] Dipartimento di Matematica e Informatica, Università degli Studi di Udine
via delle Scienze 208, I-33100, Udine, Italy
[2] Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica, Università degli Studi di Udine
via delle Scienze 208, I-33100, Udine, Italy

Abstract. We present GELATO, a general tool for encoding and solving optimization problems. Problems can be modeled using several paradigms and/or languages such as: Prolog, MiniZinc, and GECODE. Other paradigms can be included. Solution search is performed by a hybrid solver that exploits the potentiality of the Constraint Programming environment GECODE and of the Local Search framework EasyLocal++ for Large Neighborhood Search. The user can modify a set of parameters for guiding the hybrid search. In order to test the tool, we show the development phase of hybrid solvers on some benchmark problems. Moreover, we compare these solvers with other approaches, namely a pure Local Search, a pure constraint programming search, and with a state-of-the-art solver for constraint-based Local Search.

1 Introduction

The number of known approaches for dealing with constraint satisfaction problems (CSP) and constrained optimization problems (COP) is as huge as the difficulty of these problems. They range from mathematical approaches (i.e., methods from Operations Research, such as Integer Linear Programming, Column Generation, ...) to Artificial Intelligence approaches (such as Constraint Programming, Evolutionary Algorithms, SAT-based techniques, Local Search, just to name a few). It is also well-known (and formally proved) that methods that are adequate for a given set of problem instances are often useless for others [20]. Notwithstanding, there is agreement on the importance of developing tools for challenging these kind of problems in an easy and effective way. In this paper we go in that direction providing a multi-paradigm hybrid-search tool called GELATO (Gecode + Easy Local = A Tool for Optimization).

The tool comprises three main components, each of them dealing with one specific aspect of the problem solution phase. In the *modeling component* the user defines the problem at a high-level, and chooses the strategy for solving it. The *translation component* deals with the compilation of the model and the abstract algorithm defined by the user into the solver frameworks supported by the tool. Finally, the *solving component* runs the compiled program on the problem instance at hand, interleaving the solution strategies decided at the modeling stage.

GELATO currently supports a number of programming paradigms/languages for modeling. In particular, a declarative logic programming approach can be used for modeling, supporting Prolog and MiniZinc, and exploiting the front-end translator of Prolog presented in [4] and the current support of MiniZinc in GECODE. Moreover, an object-oriented approach can be used through the GECODE framework. Additional paradigm/languages can be easily added to the system, for example extending the front-end developed for the Haskell language [21] or other front-ends to GECODE.

The solving phase makes use of pure Constraint Programming tree search as well as a combination of Constraint Programming with Local Search. In particular, for this combination we have implemented a parametric schema for Large Neighborhood Search [14]. Large Neighborhood Search is a particular Local Search heuristic that relies on a constraint solver for blending the inference capabilities of Constraint Programming with the effectiveness of the Local Search techniques.

GELATO is based on two state-of-the-art, existing systems: the GECODE Constraint Programming environment [15] and the Local Search framework EasyLocal++ [6]. Both these systems are free and open C++ systems with a growing community of users. The main contribution of this paper is to blend together these two streams of works so as to generate a comprehensive tool, that allows to exploit the multi-paradigm/multi-language modeling and the multi-technique solving.

We provide some examples of the effectiveness of this approach by testing the solvers on three classes of benchmark problems (namely the Asymmetric Traveling Salesman Problem, the Minimum Energy Broadcast, and the Course Timetabling Problem) and comparing them against a pure Local Search approach, a pure Constraint Programming search, and an implementation of the same models in Comet [11], another language for hybrid systems.

We have defined a small set of few parameters that can be tuned by the user *against* a particular problem. In the paper we also provide some tuning of these parameters and show their default values.

A relevant aspect of our system is that the very same GECODEmodel (either directly written in GECODE or obtained by translation from Prolog or Minizinc) can be used either for pure Constraint Programming search or for LNS search (and by degenerating LNS with small neighborhoods, for Local Search). The resulting system is available from http://www.dimi.uniud.it/GELATO.
We compare the solvers obtained from GECODE with other approaches, namely a pure Local Search, a pure Constraint Programming search, and with Comet, a state-of-the-art solver for constraint-based Local Search. The results of the experimentation show that GECODE Large Neighborhood Search solver is able to outperform the Local Search and the Constraint Programming search on the set of benchmark problems and it achieves the same performances as Comet.

2 Preliminaries

A *Constraint Satisfaction Problem* (CSP) (see, e.g., [2]) $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$ is modeled by a set $\mathscr{X} = \{x_1, \ldots, x_k\}$ of *variables*, a set $\mathscr{D} = \{D_1, \ldots, D_k\}$ of *domains* associated to the variables (i.e., if $x_i = d_i$ then $d_i \in D_i$), and a set $\mathscr{C}$ of *constraints* (i.e., relations) over $\mathbf{dom} = D_1 \times \cdots \times D_k$.[3] A tuple $\langle d_1, \ldots, d_k \rangle \in \mathbf{dom}$ *satisfies* a constraint $C \in \mathscr{C}$ if and only if $\langle d_1, \ldots, d_k \rangle \in C$. $d = \langle d_1, \ldots, d_k \rangle \in \mathbf{dom}$ is a *solution* of a CSP $\mathscr{P}$ if d satisfies every constraint $C \in \mathscr{C}$. The set of the solutions of $\mathscr{P}$ is denoted by $\mathbf{sol}(\mathscr{P})$. $\mathscr{P}$ is said to be *consistent* if and only if If $\mathbf{sol}(\mathscr{P}) \neq \emptyset$.
A *Constrained Optimization Problem* (COP) $\mathscr{O} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, f \rangle$ is a CSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$ with an associated function $f : \mathbf{sol}(\mathscr{P}) \to E$ where $\langle E, \leq \rangle$ is a well-ordered set (typical instances of E are $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$). A *feasible solution* for $\mathscr{O}$ is any $d \in \mathbf{sol}(\mathscr{P})$. When clear from the context, we use $\mathbf{sol}(\mathscr{O})$ for $\mathbf{sol}(\mathscr{P})$. A tuple $e \in \mathbf{sol}(\mathscr{O})$ is a *solution* of the COP $\mathscr{O}$ if it minimizes the cost function f, namely if it holds that $\forall d \in \mathbf{sol}(\mathscr{O})\ f(e) \leq f(d)$.
Constraint Programming (CP) solves a CSP $\mathscr{P}$ by alternating the following two phases:

- a deterministic *constraint propagation* stage that reduces domains preserving $\mathbf{sol}(\mathscr{P})$, typically based on a *local* analysis of each constraint, one at a time;
- a non-deterministic *variable assignment*, in which one variable is selected together with one value in its current domain.

The process is repeated until a solution is found or a domain becomes empty. In the last case, the computation proceed by backtracking assignments, until possible. A COP $\mathscr{O}$ is solved by exploring the set $\mathbf{sol}(\mathscr{O})$ obtained in the previous way and storing the best value of the function f found during the search. However, a constraint analysis based on a partial assignment and on the best value already computed, might allow to sensibly prune the search tree. This complete search heuristics is called (with a slight of ambiguity with respect to Operations Research terminology) *branch and bound*.
A *Local Search* (LS) algorithm (e.g. [1]) for a COP $\mathscr{O}$ is given by defining a *search space* $\mathbf{sol}(\mathscr{O})$, a *neighborhood relation* $\mathscr{N}$, a *cost function* f, and a *stop criterion*. The neighborhood relation $\mathscr{N}$, i.e. a set $\mathscr{N}(d) \subseteq \mathbf{sol}(\mathscr{O})$, is defined for each element $d \in \mathbf{sol}(\mathscr{O})$. The set $\mathscr{N}(d)$ is called the *neighborhood* of d and each $d' \in \mathscr{N}(d)$ is called a *neighbor* of d. Commonly, $\mathscr{N}(d)$ is implicitly defined by referring to a set of possible *moves*, which are transitions between feasible solutions in the form of perturbations to be applied. Usually these perturbations insist on a small part of the problem, involving only few variables.
Starting from an initial solution $s_0 \in \mathbf{sol}(\mathscr{O})$, a LS *algorithm* iteratively navigates the space $\mathbf{sol}(\mathscr{O})$ by stepping from one solution s_i to a neighboring one $s_{i+1} \in \mathscr{N}(s_i)$, choosing s_{i+1} using a rule $\mathscr{M}$. The selection of the neighbor by $\mathscr{M}$ might depend on the whole computation that leaded s_0 to s_i and on the values of f on $\mathscr{N}(s_i)$. Moreover, it depends on the specific LS technique considered.
The stop criterion depends on the technique at hand, but it is typically based on stagnation detection (e.g., a maximum number of consecutive non improving moves, called *idle moves*) or a timeout. Several techniques for effectively exploring the search space have been presented in the literature (e.g., Montecarlo methods, Simulated Annealing, Hill Climbing, Steepest descent, Tabu Search, just to name a few).
LS algorithms can also be used to solve CSPs by relaxing some constraints and using f as a *distance to feasibility*, which accounts for the number of relaxed constraints that are violated.
Large Neighborhood Search (LNS) (e.g., [14]) is a LS method that relies on a particular definition of the neighborhood relation and on the strategy to explore the neighborhood. Differently from classical Local Search moves, which are small perturbations of the current solution, in LNS a large part of the solution is perturbed and searched for improvments. This part can be represented by a set $\mathbf{FV} \subseteq \mathscr{X}$ of variables, called *free variables*, that determines the neighborhood relation $\mathscr{N}$. More precisely, given a solution $s = \langle d_1, \ldots, d_k \rangle$ and a set $\mathbf{FV} \subseteq \{X_1, \ldots, X_k\}$ of free variables

$$\mathscr{N}(s, \mathbf{FV}) = \{\langle e_1, \ldots, e_k \rangle \in \mathbf{sol}(\mathscr{O}) : (X_i \notin \mathbf{FV}) \to (e_i = d_i)\}$$

Given **FV**, the neighborhood exploration can be performed with any searching technique, ranging from solution enumeration, to CP or Operations Research methods, and so on. In this work we focus on CP techniques for this

[3] Constraints do not necessarily relate all the variables. For instance, $x_1 + x_2 = 3$ is a binary constraint on the variables x_1 and x_2. This binary constraint, however, has a k-ary constraint counterpart $C = \{\langle d_1, d_2, \ldots, d_k \rangle : d_i \in D_i, d_1 + d_2 = 3\}$.

exploration. In this case, the search technique is known as Constraint Based Local Search (CBLS) [11]. The following aspects are crucial for the performance of this technique: *which variables* have to be selected (i.e., the definition of **FV**), and *how to perform the exploration* on these variables.
Once **FV** has been defined, the exploration of the neighborhood can be be made searching for:

- the *best neighbor*: namely, given a solution s and a set $\mathbf{FV} \subseteq \mathscr{X}$, look for a tuple $e \in \mathscr{N}(s,\mathbf{FV})$ such that $(\forall d \in \mathscr{N}(s,\mathbf{FV}))(f(e) \leq f(d))$.
- the best neighbor within a certain exploration timeout: namely the point in $\mathscr{N}(s,\mathbf{FV})$ that minimizes the value f found within a fixed timeout
- the first found neighbor point improving the value $f(s)$
- the first found neighbor point improving the objective function of at least a given value (e.g., below the 95% of $f(s)$).

Deciding *how many variables* will be free (i.e., $|\mathbf{FV}|$) affects the time spent on every large neighborhood exploration and the improvement of the objective function for each exploration. A small **FV** will lead to very efficient and fast search but, possibly at the price of very little improvement of the objective function. Otherwise, a big **FV** can lead to big improvements at each step, but every single exploration can take a lot of time. This trade-off should be investigated experimentally, looking at a size of **FV** which is a compromise between fast enough explorations and good improvements. Obviously, the choice of $|\mathbf{FV}|$ is strictly related to the search technique chosen (e.g., an efficient technique can manage more variables than a naïve one) and the fact that a timeout is used or not. The choice of *which variables* will be included in **FV** is strictly related to the problem we are solving: for simple and not too structured problems we can select the variables in a naïve way (randomly, or iterating between given sets of them); for complex and well-structured problems, we should define **FV** cleverly, selecting the variables which are most likely to give an improvement to the solution.
During the search of the new solution a portion of the neighborhood that does not fit the requirements (e.g., that do not improve the last known solution) is visited. Each of these neighbors is counted as a *failure*. The time spent on each exploration and the corresponding improvement of the cost function depends on $|\mathbf{FV}|$ (and, of course, on the structure of the problem) and a local stop criterion can be given either as a timeout or by setting a maximum number of failures (briefly, `maxF`) for the improving stage.
Experimental tests must be made for choosing values of $|\mathbf{FV}|$ and `maxF` that lead to quick explorations and good improvements. This choice might depend from other parameter choices. For instance, the choice of the order in which variables have to be instantiated in the CP stage can be naïve (e.g., choose the leftmost one) or clever (e.g., the one involved in more constraints and in case of equality break a tie choosing the one with smallest domain first). Usually, a clever technique allows to manage more variables, but this is not a general rule (it can be the case that the extra time for choosing a variable causes to exceed the timeout allowed).

2.1 A working example: the Simple Course Timetabling problem

Let us give the definition as a COP of a basic timetabling problem that we will use as a working example in the rest of the paper. The problem is referred as the SCTT problem in the rest of the paper.
Given a set of courses $S = \{c_1,\ldots,c_n\}$, each course c_i requires a number of weekly lectures $l(c_i)$ where $l : S \to \mathbb{N}$, and is taught by a teacher $t(c_i)$ where $t : S \to T = \{t_1,\ldots,t_g\}$. Five teaching days (from Monday to Friday) and two time slots for each day (Morning and Afternoon) are allowed, thus having a set $P = \{p_1,\ldots,p_{10}\}$ of ten possible time periods. Each teacher t_j can be unavailable for some periods $u(t_j)$ where $u : T \to 2^P$.
The problem consists in finding a schedule for all the courses, so that:
(C_1) All the required lectures for each course are given.
(C_2) Lectures of courses taught by the same teacher are scheduled in distinct periods.
(C_3) Teacher un-availabilities are taken into account.
Moreover, a cost function is defined on the basis of the following criterion: the lectures of each course c_i should be at least spread into a given minimum number of days $\delta(c_i)$, where $\delta : S \to \{1,\ldots,5\}$.
Let us consider the following toy instance, just to fix the ideas:

- $S = \{OS, PL, AI\}$ (Operating Systems, Programming Languages, Artificial Intelligence)
- $l(OS) = 3$, $l(PL) = 4$, $l(AI) = 3$
- $t(OS) = Schroeder$, $t(PL) = Schroeder$, $t(AI) = Linus$
- $P = \{MO_m, MO_a, TU_m, TU_a, WE_m, WE_a, TH_m, TH_a, FR_m, FR_a\}$
- $\delta(OS) = 3$, $\delta(PL) = 4$, $\delta(AI) = 3$
- $u(Schroeder) = \{TU_m, TU_a\}$, $u(Linus) = \{TH_m, TH_a, FR_m, FR_a\}$

A possible model for this problem can be defined as follows:

- $\mathscr{X}$ consists of $|S| \cdot |P|$ boolean variables $x_{c,p}$. The variable $x_{c,p} = 1$ if and only if course c is scheduled at period p (otherwise it is 0).
- The required number of lectures is assigned to each course:

$$\sum_{p \in P} x_{c,p} = l(c) \quad \forall c \in S \qquad (C_1)$$

– The constraints stating that lectures of courses taught by the same teacher are scheduled in distinct periods can be modeled as:[4]

$$x_{c,p} \cdot x_{c',p} = 0 \quad \forall c, c' \in S \text{ s.t. } (c \neq c' \wedge t(c) = t(c')), \forall p \in P \tag{C_2}$$

– The "un-available" constraints are modeled as follows:

$$x_{c,p} = 0 \quad \forall c \in S \forall p \in u(t(c)) \tag{C_3}$$

– The objective function f is defined by summing, for each course c, the difference between $\delta(c)$ and the actual number of days occupied by the lectures of c in a feasible solution:

$$f(\mathscr{X}) = \sum_{c \in S} \max(0, \delta(c) - |\{d(p) : p \in P \wedge x_{c,p} > 0\}|) \tag{1}$$

where $d : P \rightarrow \{MO, TU, WE, TH, FR\}$ is a function assigning a period to a day.

In Figure 1 we report an assignment that is not a solution (on Monday morning Prof. *Schroeder* has two different lectures in the same time slot and (moreover) Prof. *Linus* gives a lecture on Thursday, when he is unavailable), a solution which is not optimal, and an optimal solution. With γ, in Figure 1, we denote $\delta(c) - |\{d(p) : p \in P \wedge x_{c,p} > 0\}|$, where c is OS, PL, or AI.

$P \Rightarrow$	MO		TU		WE		TH		FR		f
$S \Downarrow$	m	a	m	a	m	a	m	a	m	a	γ
OS	1	0	0	0	1	0	1	0	0	0	3-3
PL	1	0	0	0	0	1	0	1	0	1	4-4
AI	0	0	0	0	1	1	1	0	0	0	3-2
(1)							Total f:				1

	MO		TU		WE		TH		FR		f
	m	a	m	a	m	a	m	a	m	a	γ
	1	0	0	0	1	0	1	0	0	0	3-3
	0	0	0	0	0	1	0	1	1	1	4-3
	1	1	1	0	0	0	0	0	0	0	3-2
(2)							Total f:				2

	MO		TU		WE		TH		FR		f
	m	a	m	a	m	a	m	a	m	a	γ
	1	0	0	0	1	0	1	0	0	0	3-3
	0	1	0	0	0	1	0	1	0	1	4-4
	1	0	1	0	1	0	0	0	0	0	3-3
(3)							Total f:				0

Fig. 1. An assignment which is not a solution (1), a solution (2), and an optimum solution (3) for the given toy instance of SCTT

3 The existing CP and LS systems used

The core of $\mathbb{GELATO}$ is based on two programming environments for CSPs/COPs, namely GECODE and EasyLocal++. We briefly introduce them in this section (for more details, see [15] and [6], respectively).

3.1 Gecode Overview

GECODE (GEneric COnstraint Development Environment) [15] is a programming environment for developing constraint-based systems and applications developed by a group led by Christian Schulte, and including Mikael Lagerkvist, and Guido Tack. It is a modular and extensible C++ constraint programming toolkit, whose development started in 2002 and that counts 28 improving versions.[5]

GECODE is *open source* and is distributed under the MIT license. All of its parts (source code, example, documentation...) are available for download from http://www.gecode.org and anyone can modify the implementation of its classes, extend or improve their functionalities, or interface GECODE to other systems.

Being completely implemented in C++ and adhering to the language standards, GECODE is *portable*, and it can be compiled and run on most current platforms. Moreover, GECODE is *documented* in a tutorial [17] that explains its architecture and some programming tricks, and in an on-line reference documentation [16] that gives the technical specifications of each module/class/function implemented in the framework.

GECODE has excellent *performance* with respect to both runtime and memory usage, it won severals competitions, such as the MiniZinc Challenge, in 2008–2011 and it can be considered a state-of-the-art constraint solver.

GECODE also implements *parallel* search by exploiting the multiple cores of today's hardware, giving to an already efficient base system an additional edge.

The discussion of GECODE architecture is out of the scope of this paper, we refer the to [16] for the specific details.

[4] A linear constraint such as: $(x_{c,p} + x_{c',p} \leq 1)$ can be used here. We have employed a non-linear constraint to point out the flexibility of Constraint Programming.

[5] At the time of writing, September 2011, the current version is 3.7.0.

3.2 Gecode Modeling

GECODE comes with extensive modeling support, that allows the user to encode his/her problem using higher-level language facilities rather than rough low-level statements. The modeling support includes: regular expressions for extensional constraints, arithmetic expressions, matrices, sets, and boolean constraints.
A GECODE model is implemented using *spaces*: a *space* is the repository for variables, constraints, objective function, and searching options. Modeling exploits the C++ notion of *inheritance*: a model must implement the class *Space*, and the subclass constructor implements the actual model. In addition to the constructor, a model must implement some other functions (e.g., those performing a copy of the space, or returning the objective function, ...). A GECODE space can be asked to perform the propagation of the constraints, to find the first solution (or the next one), exploring the search tree, or to find the best solution in the whole search space.
We survey the description of GECODE modeling using the SCTT example defined in Section 2.1. We recall that a problem encoded in GECODEcan be solved by GELATO as well.
Every GECODE model is defined into a class that inherits from a GECODE `Space` superclass. The superclasses available are `Space` for CSPs and `MinimizeSpace` or `MaximizeSpace` for COPs. In our working example, reported in Listing 1.1 we declare the class `Timetabling` to be a subclass of `MinimizeSpace` (line 1).

Listing 1.1. SCTT encoding: model heading definition

```
class Timetabling : MinimizeSpace {
  IntVarArray x;
  IntVar      fobj;
  Timetabling(const Faculty& in ):
    x(*this, in.Courses() * in.Periods(), 0, 1),
    fobj(*this, 0, Int::Limits::max)
  { (/ \ldots /)
```

On line 2 the array `x` for the variables $\mathscr{X}$ of the problem is declared, and on line 3 we declare the variable `fobj` that will store the value of the objective function. `IntVarArray` and `IntVar` are built-in GECODE data structures. Line 4 starts the constructor method. It takes as input an object of the class `Faculty`, that contains all the information specific to the instance at hand. All these input information are stored in the variable `in`. Line 5 sets variables and domains: the array `x` is constructed so to contain a number `in.Periods()*in.Courses()` of boolean variables. At line 6, the interval between 0 and the largest allowed integer value is set as the domain of the variable `fobj`.
Moving to modeling constraints, GECODE provides a `Matrix` support class for accessing an array as a two dimensional matrix (matrices are, of course, very common in modeling). Going back to our working example, in Listing 1.2, line 10 sets up a matrix interface to access the vector `x`, with the number of columns equal to the number of time periods and the number of rows equal to the number of courses.

Listing 1.2. SCTT modeling: main constraints

```
unsigned int cols =  in.Periods() ;
unsigned int rows = in.Courses();

Matrix<IntVarArgs> mat(x, cols, rows);

 for (int r = 0; r < rows; r++)
   linear(*this,  mat.row(r), IRT_EQ, in.NumberOfLectures(r));

 for (int p = 0; p < cols; p++)
   for (int r1 = 0; r1 < rows - 1; r1++ )
     for (int r2 = r1+1; r2 < rows; r2++ )
       if (in.SameTeacher(r1,r2))
         rel(*this, mat(c,r1) * mat(c,r2) == 0);

   for (int p = 0; period < in.Periods(); p++)
     for (int c = 0; c < in.Courses() ; c++ )
       if (!in.Available(c,p))
         rel(*this, mat(p,c) == 0);
```

The matrix has the same structure as those reported in Figure 1. Lines 12–13 post the constraint (C_1) (see, Section 2.1) on each row of the matrix. The constraint `linear` is a built-in arithmetical operator of GECODE, and line 12 corresponds to the formula

$$\sum_{\texttt{col}} \texttt{mat.row}(r)[\texttt{col}] = \texttt{in.NumberOfLectures}(r)$$

where `in.NumberOfLectures(r)` contains the value of function l (number of weekly lectures) for each course (i.e., for each row). `IRT_EQ` is the built-in predicate for equality between finite domain constraints.[6]
Lines 15–19 post the constraint (C_2). Line 18 corresponds to the constraint $x(c,r1)\cdot x(c,r2)=0$, applied to the variables of the same time periods sharing the same teacher.
Lines 21–24 post the constraint (C_3). In particular, at line 24, whenever $x(p,c)$ is a time slot unavailable for the given professor, the constraint $x(p,c)=0$ is posted.

Objective Function and Branching. The formula computing the objective function is reported in Listing 1.3.

Listing 1.3. SCTT modeling: cost function

```
IntVarArgs vectorWD(*this, in.Courses(), 0, Int::Limits::max);
IntVarArgs sumWD(*this, in.Courses(), 0, Int::Limits::max);
IntVarArgs diff(*this, in.Courses(), Int::Limits::min, Int::Limits::max);

for (int course = 0 ; course < in.Courses(); course++)
{
  IntVarArgs workingDays(in.Days(), 0, 1);

  for (int day = 0 ; day < in.Days(); day++)
  {
    IntVarArgs Day( in.PeriodsPerDay(), 0,1 );
    for (unsigned int slot = 0; slot < in.PeriodsPerDay(); slot++)
    {
      int period = day*in.PeriodsPerDay() + slot;
      expr(*this, (mat(period, course) != 0) == (Day[slot] == 1));
    }
    IntVar nLecturesPerDay(*this, 0, in.PeriodsPerDay());
    linear(*this, Day, IRT_EQ, nLecturesPerDay);
    expr(*this, (nLecturesPerDay != 0) == (workingDays[day] == 1));
  }
  linear(*this, workingDays, IRT_EQ, sumWD[course]);
  diff[course] = post(*this,in.CourseVector(course).MinWD()-sumWD[course]);
  Gecode::max(*this, ZERO, diff[course], vectorWD[course]);
}

linear(*this, vectorWD, IRT_EQ, fobj);

branch(*this,x,tiebreak(INT_VAR_DEGREE_MAX,INT_VAR_SIZE_MAX),INT_VAL_MED);
```

As for the objective function, some auxiliary arrays of temporary variables are introduced. They are declared as `IntVarArgs`, a GECODE built-in data type for array of temporary variables. At the end of this piece of code, the formula is bound to the `fobj` variable.
During the search, a variable is selected and a value of its domain is attempted. This operation is called *branching*, and the branching strategy is defined at line 51. The variables to branch are those in the array `x` and the variable selection strategy is `tiebreak(INT_VAR_DEGREE_MAX,INT_VAR_SIZE_MAX)`, i.e., the variable with the highest number of constraint on it is selected, breaking ties choosing the variable with largest domain size. The values selection strategy is `INT_VAL_MED`, i.e., the greatest value not greater than the median is selected. Other choices are, of course, possible.
In order to use the model defined above in a constraint based branch and bound search engine (see Section 2) the following functions need to be defined into the `Timetabling` class (see Listing 1.4):

- The function `cost`, that returns the variable representing the cost of a solution, i.e., the objective function of the model. This function is defined at lines 54–55 and simply returns the variable `fobj`.
- A branch and bound search engine needs to know what constraint to add every time a new solution is found, in order to drive the search to better solutions and cut the search space. The function `constrain` (lines 57–61) defines the desired constraint to add: it takes in input a `Space` object, (i.e., a solution of the model), and posts a constraint (line 60) stating that the value of the variable `fobj` has to be less than the `fobj` value of the solution `sol`. The GECODE branch and bound search engine calls this method, every time it finds a new solution, passing the solution found as parameter.

[6] A similar syntax is used for $\neq, <, \leq$, etc.

Listing 1.4. SCTT modeling: branch and bound

```
IntVar Timetabling::cost(void) const
{ return fobj; }

void Timetabling::constrain(const Space& sol)
{
  const Timetabling& s = static_cast<const Timetabling&>(sol);
  rel(*this, fobj, IRT_LT, s.fobj.val());
}
```

3.3 EasyLocal++

EasyLocal++ [6] is an object-oriented framework that allows programmers to design, implement and test LS algorithms in an easy, fast and flexible way. The underlying idea is to capture the essential features of most LS meta-heuristics, and their possible compositions, allowing the user to address the design and implementation issues of new LS heuristics in a more principled way. EasyLocal++ has been entirely developed in C++ with wide use of object-oriented patterns and currently it is at its 2.0. release.
Modeling a problem using EasyLocal++ requires that the C++ classes representing the problem specific layers of the EasyLocal++ hierarchy to be defined. On the other hand, the framework provides the full control structures for the invariant part of the LS algorithms. Consequently, the user is required only to supply the problem specific details by defining concrete classes and implementing concrete methods.
Going into the details of the EasyLocal++ development process is out of the scope of this paper, therefore we refer the interested reader to [5].

4 GELATO: Gecode + Easy Local = A Tool for Optimization

The aim of GELATO is to allow the programmer to easily model a CSP/COP using one of the three modeling languages: Prolog, MiniZinc, and GECODE, define or select the meta-heuristics using a tiny meta-heuristics modeling language that will allow to program search heuristics in a wide range.

4.1 High level modeling and translation

The system is currently able to handle CSPs and COPs modeled in Prolog exploiting the front-end translators of Prolog to GECODE presented in [4], and in MiniZinc, exploiting the (two steps) translator of MiniZinc in GECODE available in the GECODE distribution. Moreover, the modeling capabilities can benefit from the front-end developed for the Haskell language [21] and, in general, from other front-ends to GECODE, usually listed in the GECODE web-site [15].
We do not enter here in the translation details, but we just say that the translations are based on the low-level modeling language FlatZinc. FlatZinc models are a list of simple constraints, without other programming constructs. Basically, a model is the unfolded version of a MiniZinc model, which can be interpreted directly by GECODE.
In what follows, instead, we will show how to encode the SCTT problem (Section 2.1) in Prolog and in MiniZinc.

Prolog and Constraint Logic Programming. According to [8], Logic Programming was the first community embedding constraint programming giving raise to the so-called Constraint Logic Programming (CLP) paradigm. Nowadays all available Prolog system comes equipped with a constraints solver on finite domains (and on other domains, such as booleans, sets, rational numbers, ...). Due to the slight differences in the syntax of some primitives, we are focusing on one system, namely SICStus Prolog [13].
We are going to show the encoding of the SCTT problem in SICStus Prolog. First, let us focus on the input format. `L` is a list storing the function ℓ where courses are assumed ordered as `OS`, `PL`, `AI`. `T` is a list storing the function t, and `Unav` is a list of lists of unavailable time periods, given as numbers (precisely, Monday morning is 1, ..., Friday afternoon is 10), where the order of the teachers must be the same in the two lists (Schroeder, then Linus). `D` is a list storing the minimum desired duration of the courses. Our toy instance would be therefore represented by:

```
L    = [3,4,3]                  T = [[os,pl],[ai]]
Unav = [[3,4],[7,8,9,10]]       D = [3,4,3]
```

In Listing 1.5 we report the complete encoding of the problem in Prolog. If, on the one side, one might appreciate the compactness of the code, on the other side recursion is employed to implement the various "for loops" needed. Actually, most Prolog interpreters now implements a `foreach` iterator. However, this is not (yet) the standard way of encoding in Prolog. If the reader wants to run it, the libraries `lists` and `clpfd` must be included.

The sketch of the code is as follows: at line 2 the matrix is generated row by row and at line 3 the domains are assigned to its variables. Then the three predicates adding the corresponding constraints are called (line 4) and the objective function is defined, using constraints, by predicate `build_fobj` at line 7. Search is called at line 6 by the built-in predicate `labeling` with the option of minimizing the function `FOBJ`.

The predicates `c1`, `c2`, and `c3` called at line 4, implement the constraints (C_1), (C_2), and (C_3). They are defined at lines 10–13, 15–29, and 31–43, respectively. The function `fobj` is defined by the code at lines 45–58.

Even though the code is mostly self-contained, we just focus on some points. At line 11 the built-in `sum` predicate is used; this is similar to the use of `linear` in Listing 1.2. At lines 40–41 the variables corresponding to an unavailable period are set to 0, while the "iff" definition of line 56 states that the flag variable TD is set to 1 if and only if a class is held in that day (either in the morning or in the afternoon).

Listing 1.5. SICStus Prolog encoding of the SCTT problem

```
sctt(L, T, Unav, D, Mat):-
  length(OS, 10), length(PL, 10), length(AI, 10), Mat=[OS,PL,AI],
  append([OS,PL,AI],Vars), domain(Vars, 0, 1),
  c1(Mat, L),
  c2(Mat, T),
  c3(Mat, Unav, T),
  build_fobj(Mat,D,FOBJ),
  labeling([minimize(FOBJ)], Vars).

c1([Row|Tr],[L|Tl]):-
  sum( Row, #=, L ),
  c1(Tr,Tl).
c1([],[]).

c2(MAT,[]).
c2(MAT,[[_]|T]) :- c2(MAT, T).
c2([OS,PL,AI],[[A,B|R]|T]) :-
  c2_aux([OS,PL,AI], [A,B]),
  c2([OS,PL,AI], [[A|R]|T]),
  c2([OS,PL,AI], [[B|R]|T]).
c2_aux([OS,PL,AI],[A,B]) :-
  (A=os,B=pl) -> c2_post(OS,PL);
  (A=os,B=ai) -> c2_post(OS,AI);
  (A=pl,B=ai) -> c2_post(PL,AI).

c2_post([X1|T1],[X2|T2]):-
  X1 * X2 #= 0,
  c2_post(T1, T2).
c2_post([], []).

c3(MAT, [], []).
c3(MAT, [_|UnAvailable], [[]|T]) :-
  c3(MAT, UnAvailable, T).
c3([OS,PL,AI], [Un|Available], [[A|R]|T]) :-
  (A = os -> c3_aux(OS,Un);
  A = pl -> c3_aux(PL,Un);
  A = ai -> c3_aux(AI,Un)),
  c3([OS,PL,AI], [Un|Available], [R|T]).
c3_aux(Row, [Un|Tu]):-
  nth1(Un, Row, X),
  X#=0,
  c3_aux(Row, Tu).
c3_aux(_, []).

build_fobj([Row|M], [D|Ds], FOBJ):-
```

```
  build_fobj( M, Ds, FObjT),
  single_contribution(Row, D, FObjR),
  FObjT + FObjR #= FOBJ.
build_fobj([], _, 0).
single_contribution(Row, D, FObjR):-
  number_of_days(Row, Days),
  Days+Diff  #= D,
  max(Diff, 0 ) #= FObjR.
number_of_days([M,P|Row], TotalDays):-
  number_of_days(Row, Days),
  TD #<=> (M+P #> 0),
  TD + Days #= TotalDays.
number_of_days([], 0).
```

MiniZinc. MiniZinc is a medium-level declarative constraint modeling language developed by NICTA (National ICT Australia) designed for specifying constrained optimization and decision problems over integers and real numbers [12]. MiniZinc is designed so as to be easily interfaced to backend solvers, via compilation in a low level language called FlatZinc. We refer to the MiniZinc manual [9] for details. We just report here the encoding of the SCTT problem (Listing 1.6).

Listing 1.6. The SCTT problem encoded in MiniZinc.

```
int: courses  = 3;
int: teachers = 2;
int: periods = 10;
int: weekdays = 5;
array[1..courses]  of 1..periods: l = [3,4,3];
array[1..courses]  of 1..teachers: t = [1,1,2];
array[1..teachers] of set of 1..periods: u = [{3,4},{7,8,9,10}];
array[1..courses]  of int: delta = [3,4,3];

array[1..courses,1..periods] of var 0..1: x;
var int: fobj;

constraint forall(c in 1..courses)
  (sum(p in 1..periods)(x[c,p]) = l[c]);
constraint forall(ci, cj in 1..courses where ci<cj /\ t[ci] = t[cj])
  (forall (p in 1..periods) (x[ci,p] * x[cj,p] = 0));
constraint forall(c in 1..courses)
  (forall (p in u[t[c]]) (x[c,p] = 0));

constraint fobj = sum(c in 1..courses)
  (max(0,delta[c] - sum(d in 1..weekdays) (max(x[c,2*d],x[c,2*d-1]))));

solve minimize fobj;

output ["x = ",show(x),"\n", "Fobj = ", show(fobj), "\n"];
```

Let us focus on the lines 1–8, which store the input data. Constants for the number of courses, teachers, periods, and days are initialized and used in arrays (in particular at line 7 an array of sets is initialized). Line 10 defines the boolean matrix and line 11 defines the objective function `fobj`. Constraints (C_1), (C_2), and (C_3) are posted in a very natural way at lines 13–18, while the objective function is initialized at lines 20–21. The predicate `max` (line 21, second occurrence) between the two periods per day returns 1 if that day is used, 0 otherwise. The solver is colled at line 23 with the `minimize` option (whereas `solve satisfy` is used for a CSP) while line 25 states how to output the result.

4.2 The core of the GELATO hybrid solver

The core of the tool is constituted by a hybrid solver whose preliminary implementation has been presented in [3].
GELATO functionalities are divided into two main parts: an *internal core*, that merges GECODE and Easy-Local++ together, and an *external interface*, that provides an easy way for the user to interact with the system. Figure 2 shows the general architecture of the tool.

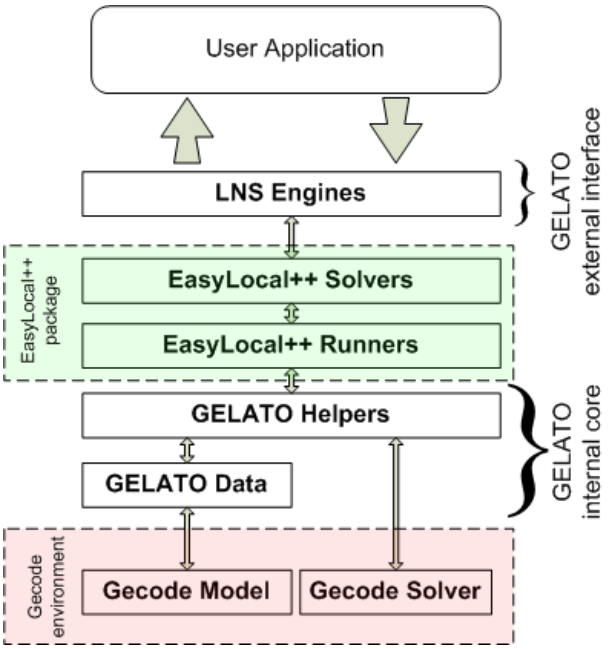

Fig. 2. Structure of GELATO and interactions with GECODE and EasyLocal++

The *internal core* is a *middle layer* that defines the interaction between EasyLocal++ and GECODE. In particular, these classes include the encodings of GECODE models and CP search engines.
The *external interface* provides high level functions that can easily be called by the user to access the inner functionalities, thus hiding all the internal GECODE and EasyLocal++ calls. This external layer act as a *Façade* object-oriented pattern (see [7]): the aim of the Façade pattern is to provide a simplified interface to a larger body of code, making the subsystem easier to use. Thus, this external layer transforms the overall GELATO tool into a black-box and allows the user to interact with it just by passing the input (i.e., the problem model and the solving meta-heuristic) and retrieving the output (i.e., the solution obtained), without caring about complicated object interactions in the GELATO internal core.
In this scenario, the user is asked to:

1. *model the CSP or COP* with a modeling language supported by GELATO
2. *specify the instance* of the problem in a different file, since GELATO requires the concepts of *problem* and *instance* to be separated;
3. *select the meta-heuristic* to be used to find the solution (e.g., Hill Climbing, Tabu Search, Large Neighborhood Search) and specify the desired parameters for its execution.

It is important to note that GELATO does not require any modification of GECODE and EasyLocal++. In this way GELATO does not affect the single development of GECODE or EasyLocal++ and every improvement of the two basic tools, such as new functionalities or performance improvements, is immediately inherited by GELATO.

5 Benchmarks

In this section we briefly describe the three benchmark problems chosen to test our tool. They are well-known COPs with several available sets of input instances.

5.1 Asymmetric Traveling Salesman Problem

This problem is taken from the TSPLib [18] and it is defined as follows.

Definition 1 (Asymmetric Traveling Salesman Problem (ATSP)). *Given a directed graph $G = (V,E)$ and a function c that assigns a cost to each directed edge (i, j), find a roundtrip of minimal total cost visiting each node exactly once.*

Let us observe that the edge costs might be asymmetric, in the sense that $c(i, j)$ and $c(j,i)$ can be different. The problem is therefore a generalization of the well-known Traveling Salesman Problem problem.

The problem is modeled as follows: let $\mathscr{X} = \{x_1, \ldots, x_n\}, n = |V|$ be the set of variables with domains $D_1 = D_2 = \ldots = D_n = \{1, \ldots, n\}$. The value of x_i represents the *successor* of the vertex i in the tour. We exploit the global constraint $\texttt{circuit}([x_1, \ldots, x_n])$ available in most CP frameworks, to constrain the solutions that represent a tour. The cost function is defined as $f = \sum_{i=1}^{n} c(i, x_i)$.

5.2 Minimum Energy Broadcast

This problem is drawn from CSPLIB (available from http://www.csblib.org, problem number 48) and it is an optimization problem for the design of Wireless Networks. Further information can be found in [19].

Definition 2 (MEB). *Given a set of n nodes $V = \{1, \ldots, n\}$ forming a complete graph K_n, a source node $s \in V$, and and a cost function $p : V \times V \to \mathbb{R}$, representing the transmission cost between two nodes, the problem consists in finding a (directed) spanning tree rooted at s that minimizes a cost function f that measures the energy needed by a node for broadcasting information, and defined as follows. Assume a tree τ is given, and let us denote by $\texttt{children}(i)$ the set of children nodes of the node i in the tree τ (namely, $\texttt{children}(i) = \{j \in V \mid (i, j) \in \tau\}$). Then $f(\tau) = \sum_{i=1}^{n} \max\{p(i, j) : j \in \texttt{children}(i)\}$.*

We model the problem using a set $\mathscr{X} = \{x_{i,j} \in \{0, 1\} \mid i, j \in V\}$ of n^2 boolean variables, with the meaning that $x_{i,j} = 1$ if and only if the edge (i, j) is in the solution τ. Let us define the function $\delta : V \to V$ as follows: $\delta(s) = s$ (s is the source node) and $\delta(j) = i$ if $j \neq s$ and $x_{i,j} = 1$ (the parent node in the tree—the function is well-defined as long as we know that for all $j \neq s$ exists exactly one i such that $x_{i,j} = 1$). Then, for $k \in \mathbb{N}$, $\delta^k(i)$ is recursively defined as follows: $\delta^0(j) = j$ and $\delta^{k+1}(j) = \delta(\delta^k(j))$.
The constraints added to the problem to ensure the tree structure of the output are the following:

$$\sum_{i=1}^{n} x_{i,j} = 1 \qquad j = 1, \ldots, n, j \neq s \qquad (C_{1.1})$$

$$\sum_{i=1}^{n} x_{i,s} = 0 \qquad (C_{1.2})$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} = n - 1 \qquad (C_2)$$

$$\delta^n(j) = s \qquad j = 1, \ldots, n \qquad (C_3)$$

and the objective function is: $f = \sum_{i=1}^{n} \max_{j=1}^{n}(x_{i,j} p(i, j))$.
As a minor implementation remark, we have multiplied the input data by 100 so as to use (finite) integer values for p instead of the real values stored in the problem instances used.

5.3 Course Timetabling

This problem has been introduced as Track 3 of the second International Timetabling Competition held in 2007 [10]. It consists in the weekly scheduling of the lectures of a set of university courses on the basis of a set of predefined curricula published by the University.

Definition 3 (CTT). *Given a set of courses $C = \{c_1, \ldots, c_n\}$, each course c_i consists of a set of lectures $L_i = \{l_{i_1}, \ldots, l_{i_a}\} \in \mathscr{L}$, is taught by a teacher $t : C \to T = \{t_1, \ldots, t_g\}$, it is attended by a number of students $s : C \to \mathbb{N}$, and belongs to one or more curricula $Q = \{q_1, \ldots, q_b\}$, where $q_i \subseteq C, i = 1, \ldots, b$, which are subset of courses that have students in common. Moreover it is given a set of periods $P = \{1, \ldots, p\}$, each period belonging to a single teaching day $d : P \to \{1, \ldots, h\}$, and a set of rooms $R = \{r_1, \ldots, r_m\}$, each with a capacity $w : R \to \mathbb{N}$. Each teacher t_i can be unavailable for some periods $u : T \to 2^P$.*
The problem consists in finding an assignment $\tau : \mathscr{L} \to R \times P$ of a room and period to each lecture of a course so that:
(H_1) all lectures of courses are assigned;
(H_2) for each (room, period) pair, only one lecture is assigned;
($H_{3.1}$) lectures of courses in the same curriculum are scheduled in distinct periods;
($H_{3.2}$) lectures of courses taught by the same teacher are scheduled in distinct periods;
(H_4) teacher unavailabilities are taken into account.
Moreover, a cost is defined for the following criteria (soft constraints):
(S_1) each lecture should be scheduled in a room large enough for containing all its students;
(S_2) the lectures of each course should be spread into a given minimum number of days $\delta : C \to \{1, \ldots, h\}$;
(S_3) lectures belonging to a curriculum should be adjacent to each other;
(S_4) all lectures of a course would be preferably assigned to the same room.

We model the CTT problem by defining the set $\mathscr{X}$ of $C \cdot P$ variables $x_{c,p} \in \{0,\ldots,r\}$, with the intuitive meaning that $x_{c,p} = r > 0$ if and only if course c is scheduled at period p in room r, and $x_{c,p} = 0$ if course c is not scheduled at period p. The constraints are modeled as follows:

$$\sum_{j \in P} (x_{c_i,j} = 0) = p - |L_i| \qquad i = 1,\ldots,n \qquad (H_1)$$

$$\sum_{i=1}^{n} (x_{c_i,j} = 0) \geq n - m \qquad j = 1,\ldots,p \qquad (H_2)$$

$$x_{c,j} \cdot x_{c',j} = 0 \qquad c,c' \in C \text{ s.t. } t(c) = t(c'), j = 1,\ldots,p \qquad (H_{3.1})$$

$$\sum_{c \in q_i} x_{c,j} \leq 1 \qquad i = 1,\ldots,b, j = 1,\ldots,p \qquad (H_{3.2})$$

$$x_{c,j} = 0 \qquad j \in u(t(c)), c \in C \qquad (H_4)$$

The objective function f is the sum of the following four components:

$$s_1 = \sum_{i=1}^{n} \sum_{j=1}^{p} \max(0, s(c_i) - w(x_{c_i,j})) \qquad (S_1)$$

$$s_2 = 5 \cdot \sum_{i=1}^{n} \max(0, \delta(c_i) - |\{d(j) : x_{c_i,j} > 0\}|) \qquad (S_2)$$

$$s_3 = 2 \cdot \sum_{i=1}^{b} \Big|\{k \in P : \mathsf{start}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k+1} = 0\}\Big| + \qquad (S_3)$$
$$\Big|\{k \in P : \mathsf{end}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k-1} = 0\}\Big| +$$
$$\Big|\{k \in P : \mathsf{mid}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k-1} = 0 \wedge \sum_{c \in q_i} x_{c,k+1} = 0\}\Big|$$

$$s_4 = \sum_{i=1}^{n} (|\{r : \exists p(x_{c_i,p} = r)\}| - 1) \qquad (S_4)$$

where start, end, and mid are three boolean functions that state if a period is initial, ending, or in the middle of a day, respectively.

6 Experiments and comparison

We briefly explain how experiments drove us in tuning search parameters and show that the performances of GELATO are comparable to those of an LNS solver implemented in Comet. All computations were run on an AMD Opteron 2.2 GHz Linux Machine. We used GECODE 3.1.0, EasyLocal++ 2.0, and Comet 2.0.1. Additional tests have shown that GELATO is fully compatible with GECODE 3.7.0, the latest release at the time of writing.

6.1 Solving techniques

The problem instances have been solved using:

1. a pure constraint programming approach in GECODE
2. a pure Local Search approach in EasyLocal++, and
3. LNS meta-heuristics encoded in GELATO.

We use the same model, the same meta-heuristics, and the same parameters used for (3) in the Comet system to guarantee a fair comparison with that system.

We tested the ATSP on the following instances of growing size, taken from the TSPLib [18]: br17 (that we call *instance 0*, with $|V| = 17$), ftv33 (*inst. 1*, $|V| = 34$), ftv55 (*inst. 2*, $|V| = 56$), ftv70 (*inst. 3*, $|V| = 71$), kro124p (*inst. 4*, $|V| = 100$), and ftv170 (*inst. 5*, $|V| = 171$). For the MEB problem we selected the following six instances of size $|V| = 20$ (hence, $|\mathscr{X}| = 400$) from the set used in [19]: p20.02/03/08/14/24/29. For the CTT problem we selected from the website http://tabu.diegm.uniud.it/ctt/ six instances with different features (instance size, average number of conflicts, average teacher availability, average room occupation, ...): comp01/04/07/09/11/14.

Pure CP Approach: GECODE. The pure CP approach in GECODE exploits the straightforward encoding of the models described in Sect. 5. For the sake of showing the possibility of using $\mathbb{GELATO}$ starting from already available CP models in the case of ATSP we used the model reported in the set of GECODE examples, slightly adapted for its integration. For MEB and CTT, instead, we encoded the GECODE models from scratch. We tried different search strategies for each problem, and choose the ones with best performances: for ATSP the variable with smallest domain is selected, and the values are chosen in increasing order; for MEB the variable with smallest domain is selected and the values are chosen randomly; for CTT the most constrained variable is selected, breaking ties randomly, and values are chosen randomly.

Pure LS Approach: EasyLocal++. The pure LS approaches in EL are based on elementary move definitions. For ATSP we use a *swap-move*: given a tour, two nodes are randomly selected and swapped. For MEB we define a *change-parent-move*: given a directed rooted tree, two nodes A and B are selected, so that B is not an ancestor of A; then A becomes the parent of B. For CTT we use a *time-and-room* exchange move: given a timetable, we select randomly a lesson scheduled at period p in room r and move it to another period p_1 into another room r_1, chosen from the empty ones.
In order to compare the algorithms based on pure LS with LNS on a fair base, we decided to drive the Local Search by means of a Randomized Hill Climbing scheme. Indeed, LNS actually perform a "large" hill climbing in a wider neighborhood, we call *mountain* climbing. Developing LS strategies with more complex move definitions and more elaborated meta-heuristics is out of the scope of the present work.
It is worth noticing that for LNS implementation we can directly reuse the existing GECODE models, whereas using EasyLocal++ we had to implement from scratch the basic LS classes for the problem.
As a final note, all LS algorithms are stopped after 1000 iterations without improvement.

Large Neighborhood Search Approach: $\mathbb{GELATO}$ *(and Comet).* The LNS meta-heuristic is composed by a deterministic CP search for the first solution and then a LNS exploration based on a mountain climbing algorithm with a maximum number of idle iteration.
Given a COP $\mathscr{O} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C}, f \rangle$, the first solution is obtained by a CP search using GECODE without any pre-assigned value of the variables, and without any timeout. The Large Neighborhood definition we have chosen is the following: given a number $N < |\mathscr{X}|$ and given a solution in $\mathbf{sol}(\mathscr{O})$ we randomly choose a set $\mathbf{FV} \subseteq \mathscr{X}$ of free variables, so that $|\mathbf{FV}| = N$. Therefore, the exploration of the neighborhood consists in the possible assignments to the constrained variables $\mathbf{FV}$ and it is regulated by the value of N and by the number `maxF` of maximum number of failures in an improving stage. At each stage variables and values selection are the same used for the first solution.

6.2 Parameters tuning and data analysis

For all instances of each problem we perform just one run of GECODE with a timeout of one hour. Let us observe that even in MEB and CTT some random choices for tie breaking are used, GECODE is in fact deterministic.
We tried several LNS approaches, that differs on the parameters N and `maxF`. N is determined as a proportion of $|\mathscr{X}|$ (10%, 20%, 30%, etc.). For relating the number `maxF` to the exponential growth of the search space w.r.t. the number N of free variables, we calculate $\texttt{maxF} = 2^{\sqrt{N * \texttt{Mult}}}$, and allow to fix the values for the parameter `Mult`. Reasonable values for `Mult` range from 0.01 to 2. Of course, these values cannot be chosen independently of the size of the problem instance and of the problem difficulty. These values are given by optional arguments of the command-line call to the execution.
The range of parameter we have tested is the following:
ATSP: $N \in \{20\%, 25\%, 30\%, 35\%, 40\%, 45\%\}$ and $\texttt{Mult} \in \{1, 1.5, 2\}$
MEB: $N \in \{35\%, 40\%, 45\%, 50\%\}$ and $\texttt{Mult} \in \{0.5, 0.75, 1\}$
CTT: $N \in \{2\%, 3\%, 4\%, 5\%, 10\%, 15\%\}$ and $\texttt{Mult} \in \{0.1, 0.5, 1\}$
Choosing a range to analyze can be done by few preliminary runs setting $\texttt{Mult} = 1$ and start by small N (e.g. 1% and then double it iteratively) in which the number k of consecutive idle iterations is kept low (e.g., 20).
With small values of N the algorithm performs little improvements at each step (and frequently a number of steps without improvements). Execution stops soon at high values of the objective function.
With increasing values of N the running time become slower, but one might notice the computation of better optima. At a certain point one notice that the algorithm is slower and slower, but, even worse, the optimum is not improved. In the next subsection we will come back on this training stage.
Once an interval of "good" behavior is found, one might enlarge the number k of consecutive idle iterations allowed before forcing the termination (in all our tests $k = 50$) and starting tests with different values of `Mult`.
Since LNS and pure LS computations are stochastic in nature, we collect the results of 20 runs in order to allow for basic statistical analyses. During each run, we stored the values of the objective function that correspond to improvements of the current solution, together with the running time spent. These data have been aggregated in

order to analyze the average behavior of the different LNS and pure LS strategies on each sample. To this aim we performed a discretization of the data in regular time intervals; subsequently, for each discrete interval we computed the average value of the objective function.
We experimentally determine that the best parameter combinations are: $N = 35\%$ and $\texttt{Mult} = 2$ for ATSP; $N = 50\%$ and $\texttt{Mult} = 0.5$ for MEB; $N = 5\%$ and $\texttt{Mult} = 0.5$ for CTT. We show the plot representing the behavior of the various parameters in the CTT problems (see also Figure 3). Diagrams for other problems are similar. Further LNS experiments (either in GELATO or Comet) have been run with this parameters setting.

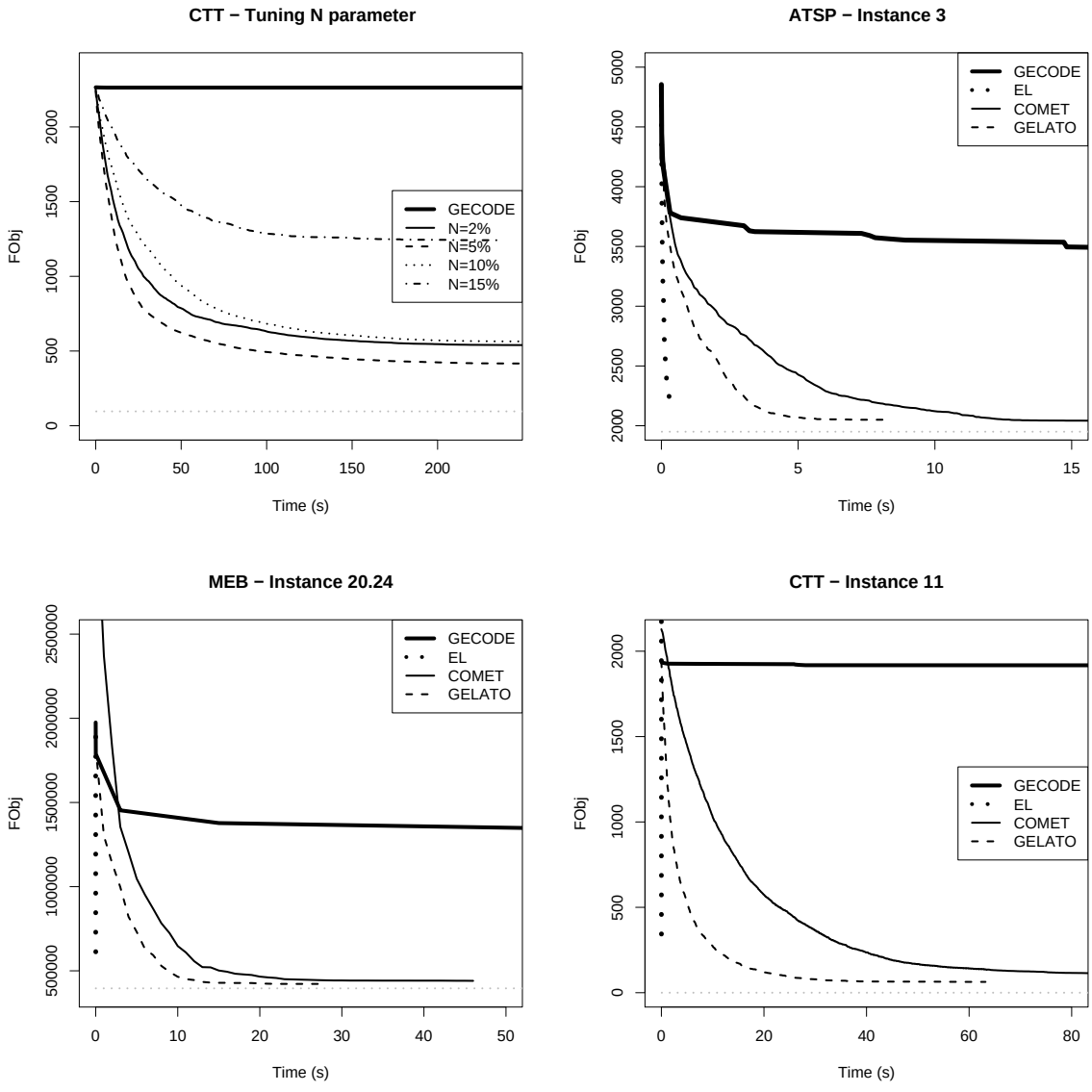

Fig. 3. Parameters and methods comparison (FObj is f)

6.3 Comparison

Let us briefly analyze the results of the comparison of GECODE, hill-climbing EasyLocal++ , GELATO, and Comet. In Figure 3 we show an excerpt of these results.[7] In those pictures, the horizontal dotted line represents the best known solution for the instance considered.

[7] An exhaustive comparison can be found at http://www.dimi.uniud.it/GELATO.

From the results, we can make some observations. First, it is clear that pure CP (GECODE) is unsuitable to improve the objective function f and to reach a good solution in reasonable time, while it is very useful to provide a good starting point for LS/LNS approaches. Conversely, hill-climbing is very fast (providing large cost improvements in few seconds) but for difficult instances it falls quickly in a local minimum and stops improving the solution.
LNS seems to be the most effective approach: for difficult instances it finds better solution than hill-climbing, because it can perform a deeper search and fall (later) into higher quality local-optima. Moreover, within this class the GELATO implementation has a behavior comparable to those of Comet[8]. This happens also in the majority of the other tests we omitted due to lack of space.
The fact that LNS does not work well with small values of N is due to the small size of the neighborhood: even though GECODE is able to visit it within the allowed time, it is difficult to make improvements when too few free variables are allowed to change. Conversely, when N is large also the spaces to be analyzed are bigger, allowing several variables to change. In these spaces it is easier to lead to better solutions, but at the same time it might happen that an improving solution is hidden within a huge space and it is not reached within the number of failures allowed. Obviously this trade-off depends on the problem at hand and has to be investigated experimentally. However, on the basis of the analysis performed in this paper, we find out that a set of reasonable values for the number of variables and the size of the search spaces is $N = 10\%, \texttt{mafF} = 1$. We set these value as default for our tool.
The constraint models employed in these tests were obtained by a direct encoding in GECODE. However, since GELATO is a multi-paradigm language other modeling languages could be employed. To evaluate this possibility, we also encode the CTT model in SICStus Prolog and compile it to GECODEusing the translator presented in [4]. The outcoming GECODE model was tested in the same settings as the previously presented solvers.
As one might expect, a direct encoding allows more efficient executions. However, running time has a behavior similar to those reported in Figure 3, but in this case the performances (in terms of running times) are slightly worse than those obtained by Comet.

7 Conclusions and future developments

We have presented a multi-language tool for combinatorial optimization, called GELATO, which is able to deal with CSP/COP models expressed in different modeling languages and to use a combination of Local Search and Constraint Programming techniques for solving them. Being built upon GECODE the user might exploit the emerging literature of problems already encoded in GECODE and use GELATO for speeding-up the search. Being also build upon EasyLocal++, the user can inherit various Local Search techniques from EL, without the need of reformulating the model.
The immediate future work is the first release of the tool that will be made available in the near future. We will improve the quality of the GECODE models obtained through the translation of the Prolog/MiniZinc models and to simplify the compilation task (currently demanded to many routines written in different languages).
We also plan to develop a more clever technique for determining good candidates for the values of the tool parameters so as to allow the user to exploit the solver as a black-box. However, being completely written in C++ the skilled programmer can use it as a free fully configurable search engine.

Acknowledgments

This work is partially supported by MUR-PRIN project "*Innovative and multi-disciplinary approaches for constraint and preference reasoning*". We would like to thank Andrea Schaerf, Tom Schrijvers, and Guido Tack for their precious advises in several phases of the developing of the tool.

References

1. Emile Aarts and Jan K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester (UK), 1997.
2. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, Cambridge (UK), 2003.
3. Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. A Hybrid Solver for Large Neighborhood Search: Mixing Gecode and EASYLOCAL++. In Maria J. Blesa Aguilera et al., editor, *Hybrid Metaheuristics*, volume 5818 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2009.

[8] Somehow slightly better than Comet.

4. Raffaele Cipriano, Agostino Dovier, and Jacopo Mauro. Compiling and executing declarative modeling languages to gecode. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 744–748, Berlin, Heidelberg, 2008. Springer-Verlag.
5. Luca Di Gaspero and Andrea Schaerf. Writing local search algorithms using EASYLOCAL++. In Stefan Voß and David L. Woodruff, editors, *Optimization Software Class Libraries*, OR/CS. Kluwer Academic Publisher, Boston (MA), USA, 2002.
6. Luca Di Gaspero and Andrea Schaerf. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software — Practice & Experience*, 33(8):733–765, July 2003.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
8. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
9. K. Marriot, P. J. Stuckey, L. De Koninck, and H. Samulowitz. An introduction to minizinc. http://www.g12.cs.mu.oz.au/minizinc/downloads/doc-1.3/minizinc-tute.pdf.
10. Barry McCollum, Andrea Schaerf, Ben Paechter, Paul McMullan, Rhyd Lewis, Andrew J. Parkes, Luca Di Gaspero, Rong Qu, and Edmund K. Burke. Setting the research agenda in automated timetabling: The second international timetabling competition. *INFORMS Journal on Computing*, 22(1):120–130, 2010.
11. Laurent Michel and Pascal Hentenryck. The comet programming language and system. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 881–881. Springer Berlin / Heidelberg, 2005. 10.1007/11564751/119.
12. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessiere, editor, *CP 2007*, volume 4741 of *LNCS*, pages 529–543, 2007.
13. Swedish Institute of Computer Science. Sicstus prolog. www.sics.se/sicstus.html.
14. Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.
15. Gecode Team. Gecode: Generic constraint development environment. http://www.gecode.org.
16. Gecode Team. Gecode reference documentation. http://www.gecode.org/doc-latest/reference/index.html.
17. Gecode Team. Modeling and programming with gecode. http://www.gecode.org/doc-latest/MPG.pdf.
18. Universität Heidelberg, Institut für Informatik. TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.
19. Steffen Wolf and Peter Merz. Evolutionary Local Search for the Minimum Energy Broadcast Problem. In Jano van Hemert and Carlos Cotta, editors, *EvoCOP 2008 – Eighth European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 4972 of *LNCS*, pages 61–72, Naples, Italy, mar 2008. Springer.
20. David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
21. Pieter Wuille and Tom Schrijvers. Monadic Constraint Programming with Gecode. In *Proc. of ModRef 2009, Eighth International Workshop on Constraint Modelling and Reformulation*, Lisbon, Portugal, September 2009.