A Hybrid Solver for Large Neighborhood Search: Mixing Gecode and EasyLocal^{++}

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

[1] DIMI `(cipriano|dovier)@dimi.uniud.it`
[2] DIEGM `l.digaspero@uniud.it`
Università di Udine, via delle Scienze 208, I-33100, Udine, Italy

Abstract. We present a hybrid solver (called GELATO) that exploits the potentiality of a Constraint Programming (CP) environment (Gecode) and of a Local Search (LS) framework (EasyLocal^{++}). GELATO allows to easily develop and use hybrid meta-heuristic combining CP and LS phases (in particular Large Neighborhood Search). We tested some hybrid algorithms on different instances of the Asymmetric Traveling Salesman Problem: even if only naive LS strategies have been used, our meta-heuristics improve the standard CP search, in terms of both goodness of the solution reached and execution time. GELATO will be integrated into a more general tool to solve Constraint Satisfaction/Optimization Problems. Moreover, it can be seen as a new library for approximate and efficient searching in Gecode.

1 Introduction

Combinatorial problems like planning, scheduling, timetabling and, in general, resource management problems, are daily handled by industries, companies, hospitals and universities. Their intractability however poses challenging problems to the programmer: ad-hoc heuristics that are adequate for one problem are often useless for others [16]; classical techniques like integer linear programming (ILP) need a big tuning in order to be effective; any change in the specification of the problem requires restarting almost from scratch. In the last years many efforts have been put in the development of general techniques that allow high-level primitives to encode search heuristics. Noticeable examples of these techniques are constraint programming (CP—with the various labeling heuristics) [12] and local search (LS—with the various techniques to choose and visit the neighborhood). We are not dealing with the problem of finding the optimum of a problem but with a *reasonable* solution to be computed in reasonable time. The future seems to stay in the combinations of these techniques in order to exploit the best aspect of each technique for the problem at hand (after a tuning of the various parameters on small instances of the problem considered). This is also witnessed by the success of the CP-AI-OR meetings [15]. In particular, Large Neighborhood Search (LNS) can be viewed as a particular heuristic for local search that strongly relies on a constraint solver [3] and it is a reasonable way to blend the inference capabilities of LS and CP techniques.

In this work we develop a general framework called GELATO that integrates

CP and LS techniques, defining a general LNS meta-heuristic that can be modified, tuned and adapted to any optimization problem, with a limited programming effort. Instead of building a new solver from scratch, we based our framework on two state-of-the-art, existing systems: the Gecode CP environment [13] and the LS framework EasyLocal^{++} [5]. We choose these two systems (among other ones, such as [11,10,7]) because both of them are free and open, strong and complete, written in C^{++}, and with a growing community using them.

We show how to model a LNS algorithm combining CP and LS in $\mathbb{GELATO}$: this modeling is easy to implement and it allows efficient computations. We test the framework on hard Asymmetric Travel Salesman Problem instances and show its effectiveness w.r.t. the traditional use of a pure CP approach.

The results of this paper will be combined to those of [2] so as to obtain a multi-language system able to model and solve combinatorial problems. This tool will comprise three main parts: a modeling component, a translator, and a solver. In the modeling component the user will be able to define in a high-level style the problem and the instance he/she wants to solve and the algorithm to use (CP search, possibly interleaved with LS, integer linear programming, heuristics or meta-heuristics phases). The translation component will handle the compilation of the model and the meta-algorithm defined by the user into the solver frameworks, like Gecode or others. In the third phase, the overall compiled program will be run on the problem instance specified by the user and the various solvers will interact as set by the user in the model. A side-effect of our work is a new library of search strategies for the Gecode system. Gecode is a free development environment for CP; $\mathbb{GELATO}$ provides some primitives that allow a Gecode user to easily develop approximate search algorithms, based on LS (in particular, with LNS). On the other hand, $\mathbb{GELATO}$ can be considered also as a plug-in for the EasyLocal^{++} system, a framework that allows the user to easily develop, test and combine several Local Search algorithms. With $\mathbb{GELATO}$, an EasyLocal^{++} user can easily exploit the benefits of CP in its LS algorithms.

2 Preliminary concepts

Discrete or continuous problems can be formalized using the concept of *Constraint Satisfaction Problem* (CSP). A CSP is defined as the problem of associating values (taken from a set of domains) to variables subject to a set of constraints. A *solution* of a CSP is an assignment of values to all the variables so that all the constraints are satisfied. In some cases not all solutions are equally preferable, but we can associate a cost function to the variable assignments. In these cases we talk about *Constraint Optimization Problems* (COPs), and we are looking for a solution that minimizes the cost value. The solution methods for CSPs and COPs can be split into two categories:

- *Complete methods*, which systematically explore the whole solution space in search of a feasible (for CSPs) or an optimal (for COPs) solution.
- *Incomplete methods*, which rely on heuristics that focus only on some areas of the search space to find a feasible solution (CSPs) or a “good” one (COPs).

2.1 Constraint Programming basics

Constraint Programming (CP) [12] is a declarative programming methodology parametric on the constraint domain. Combinatorial problems are usually encoded using constraints over *finite domains*, currently supported by all CP systems (e.g., [13,10,11]).

A CSP $\mathcal{P}$ is modelled as follows: a set $X = \{x_1, \ldots, x_k\}$ of *variables*; a set $D = \{D_1, \ldots, D_k\}$ of *domains* associated to the variables (i.e., if $x_i = d_i$ then $d_i \in D_i$); a set $\mathcal{C}$ of *constraints* (i.e., relations) over $dom = D_1 \times \cdots \times D_k$. $\langle d_1, \ldots, d_k \rangle \in dom$ satisfies a constraint $C \in \mathcal{C}$ iff $\langle d_1, \ldots, d_k \rangle \in C$. A tuple $d = \langle d_1, \ldots, d_k \rangle \in dom$ is a solution of a CSP $\mathcal{P}$ if d satisfies every constraint $C \in \mathcal{C}$. The set of solutions of $\mathcal{P}$ is denoted with $sol(\mathcal{P})$. If $sol(\mathcal{P}) \neq \emptyset$, then $\mathcal{P}$ is consistent. Often a CSP is associated to a function $f : sol(\mathcal{P}) \to E$ where $\langle E, \leq \rangle$ is a well-ordered set (e.g., $E = \mathbb{N}$ or $E = \mathbb{R}$). A COP is a CSP with an associated function f. A solution for this COP is a solution $d \in sol(\mathcal{P})$ that minimizes the function f: $\forall e \in sol(\mathcal{P})(f(d) \leq f(e))$.

This paradigm is usually based on *complete methods* that analyze the search space alternating deterministic phases (constraint propagation—values that cannot be assigned to any solution are removed by domains) and non-deterministic phases (variable assignment—a variable is selected and a value from its domain is assigned to it). This process is iterated until a solution is found or unsatisfiability is reached; in the last case the process backtracks to the last choice point (i.e., the last variable assignment) and tries other assignments.

2.2 Local Search basics

Local Search (LS) methods (see [1,4]) are a family of meta-heuristics to solve CSPs and COPs, based on the definition of *proximity* (or *neighborhood*): a LS algorithm typically moves from a solution to a near one, trying to improve an objective function, iterating this precess. LS algorithms generally focus the search only in specific areas of the search space, so they are *incomplete methods*, in the sense that they do not guarantee to find a feasible (or optimal) solution, but they search non-systematically until a specific stop criterion is satisfied.

To define a LS algorithm for a given COP, three parameters must be defined: the *search space*, the *neighborhood relation*, and the *cost function*. Given a COP P, we associate a *search space* S to it, so that each element $s \in S$ represents a solution of P. An element s is a *feasible* solution iff it fulfills the constraints of P. S must contain at least one feasible solution.

For each element $s \in S$, a set $\mathcal{N}(s) \subseteq S$ is defined. The set $\mathcal{N}(s)$ is called the *neighborhood* of s and each member $s' \in \mathcal{N}(s)$ is called a *neighbor* of s. In general $\mathcal{N}(s)$ is implicitly defined by referring to a set of possible *moves*, which define transitions between solutions. Moves are usually defined in an intensional fashion, as local modifications of some part of s. A *cost function* f, which associates to each element $s \in S$ a value $f(s) \in E$, assesses the quality of the solution. f is used to drive the search toward good solutions and to select the move to perform at each step of the search. For CSP problems, the cost function

f is generally based on the so-called *distance to feasibility*, which accounts for the number of constraints that are violated. A LS *algorithm* starts from an initial solution $s_0 \in S$, and uses the moves associated with the neighborhood definition to navigate the search space: at each step it makes a transition between one solution s to one of its neighbors s', and this process is iterated. When the algorithm makes the transition from s to s', we say that the corresponding move m has been accepted. The selection of moves is based on the values of the cost function and it depends on the specific LS technique.

Large Neighborhood Search (LNS) is a LS method that relies on a particular definition of the neighborhood relation and of the strategy to explore the neighborhood. Differently from traditional LS methods, an existing solution is not modified just by making small changes to a limited number of variables (as is typical with LS move operators), instead a subset of the problem is selected and searched for improving solutions. The subset of the problem can be represented by a set FV of variables, that we call *free variables*, which is a subset of the variables X of the problem. Defining FV corresponds to define a neighborhood relation.

For example, a LS move could be to swap the values of two variables, or, more generally the permutation of the values of a set of variables. Another possibility for a neighborhood definition is to keep the values of some variables and to leave the other variables totally free, constrained only by their domains.

Three aspects are crucial in LNS definition, w.r.t. the performance of this technique: *(1) which and (2) how many variables* have to be selected (i.e., the definition of FV), and *(3) how to perform the exploration* on these variables. Let us briefly analyze these three key-points, starting from the third one.

(3) Given FV, the *exploration* can be performed with any searching technique: CP, Operation Research algorithms, and so on. We can be interested in searching for: the best neighborhood; the best neighborhood within a certain exploration timeout; the first improving neighborhood; the first neighborhood improving the objective function of at least a given value, and so on. (2) Deciding *how many variables* will be free ($|FV|$) 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 with very little improvement of the objective function. Otherwise, a big FV can lead to big improvement at each step, but every single exploration can take a lot of time. This trade-off should be investigated experimentally, looking at a dimension of FV that leads to fast enough explorations and to good improvements. Obviously, the choice of $|FV|$ is strictly related to the search technique chosen (e.g., a strong technique can manage more variables than a naive one) and to the use or not of a timeout. (1) 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 naive 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.

2.3 Hybridization of CP and LS

Two major types of approaches to combine the abilities of CP and LS are presented in the literature [6,8]:

1. a systematic-search algorithm based on constraint programming can be improved by inserting a LS algorithm at some point of the search procedure, e.g.: (a) at a leaf (i.e., on complete assignments) or on an internal node (i.e., on a partial assignment) of the search tree explored by the constraint programming procedure, in order to improve the solution found; (b) at a node of the search tree, to restrict the list of child-nodes to explore; (c) to generate in a greedy way a path in the search tree;
2. a LS algorithm can benefit of the support of constraint programming, e.g.: (a) to analyze the neighborhood and discarding the neighboring solutions that do not satisfy the constraints; (b) to explore a fragment of the neighborhood of the current solution; (c) to define the search of the best neighboring solution as a problem of constrained optimization (COP).

In these hybrid methods one of the two paradigms is the master and the second one acts as a slave, supporting the master at some point of the search algorithm. Paradigms not based on the master-slave philosophy have also been proposed. In [9] LS and constraint propagation are split in their components, allowing the user to manage the different basic operators (neighborhood exploration, constraint propagation and variable assignment) at the same level. In [7] CP and LS are combined in a programming language (COMET), that supports both modeling and search abstractions, and where constraint programming is used to describe and control LS.

LNS is a LS strategy that can be naturally implemented using CP, leading to hybrid algorithms that involves the approaches 2(a), 2(b) and 2(c) of the above enumeration. CP can manage and exhaustively explore a lot of variables subjected to constraints, so it is a perfect method for the exploration of large neighborhoods.

2.4 The Solvers used

There are several tools commonly used by the community to solve CSPs and COPs with the CP and LS paradigms. We focused on Gecode , for the CP aspects, and EasyLocal^{++}, for the LS algorithms, for three main reasons: goodness of the solvers (they both are very strong, complete and efficient); guarantee of maintenance during time (they have a growing user community and they are actively maintained by their developers); ease of integration (they are free, open and written in C++, so they can be employed as C++ libraries). Moreover, all these characteristics ensure the possibility in the future of easily integrating these solvers with other C++ libraries. Here we can give only a short explanation of these two solvers, suggesting the reader to take a look at [13] for Gecode and [5] for EasyLocal^{++}.

Gecode It is an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications. It is implemented in C++

and offers competitive performances w.r.t. both runtime and memory usage. It implements a lot of data structures, constraints definitions, and search strategies, allowing also the user to define his own ones.

In the Geode philosophy, a model is implemented using *spaces*. A space is the repository for variables, constraints, objective function, searching options. Being C++ an object-oriented language, the modeling approach of Gecode exploits the *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., performing a copy of the space, 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, to find the best solution in the whole search space.

EasyLocal^{++} It is an object-oriented framework that allows to design, implement and test LS algorithms in an easy, fast and flexible way. The basic idea of EasyLocal^{++}(briefly EL) is to capture the essential features of most LS meta-heuristics, and their possible compositions. This allows the user to address the design and implementation issues of new LS heuristics in a more principled way.

The frameworks are characterized by the inverse control mechanism: the functions of the framework call the user-defined ones and not the other way round. The framework thus provides the full control structures for the invariant part of the algorithms, and the user only supplies the problem specific details.

Modeling a problem using EL means to define the C++ classes representing the basic concepts of a LS algorithm: e.g., the structure of a solution (or *State*, in the EL language), the neighborhood (or *Move*), the cost function, the strategy to navigate the neighborhoods (*NeighborhoodExplorer*). Once these basic concepts are defined (in C++ classes that inherit from EL base classes and implement their characteristics), the user selects the desired LS algorithm and run it. EL provides a wide range of LS heuristic and meta-heuristic (Hill Climbing, Steepest Descent, Tabu Search, Multi Neighborhood, ...), allowing also the development of new ones.

3 A Hybrid Solver for Large Neighborhood Search

In this section we describe the hybrid solver we have developed to implement LNS algorithms that exploit a CP solver in the exploration of the large neighborhoods; we called it $\mathbb{GELATO}$ (**G**ecode+**E**asy**L**ocal = **A** **T**ool for **O**ptimization). We first define the basic elements of this tool, in a general way, without any implementation detail. Then we explain the choices we made to integrate these tools in a unique framework.

3.1 The key ingredients of the solver

$\mathbb{GELATO}$ is made-up of five main elements (see Fig. 1): a *constraint model*, a *move enumerator*, a *move performer*, a *CP solver*, and a *LS framework*. The con-

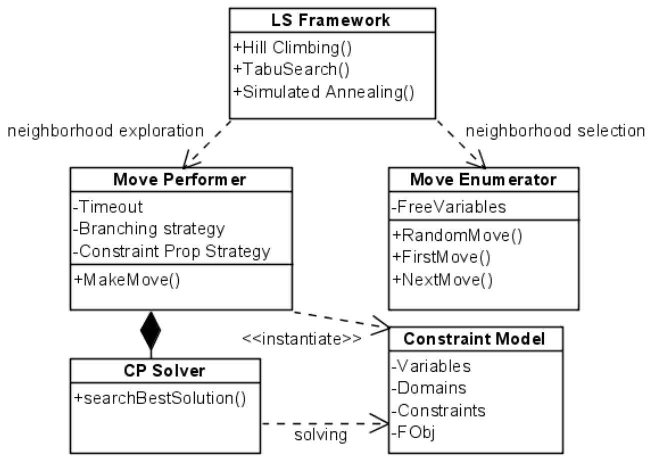

Fig. 1: A UML class diagram of the main components of 𝔾𝔼𝕃𝔸𝕋𝕆

straint model specify the problem we want to solve, according with the definition given in Section 2.1. The move enumerator and the move performer define the LNS characteristics (definition of the neighborhood and how to explore it, as in 2.2). The CP solver takes care of exploring the neighborhood. The LS framework manages all these element together into a LS algorithm. We briefly analyze each element.

Constraint model Here we define the variables, the domains, the constraints and objective function of the problem we want to solve. At these level we do not specify neither the instance input characteristics, that will be passed to the model as a parameter at run time, nor the search options (variable and value selection, timeout, ...), that will be specified by the *move performer* when the model will be actually solved. In 𝔾𝔼𝕃𝔸𝕋𝕆, the constraint model is specified using the Gecode language, but in general it can be expressed using any high-level modeling language (SICStus Prolog, Minizinc, OPL) or low level language, according to the CP solver that will be used. This model will be actually instantiated and solved by the CP solver during the exploration of a large neighborhood. It can be also used to find good starting solution for the main LS algorithm (hybrid approach 1(a) in the enumeration of section 2.3).

Move enumerator (m.e.) It deals with the definition of the set FV for the LNS, specifying which variable of the constraint model will be free in the exploration of the neighborhood. According with the *LS framework*, we can specify several kind of move enumerator: e.g., a random m.e., that randomly selects a specified number of variables of the problems; an iterative m.e., that iterates among all the N combination on the variables of the problem; a sliding m.e., that considers sliding window of K variables (i.e., $FV = \{X_1, \ldots, X_{1+k}\}$, then $FV = \{X_2, \ldots, X_{2+k}\}$ and so on).

Move performer (m.p.) A move performer collects all the information about how to search a large neighborhood, like searching timeout, variable and value selection, constraint propagation policy, branching strategies and so on. It instantiates the *constraint model*, according to the definition of the problem, the instance, the move (given by the m.e.), and the CP solver. It invokes the CP solver passing all these information and obtain the result of the exploration.

CP solver It is attached to a m.p. and must be able to recognize the *constraint model* specified and to perform an exploration on the neighborhoods, with the searching parameters given by them m.p.. It is expected to return the best neighborhood found (according to the m.p. policy) and the value of the objective function for it. In $\mathbb{GELATO}$ the *CP solver* used is Gecode, but in principle, there is no restriction about what kind of solver to use: a solver that can be easily interfaced with the other components could be a good choice.

LS framework It defines the high level interaction between the above components, building up a real LS algorithm: it navigates the searching space through the neighborhoods defined by the m.e., exploring them using the m.p.. Any traditional LS search algorithm can be used at this step (e.g., Hill Climbing, Steepest Descent, Tabu Search). Also meta-heuristic LS algorithm are allowed (e.g., Multi-Neighborhood, Token Ring Search), if different kinds of m.e. and m.p. are defined for the same problem. The *LS framework* used in $\mathbb{GELATO}$ is EasyLocal^{++}.

3.2 Mixing Gecode and EasyLocal^{++}

Gecode and EasyLocal^{++} are two frameworks that differs in many ways: they have different aims, use different data structures, and of course implement different algorithms. However, they have some architectural similarities: they both are written in C++; they both have base classes for the basic concepts and ask the user to define his problem implementing these base classes; they provide general algorithms (for CP and LS) that during executions call and use the derived classes developed by the user.

To define a hybrid tool for LNS, the natural choice is to use the EL framework as a main algorithm and Gecode as a module of EL that can be invoked to perform large neighborhood explorations.

According to the architecture of Gecode and EL and to the main components of the tools described in section 3.1, we have defined a set of classes (described below) that make possible the integration between the two frameworks. Only the first two (ELState and GEModel) must be implemented by the user (with derived class), because they contain the problem specific information; the other ones are provided by our tools and the user has only to invoke them (but he is also allowed to define his own ones).

ELState It is an EL state, that derives from the base class State of the EL framework. It typically contains a vector of variables that represents a solution, and an integer that contains the value of the objective function of the state.

GEModel It is the Gecode model, and it implements the methods requested to work with the Gecode language and the EL framework. It inherits methods and properties from the Gecode Space class.

LNSMove Base abstract class for the LNS moves of our framework. An actual move must implement the method `bool containsVar(Var)` that says if a given `Var` is free w.r.t. the move. We have defined two actual derived classes: DistrictMove and NDifferentMove. DistrictMove is made up by several sets of variables and of an active set: at a given time t only the variables of the active set are free; the active set can be changed by the move enumerator. NDifferentMove is made up by a single set of variables and a fixed integer N: at a given time t only N variables of the set are free (they can be chosen randomly or iterating between the variables in the set).

MoveEnumerator Base abstract class for m.e., that cycles between a particular kind of LNSMove. MoveEnumerator actual classes must implement the functions `RandomMove()`, `FirstMove()` and `NextMove()`. `RandomMove()`, given a LNSMove, selects randomly the set of free variables, according to the specific LNSMove definition (e.g., selecting randomly an active set of a DistrictMove). `FirstMove()` and `NextMove()` are used to cycle on all the possible moves (e.g., cycling on all the active neighborhoods of a DistrictMove, or cycling on all the N-combinations of the variables contained in a NDifferentMove). Our tool provides an actual MoveEnumerator for each LNSMove: a DistrictMoveME and a NDifferentME.

MovePerformer This module apply a given move to a given state, returning the new state reached. We define the GecodeMovePerformer, that takes a ELState and a LNSMove and, according to the defined GEModel class, builds up a Gecode Space and solves it.

LNSNeighborhoodExplorer It is the main class that EL uses for neighborhood explorations. It wraps together a MoveEnumerator and a MovePerformer.

LNSStateManager It provides some basics functionalities of an ELState, such as calculating the objective function of a state, and finding an initial state for the search. This last task is performed using CP: an instance of the GEModel of the problem is instantiated and explored until a first feasible solution is reached.

4 A simple experiment

We tested $\mathbb{GELATO}$ on instances of growing sizes of the Asymmetric Travel Salesman Problem, taken from the TSPLib [14]. In this section we describe the problem, the solving algorithms we used, the experiment we have performed, and the results obtained.

The *Asymmetric Travel Salesman Problem (ATSP)* is defined as follows: given a complete 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. We speak of asymmetric TSP if there exists (i, j) such that $c(i, j) \neq c(j, i)$ (imagine a climbing road).

4.1 CP Model, LNS Definition and LS algorithm for ATSP

The CP Model we used is chosen from the set of examples in the Gecode package. We used an existing model in order to point out the possibility of using $\mathbb{GELATO}$ starting from existing CP models, adding only some little modifications to them (e.g., changing the class headings and a couple of statements in the constructor).

The first state is obtained by a CP search over the same model, without any pre-assigned value of the variables, and without a timeout (because we need an initial feasible solution to start the LS algorithm).

The Large Neighborhood definition we have chosen is the following: given a number $N < |V|$ and given a solution (a tour on the nodes, i.e. a permutation of $|V|$), N variables are randomly selected and left free, while the other ones remain fixed. Therefore, the exploration of the neighborhood consists in exploring the N free variables and it is regulated by the following four parameters: (1) the variables with the smallest domain are selected, (2) the values are chosen randomly, (3) a timeout is set, and (4) the best solution found before reaching the timeout is returned.

The LS algorithm used is a traditional Hill Climbing, with a parameter K. At each step it selects randomly a set of N variables, frees them and searches on them for the best neighborhood, until the timeout expires. If the best neighborhood found is better or equal than the old one, it becomes the new current solution; otherwise (i.e., the value is worse), another random neighborhood is selected and explored (this is said *idle iteration*). The algorithm stops when K consecutive idle iterations have been performed (i.e., stagnation of the algorithm has been detected). Every single large neighborhood is explored using CP.

4.2 Experiments

The experiments have been performed on the following instances, taken from the TSPLib [14]: br17 (that we call *instance 0*, with $|V| = 17$), ftv33 (*instance 1*, $|V| = 34$), ftv55 (*instance 2*, $|V| = 56$), ftv70 (*instance 3*, $|V| = 71$), kro124p (*instance 4*, $|V| = 100$), and ftv170 (*instance 5*, $|V| = 171$).

The instances has been solved either using a pure constraint programming approach in Gecode or using different LNS approaches encoded in $\mathbb{GELATO}$. The LNS approaches differs for the number of variables of the neighborhood ($|FV|$): 5,10,15 for instance 0; 10,20,30 for instance 1; 10,20,30,40 for instances 2 and 3; 10,20,30,40,50 for instances 4 and 5. We tested different timeouts for the exploration of a single large neighborhood: one is $0.5N$ sec, the other is $0.25N$ sec. For instance 4 and 5 we also tested shorter timeouts. The number K of consecutive idle iteration before stop has been fixed to 50.

The pure Gecode CP uses the following parameters: (1) the leftmost variable is selected, and (2) the values are chosen increasingly. We run it only once, since it is deterministic, with a timeout of one hour. We also tried a first-fail strategy (the variable with the smallest domain is selected), but its performance are slightly worse than the leftmost one. This is probably due to the regular structure of the problem.

Since LNS computations uses randomness, we repeated each of them 20 times. During each run, we stored the values of the objective function, that corresponds 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 strategies on each group of 20 runs. To this aim we perform a discretization of the data on regular time intervals (with a step of $0.01s$ for instance 0 and $1s$ for the others); subsequently, for each discrete interval we compute the average value of the objective function on the 20 runs.

4.3 Results

Here we collect the results obtained in the experiments. Every picture shows the comparison of different algorithms on a single instance: in Fig. 2, the pure CP approach is compared with LNS with a timeout of $0.5|FV|s$ for the exploration of a single large neighborhood; in Fig. 3, the CP approach remains the same, but the timeout for LNS is set to $0.25|FV|s$. In Fig. 4 we concentrate on larger instances (4 and 5), analyzing three different timeouts for the exploration of a single neighborhood; we also considered a LNS with $|FV| = 50$, which has been experimentally proved to be effective for instance 5. We also tried neighborhoods with $|FV| = 60$ but they turned out to be computationally intractable (with too many variables the explorations of the neighborhood is not effective).

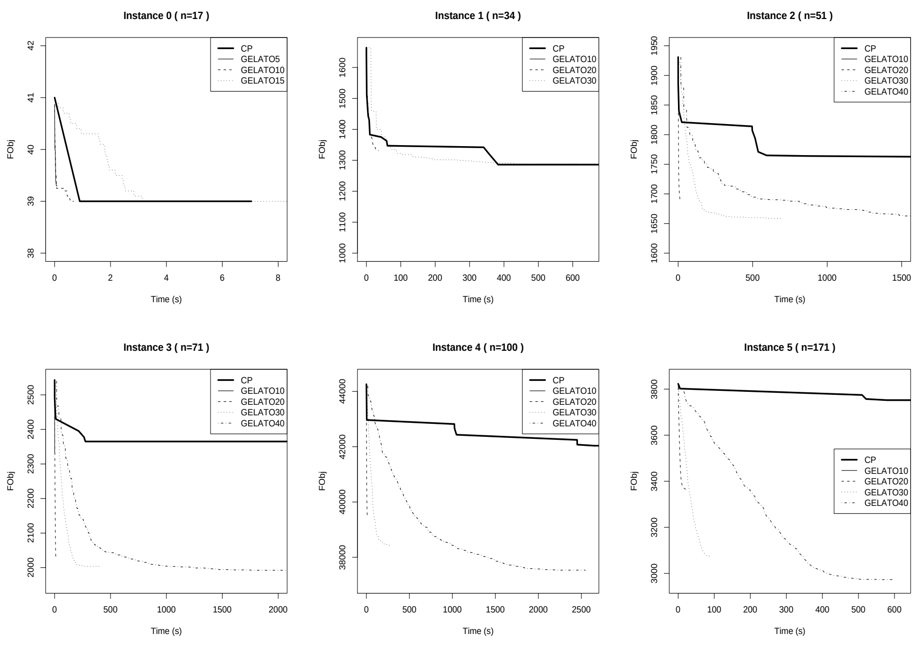

Fig. 2: Comparison between CP and 𝔾𝔼𝕃𝔸𝕋𝕆 (timeout for LNS is $0.5|FV|s$)

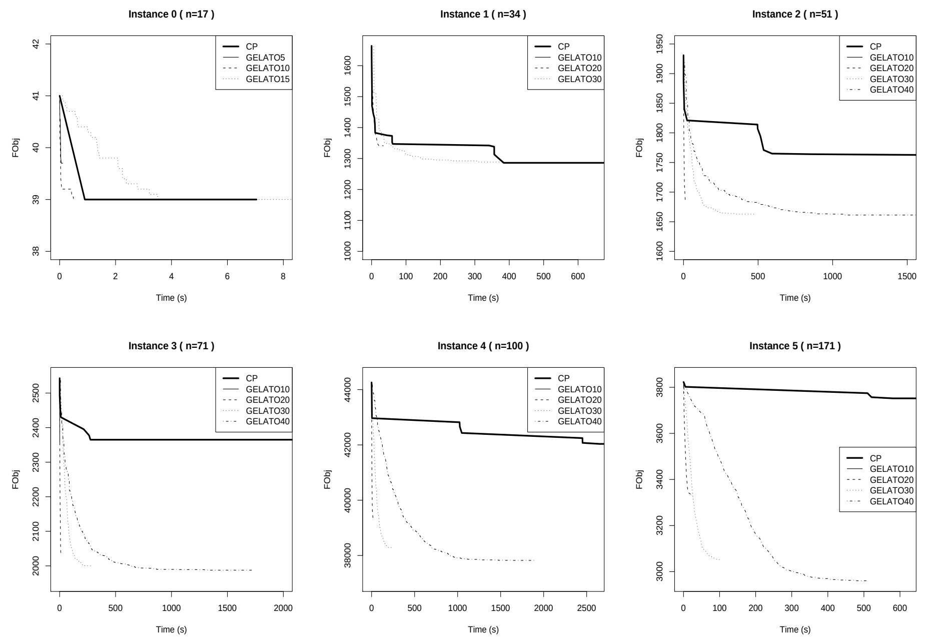

Fig. 3: Comparison between CP and $\mathbb{GELATO}$ (timeout for LNS is $0.25|FV|s$)

4.4 Discussion

Let us add some consideration about the results obtained, that can be summarized as: as instances grow, $\mathbb{GELATO}$ definitely outperforms Gecode (if we are happy with an approximate solution).

For the two small instance (instance 0 and 1), the two approaches are quite similar. In fact exploring a neighborhood of big dimension, very close to the whole search space (that with these instance is still tractable), is quite equivalent to solve the initial CP model. From instance 2 to the last, the trouble of CP in optimizing large search spaces comes out, and LNS gives better results.

The behavior of the CP algorithms is similar for all the instances: in the first seconds of the execution, CP finds some improving solutions, but from that moment it starts an exhaustive exploration of the search space without finding any other significant improving solution (the line of the objective function becomes quite horizontal). On the other hand, LNS algorithms have a more regular trend: the objective function is permanently improved during time, since a local minimum is found and then the algorithm is stopped (after 50 idle iterations).

Except for the LNS of 10 variables (too few), all the other LNS strategies find solutions improving the CP ones. They are also very competitive in time: some LNS algorithms overcome a lot the CP approach since the first seconds of the search (e.g. LNS with $|FV| = 20$ in instances 2,3,4,5, and LNS with $|FV| = 30$ in instances 3,4,5). This is due to the efficiency of LNS, that continuously explores different short parts of the search space, but in an exhaustive way, exploiting

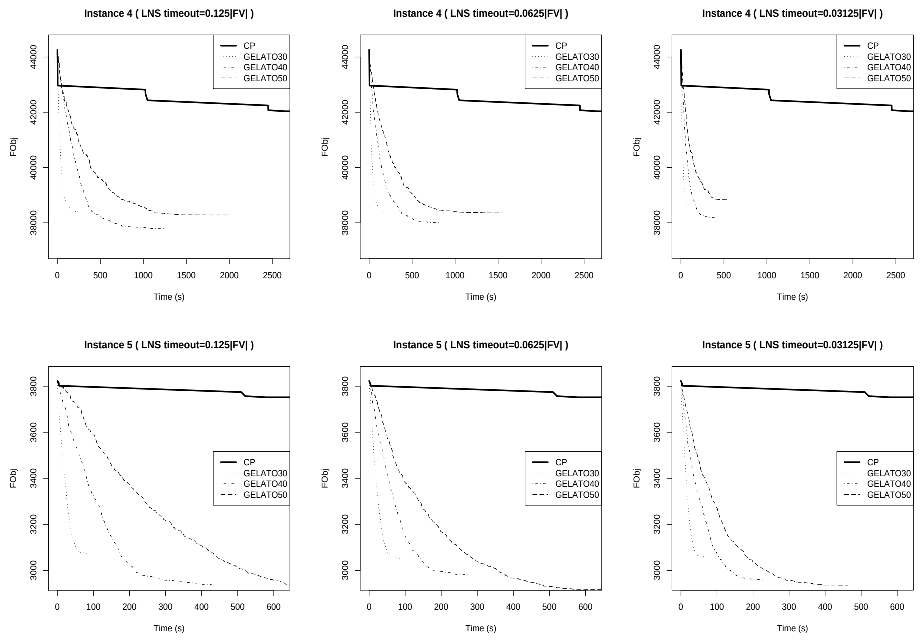

Fig. 4: Comparison between different LNS timeouts on big instances

the constraint programming approach in any single neighborhood. On the other hand, the CP approach explores the whole search tree, but it remains bordered on a limited portion of it, so any improving solution outside this portion cannot be reached in acceptable time.

Some comparison between the different LNS parameters (number of variables involved and timeout) can also be done: small LNS algorithms ("small" stands for LNS with few free variables) are very fast, and can make very big improvements in some seconds. Larger LNS algorithms are slower (they have bigger spaces to analyze), but they can reach better solutions: in fact they optimize a larger number of variables, so their exploration of the search space is more "global"; it could also happen that a neighborhood is too big to allow CP to perform an effective exploration. This trade-off must be investigates experimentally: e.g., from our tests it comes out that for instance 4, neighborhoods of size 40 are the best ones, while the ones of size 50 are too big, so ineffective.

The choice of the timeout for the exploration of a single large neighborhood is strictly connected to the dimension of the neighborhood: with small LNS (10, 20 or 30 variables) a long timeout seems to be better, because it can explore the whole (small) neighborhood and hopefully return the best neighbor. With bigger LNS (40,50 variables) it is the opposite: exploring the whole large neighborhood becomes a waste of time (at some point, constraint programming stops to find improvements), so it is better to stop earlier the search on the large neighborhood and continue quickly with another large neighborhood exploration.

A possible meta-heuristic that comes out from all these considerations could be the following:

1. start with small LNS with a short timeout(in this way we try to get the best improvement in the shortest time);
2. when no better solutions can be found, increase the timeout then launch Large Neighborhood Search (in this way an exploration has more time to explore the neighborhood and could fine better improvements);
3. iterate 2 since a big timeout value is reached;
4. increase the neighborhood's dimension, set the timeout the initial short value, then launch Large Neighborhood Search;
5. points 2–4 are iterated since the neighborhood's dimensions increases to intractable/ineffective ones.

This Multi-Neighborhood meta-heuristic should find large improving solutions in very short time (that can be quickly output to the user), and then run progressively deeper searches (even if more time expensive). Source codes and other technical details are in `http://tabu.diegm.uniud.it/EasyLocal++/`.

5 Conclusions and Future works

In this paper we showed that: using our $\mathbb{GELATO}$ hybrid framework, it is possible to combine a given CP model into a LS framework in a straightforward way; in particular, we can use a Gecode model as a base to define every kind of neighborhood, using the functionality provided by EasyLocal^{++} to execute competitive Large Neighborhood Search algorithms. We tested $\mathbb{GELATO}$ on several instances of the ATSP, and showed that performances of the hybrid LNS algorighms are very faster w.r.t. the pure CP approach, on all the non-trivial ATSP instances, even if the LS strategy is naive (Hill Climbing with random neighborhood selection). We also proposed a general Multi Neighborhood hybrid meta-heuristic that should improve the results we obtained so far.

We wish to extend the research pursued in this paper along two lines: developing and testing new hybrid algorithms; extending $\mathbb{GELATO}$ into a more general modeling framework. First of all we want to develop and test the Multi Neighborhood meta-heuristic proposed and other hybrid algorithms (e.g., based on Tabu Search, Steepest Descent, . . .). We will implement these algorithms using $\mathbb{GELATO}$ and test them on a set of benchmark problems, more structured than the ATSP, also trying new problem-specific neighborhood definitions.

Concerning the second research line, we want to simplify the class hierarchy of $\mathbb{GELATO}$ (some classes and functionality must be added, other ones need to be cleaned up and refactored). Once $\mathbb{GELATO}$ has a more simple and clear interface, it will be integrated into a more general modeling tool to easily model and solve CSPs and COPs, that we already presented in [2]. In this tool, the user will be able to *model* a problem in a high-level language (e.g., Prolog, Minizinc, OPL), and specify the meta-heuristic he want to use to solve the problem; the model and the meta-algorithm defined will be automatically *compiled* into the solver languages, e.g. Gecode and EasyLocal^{++}, since we use GELATO; at the end, the overall compiled program will be run and the various tools will interact in the way specified by the user in the modeling phase, to *solve* the instance of the problem modeled. We have already implemented two compilers from declarative languages to a low-level solvers (the Prolog-Gecode and MiniZinc-Gecode compilers presented in [2]). We want to extend the functionalities of the compilers already realized and develop the *modeling-translating-solving* framework described above. Once this high-level modeling framework is well tested and reliable, developing hybrid algorithms will be flexible and straightforward and their executions will benefit from the use of low level efficient solvers.

References

1. E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization.* John Wiley and Sons, Chichester, UK, 1997.
2. R. Cipriano, A. Dovier, and J. Mauro. Compiling and executing declarative modeling languages to Gecode . ICLP2008, LNCS 5366:744–748, Springer, 2008.
3. E. Danna and L. Perron. Structured vs. unstructured large neighborhood search. In *Principles and Practice of Constraint Programming*, CP 2003.
4. L. Di Gaspero. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools.* PhD thesis, Univ. di Udine, DIMI, 2003.
5. L. Di Gaspero and A. Schaerf. EasyLocal++: An object-oriented framework for flexible design of local search algorithms. *Software — Practice & Experience*, 33(8):733–765, July 2003.
6. F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In F. Glover and G. Kochenberger, eds, *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming, pages 369–403. Kluwer, 2003.
7. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, 2005.
8. N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristic. *Artificial Intelligence*, 139(1):21–45, 2002.
9. E. Monfroy, F. Saubion, and T. Lambert. On hybridization of local search and constraint propagation. ICLP 2004, LNCS 3132:299–313, Springer, 2004.
10. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. CP2007, LNCS 4741:529–543, 2007.
11. Swedish Institute of Computer Science. Sicstus prolog. http://www.sics.se/isl/sicstuswww/site/index.html.
12. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY, USA, 2006.
13. Gecode Team. Gecode : Generic constraint development environment. http://www.gecode.org.
14. Institut für Informatik Universität Heidelberg. Tsplib. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.
15. Various Authors. CP-AI-OR conference series. http://www.cpaior.org/.
16. D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.