Constraints (2014) 19:339–379
DOI 10.1007/s10601-014-9166-6

Explanation-based large neighborhood search

Charles Prud'homme · Xavier Lorca · Narendra Jussien

Published online: 20 July 2014

Abstract One of the most well-known and widely used local search techniques for solving optimization problems in Constraint Programming is the Large Neighborhood Search (LNS) algorithm. Such a technique is, by nature, very flexible and can be easily integrated within standard backtracking procedures. One of its drawbacks is that the relaxation process is quite often problem dependent. Several works have been dedicated to overcome this issue through problem independent parameters. Nevertheless, such generic approaches need to be carefully parameterized at the instance level. In this paper, we demonstrate that the issue of finding a problem independent neighborhood generation technique for LNS can be addressed using explanation-based neighborhoods. An explanation is a subset of constraints and decisions which justifies a solver event such as a domain modification or a conflict. We evaluate our proposal for a set of optimization problems. We show that our approach is at least competitive with or even better than state-of-the-art algorithms and can be easily combined with state-of-the-art neighborhoods. Such results pave the way to a new use of explanation-based approaches for improving search.

Keywords LNS · Explanations · Optimization · Neighborhoods

Electronic supplementary material The online version of this article (doi: 10.1007/s10601-014-9166-6) contains supplementary material, which is available to authorized users.

C. Prud'homme (✉) · X. Lorca
EMNantes, INRIA TASC, CNRS LINA, 44307 Nantes Cedex 3, France
e-mail: charles.prudhomme@mines-nantes.fr

X. Lorca
e-mail: Xavier.Lorca@mines-nantes.fr

N. Jussien
Télécom Lille, 59653 Villeneuve d'Ascq Cedex, France
e-mail: Narendra.Jussien@telecom-lille.fr

1 Introduction

Local search techniques are very effective to solve hard optimization problems. Most of them are, by nature, incomplete. In the context of constraint programming (CP) for optimization problems, one of the most well-known and widely used local search techniques is the Large Neighborhood Search (LNS) algorithm [25, 31]. The basic idea is to iteratively relax a part of the problem, then to use constraint programming to evaluate and bound the new solution. A generic and common way to reinforce diversification of LNS is to introduce restart during the search process. This technique has proven to be very flexible and to be easily integrated within standard backtracking procedures [22]. Various generic techniques have been studied in [23], but only one of them appear to be efficient in practice, which was defined by the authors as "accepting equivalent intermediate solutions in a search iteration instead of requiring a strictly better one". One drawback of LNS is that the relaxation process is quite often problem dependent. Some works have been dedicated to the selection of variables to relax through general concept not related to the class of the problem treated [5, 24]. However, in conjunction with CP, only one generic approach, namely Propagation-Guided LNS [24], has been shown to be very competitive with dedicated ones on a variation of the Car Sequencing Problem. Nevertheless, such generic approaches have been evaluated on a single class of problem and need to be thoroughly parametrized at the instance level, which may be a tedious task to do. It must, in a way, automatically detect the problem structure in order to be efficient.

During the last decade, explanation-based techniques have regained attention in CP. Explanations, in a nutshell, can be seen as an explicit trace of the propagation mechanism making it possible to identify a set of constraints and decisions (variable assignments, cuts) responsible for the current state of the domain of a variable [21, 33]. Explanations have been used to identify hidden structures in problem instances [2] and to improve search [12, 14]. However, explanations are quite intrusive when computed and quite space consuming when explicitly maintained during search.

In this paper, we show that the issue of finding a problem independent neighborhood generation technique for LNS can be addressed using explanations. A first contribution relies on generic, configuration-free approaches to choose variables to relax. One is based on an explanation of the inability to repair a solution, the other is based on an explanation of the non-optimal nature of the current solution. For this purpose we will show how classical explanation-based search techniques can be modified and simplified to be efficiently integrated within a standard backtrack-based algorithm. A second contribution is the operational implementation of those neighborhoods for further selecting variables to fix in a partial solution. We suggest three combinations of neighborhoods based on explanations and evaluate them on a set of optimization problems extracted from the MiniZinc distribution.[1] We show that our approach is at least competitive with or even better than state-of-the-art generic neighborhoods and can be easily combined with them. Finally, we evaluate an ultimate combination made of explanation-based neighborhoods and propagation-guided ones. This last combination performs slightly better than the individual approaches.

The paper is organized as follows: First, Section 2 introduces the required concepts to present our approach; Next, Section 3 details the explanation-based neighborhoods for LNS;

[1] http://www.minizinc.org/

Finally, Section 4 shows the improvements brought by our proposal with respect to the state-of-the-art CP approaches for LNS.

2 Background

Constraint programming is based on relations between variables, which are stated by constraints. A *Constraint Satisfaction Problem* (CSP) is defined by a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ and consists in a set of n variables $\mathcal{V}$, their associated domains $\mathcal{D}$, and a collection of s constraints $\mathcal{C}$. The domain $dom_v \in \mathcal{D}$ associated with a variable $v \in \mathcal{V}$ defines a finite set of integer values v can be assigned to. low_v (respectively, upp_v) denotes the lower bound (respectively, upper bound) of dom_v. The initial domain of v, its lower bound and its upper bound are denoted respectively Dom_v, Low_v and Upp_v. An *assignment*, or *instantiation*, of a variable v to a value x is the reduction of its domain to a singleton, $dom_v = \{x\}$; v^* denotes the value assigned to a variable v.

A constraint $c \in \mathcal{C}$ on k variables $(v_1, \dots, v_k)$ is a logic formula that defines allowed combinations of values for the variables $(v_1, \dots, v_k)$. A constraint c is equipped with a (set of) filtering algorithm(s), named *propagator*(s). A propagator removes, from the domains of $(v_1, \dots, v_k)$, values that cannot correspond to a valid combination of values. A solution of a CSP is an assignment of all its variables simultaneously verifying the constraints in $\mathcal{C}$.

Solving a CSP is usually performed with a tree-based search, basically, a depth first search algorithm. A branching decision for a CSP (a *decision* in the following) δ is a triplet $\langle v, o, x \rangle$ composed of a variable $v \in \mathcal{V}$ (not yet assigned), an operator o (most of the time "=") and a value $x \in dom_v$. This triplet can be considered as a unary constraint over dom_v.

Each time a decision is applied or negated, its impact is propagated through the constraint network of the CSP. After the propagation step, if the domain of a variable becomes empty (*domain wipe out*) or if no valid combination of values can be got for at least one constraint (*inconsistency*), there is no feasible solution within the current branch of the search tree. A classical search algorithm backtracks to a previous decision to negate it, if any, or eventually stops. If all the domains are reduced to singletons, a *solution* of a constraint network is found. Finally, if at least one domain is not reduced to a singleton, another decision is selected and applied. A *decision path* is a chronologically ordered sequence of decisions.

A *Constraint Optimization Problem* (COP) is a CSP augmented with a cost function over an objective variable o. The aim of a COP is to find a solution for which o is maximized or minimized. When a solution S is found for a COP, a *cut* C_S is posted on the objective variable. A cut states that the next solution should be *better* than the current one until the optimal value of the objective is reached. Most of the time, the initial domain of the objective variable is unbounded. Nevertheless, a convenient way to represent it is to store its bounds.

2.1 Large neighborhood search

The Large Neighborhood Search metaheuristic was proposed in [25, 31]. It was initially designed to compute *moves* for the Vehicle Routing Problem whose evaluation and validation were made thanks to a tree-based search [31]. LNS is a two-phase algorithm which partially *relaxes* a given solution and *repairs* it. Given a solution as input, the relaxation phase builds a *partial solution* (or *neighborhood*) by choosing a set of variables to reset to their initial domain; The remaining ones are assigned to their value in the solution.

This phase is directly inspired from the classical Local Search techniques [25]. Even though there are various ways to repair the partial solution, we focus on the original technique, proposed by [31], in which Constraint Programming is used to bound the objective variable and to assign a value to variables not yet instantiated. These two phases are repeated until the search stops (optimality proven or limit reached). While the implementation of LNS is straightforward, the main difficulty lies in the design of neighborhoods able to move the search further. Indeed, the balance between diversification (i.e., evaluating unexplored sub-tree) and intensification (i.e., exploring them exhaustively) should be well-distributed. The general behavior of LNS is described by Algorithm 1.

Algorithm 1 Large Neighborhood Search

```
Require: an initial solution $S$
1: procedure LNS
2:     while Optimal solution not found and a stop criterion is not encountered do
3:         RELAX($S$)
4:         $S' \leftarrow$ FINDSOLUTION()              ▷ The cut $C_S$ is automatically posted
5:         if $S' \neq$ NULL then                    ▷ An improving solution has been found
6:             $S = S'$
7:         end if
8:     end while
9: end procedure
```

Starting from an initial solution S, LNS selects and relaxes a subset of variables (RELAX, line 3). The current partial solution is then repaired in order to improve the current solution S (line 4). If such a solution S' is found (line 5), it is stored (line 6). These operations are executed until the optimal solution is found or a stopping criterion (for instance a time limit) is encountered (line 2). Note that proving the optimality of a solution is not what LNS is designed for.

Selecting the variables to relax is the tricky part of the algorithm. A random selection of the variables to unfix may be considered first [10, 31]. But, problem dedicated neighborhoods tend to be more efficient in practice. In [31], the authors solved Vehicle Routing Problems by selecting the set of customer visits to remove and re-insert. On the Network Design Problem, the structure of the problem is exploited to define accurate neighborhoods [3]. On the Job Shop Scheduling Problem, a neighborhood that deals with the objective function has been studied [5]; the sub-problems were solved with MIP. Another approach relies on a portfolio of neighborhoods and Machine Learning techniques to converge on the most efficient neighborhoods: it has been successfully evaluated on scheduling problems with specific neighborhoods [17]. In [23], the authors have tested several techniques to improve the global behavior of their LNS solver while solving the Car Sequencing Problem. The most remarkable technique, named *walking*, consists in accepting equivalent intermediate solutions. In [18], the authors use Reinforcement Learning to dynamically configure the size of the partial solution, the search limit and the selection of the neighborhoods. They compare various configuration and conclude on the selection of the two former parameters but the dedicated neighborhood of [23] still betters the class of problem treated (the Modified Car Sequencing Problem). Other classes of problem have been tackled using

LNS in the last decade: the Service Technician Routing and Scheduling Problems [16], the Pollution-Routing Problem [7], the Founder Sequence Reconstruction Problem [29], Strategic Supply Chain Management Problem [4], the Machine Reassignment Problem [20], to name the most recent ones.

Another approach is to design generic neighborhoods. In [24], sophisticated neighborhoods, based on the graph of variable dependencies, have been proposed. The authors introduced *propagation-guided* neighborhoods in which the volume of domain reduced, thanks to the propagation, helps to link variables together inside or outside partial solutions. Hence, they suggest three neighborhoods which, combined together, tackle a modified version of the Car Sequencing Problem. Even though it is not problem-dedicated, such an approach relies on an accurate parametrization of the heuristics: the initial size of the partial solution, its evolution during the resolution, number of variables "close" to the selected one. However, this approach is the reference while dealing with generic neighborhoods design. Most recently, a generic calculations of neighborhoods has been published in a workshop on Local Search Techniques [19]. The authors suggest generic approaches to automatically choose the subset of variables to fix, they obtain interesting results in comparison with the standard random choice. But no comparison with [24] has been done and there has been no further action on this preliminary work.

Another point to consider while designing neighborhoods is the size of the partial solution. If the relaxed part of the solution is very large, LNS relies too much on the tree-based search: finding a new solution depends more on the search strategy than on the neighborhoods, and finally suffers from poor *diversification*. On the contrary, if the size is too small, the tree-based search may not have enough *space* to explore and may have trouble finding solutions. Thus, in [31], the authors proposed to gradually increase the part of the solution to reconsider: it reinforces the *diversification* (it may also bring completeness to the search process).

In order to improve the robustness of LNS, the authors of [22] have shown that it is worth imposing a small search limit (for instance, a fail limit) during the reparation phase. If the search reaches the limit with no solution, a new neighborhood is generated and the search is launched again. Indeed, it is worth diversifying search by quickly computing a new neighborhood instead of trying to repair a unique one, without limitation nor guarantee of success. This method limits the trashing and reduces the tradeoff between diversification and intensification. Evaluations have shown that it helps finding better quality solutions.

In summary, the issues in LNS are twofold. The first one is to maintain a fair tradeoff between diversification and intensification in neighborhoods computation. This is commonly addressed by introducing randomization in the neighborhoods or alternating random neighborhoods with more sophisticated ones, combined with a fast restart policy.

2.2 Explanations

Nogoods and explanations have long been used in various paradigms for improving search [9, 12, 27, 30, 33]. An explanation records some sufficient information to justify an inference made by the solver (domain reduction, contradiction, etc.). It is made of a subset of the original propagators of the problem and a subset of decisions applied during search. Explanations represent the logical chain of inferences made by the solver during propagation in an efficient and usable manner. In a way, they provide some kind of a trace of the behavior of the solver as any operation needs to be *explained* [6].

Explanations have been successfully used for improving constraint programming search process. Both complete (as the `mac-dbt` algorithm [14]) and incomplete (as the `decision-repair` algorithm [12, 26]) techniques have been proposed. Those techniques follow a similar pattern: learning from failures by recording each domain modification with its associated explanation (provided by the solver) and taking advantage of the information gathered to be able to react upon failure by directly pointing to relevant decisions to be undone. Complete techniques, in this context, follow a most-recent based pattern while incomplete technique design heuristics to be used to focus on decisions more prone to allow a fast recovery upon failure.

Example 1 Let consider the following $COP = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$:

- $\mathcal{V} = \langle x_1, x_2, x_3, x_4, x_5, x_6, o \rangle$,
- $\mathcal{D} = \langle [0, 4], [0, 4], [-1, 3], [-1, 3], [0, 4], [0, 4], [0, 10] \rangle$ and
- $\mathcal{C} = \langle C1 \equiv \sum_{i=1}^{6} x_i = o, C2 \equiv x_1 \geq x_2, C3 \equiv x_3 \geq x_4$ and $C4 \equiv x_5 + x_6 > 3 \rangle$.
- where the objective is to minimize o.

The initial solution $S_1 = \langle 0, 0, 2, 0, 2, 2, 6 \rangle$ is found by applying the following decision path $P_{S_1} = (\delta_1, \delta_2, \delta_3, \delta_4, \delta_5)$, where $\delta_1 : \langle x_1, =, 0 \rangle$, $\delta_2 : \langle x_4, =, 0 \rangle$, $\delta_3 : \langle x_3, =, 2 \rangle$, $\delta_4 : \langle x_5, =, 2 \rangle$ and $\delta_5 : \langle x_6, =, 2 \rangle$. Table 1 depicts the search trace.

The first step (Step 1.a) describes the effect of the application of δ_1: x_1 is instantiated to 0, and thanks to $C2$, x_2 is also instantiated to 0 (Step 1.b). Then, the application of δ_2 instantiates x_4 to 0 (Step 2.a); It triggers the execution of the propagator of $C3$ which updates the lower bound of x_3 to 0 (Step 2.b). The application of δ_3 instantiates x_3 to 2 (Step 3.a), the lower bound of o is then updated to 2 because of $C1$ (Step 3.b). The application of δ_4 instantiates x_5 to 2 (Step 4.a), it triggers the execution of the propagator of $C4$ which updates the lower bound of x_6 to 2 (Step 4.b); Then, the update of the domain of o stems from those two previous modifications (Step 4.c). Finally, the application of δ_5 instantiates x_6 to 2 (Step 5.a). This events instantiates o to 6 by executing the propagator of $C1$ (Step 5.b).

Table 2 shows the explanations per value removals computed during the resolution of the COP when reaching solution S_1. A line separates the explanations of each variable, and a dash line separates groups of explanations, i.e., equivalent explanations for various value removals from the same variable. In that example, some variables are uniformly explained, e.g., every value removal from x_1 is explained by δ_1. That is also the case for x_2, x_4 and

Table 1 The trace of the search of the COP defined in Example 1

Step	Cause	Consequences
1.a	δ_1	$x_1 = [0, \mathbf{0}]$
1.b	$x_1 \wedge C2$	$x_2 = [0, \mathbf{0}]$
2.a	δ_2	$x_4 = [0, \mathbf{0}]$
2.b	$x_4 \wedge C3$	$x_3 = [\mathbf{0}, 3]$
3.a	δ_3	$x_3 = [\mathbf{2}, \mathbf{2}]$
3.b	$x_3 \wedge C1$	$o = [\mathbf{2}, 10]$
4.a	δ_4	$x_5 = [\mathbf{2}, \mathbf{2}]$
4.b	$x_5 \wedge C4$	$x_6 = [\mathbf{2}, 4]$
4.c	$x_5 \wedge x_6 \wedge C1$	$o = [\mathbf{6}, \mathbf{8}]$
5.a	δ_5	$x_6 = [\mathbf{2}, \mathbf{2}]$
5.b	$x_6 \wedge C1$	$o = [6, \mathbf{6}]$

Bold values highlight direct Consequences

Table 2 Explanations per value removals for all variables of the COP

Variable	{removed value} ← Explanation
x_1	$\{1\} \leftarrow \delta_1$,
	$\{2\} \leftarrow \delta_1$,
	$\{3\} \leftarrow \delta_1$,
	$\{4\} \leftarrow \delta_1$;
x_2	$\{1\} \leftarrow (\delta_1 \wedge C2)$,
	$\{2\} \leftarrow (\delta_1 \wedge C2)$,
	$\{3\} \leftarrow (\delta_1 \wedge C2)$,
	$\{4\} \leftarrow (\delta_1 \wedge C2)$;
x_3	$\{\text{-}1\} \leftarrow (\delta_2 \wedge C3)$,
	$\{0\} \leftarrow \delta_3$,
	$\{1\} \leftarrow \delta_3$,
	$\{3\} \leftarrow \delta_3$,
	$\{4\} \leftarrow \delta_3$;
x_4	$\{\text{-}1\} \leftarrow \delta_2$,
	$\{1\} \leftarrow \delta_2$,
	$\{2\} \leftarrow \delta_2$,
	$\{3\} \leftarrow \delta_2$,
	$\{4\} \leftarrow \delta_2$;
x_5	$\{0\} \leftarrow \delta_4$,
	$\{1\} \leftarrow \delta_4$,
	$\{3\} \leftarrow \delta_4$,
	$\{4\} \leftarrow \delta_4$;
x_6	$\{0\} \leftarrow (\delta_4 \wedge C4)$,
	$\{1\} \leftarrow (\delta_4 \wedge C4)$,
	$\{3\} \leftarrow \delta_5$,
	$\{4\} \leftarrow \delta_5$;
o	$\{0\} \leftarrow (\delta_3 \wedge C1)$,
	$\{1\} \leftarrow (\delta_3 \wedge C1)$,
	$\{2\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{3\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{4\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{5\} \leftarrow (\delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C3 \wedge C4)$,
	$\{7\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge \delta_5 \wedge C1 \wedge C2)$,
	$\{8\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge \delta_5 \wedge C1 \wedge C2)$,
	$\{9\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C2)$,
	$\{10\} \leftarrow (\delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta_4 \wedge C1 \wedge C2)$;

x_5. Explanations may not be as obvious. Concerning o, all value removals are implied by propagation (e.g., 0 is explained by $\delta_3 \wedge C1$), some of them are explained by a several decisions and constraints. For instance, the explanation of the removal of 2 from o is not trivial to recover. Retrospectively, the increase of the lower bound of o is related to the sum constraint ($C1$) and the lower bounds of the variables involved, more precisely the lower bounds of x_3, x_5 and x_6. The lower bounds of x_5 and x_6 depends on the application of δ_4 through $C4$ and $C1$. The lower bound of x_3 depends on the application of δ_2, $C3$ and δ_3.

Key components of an explanation system Adding explanations capabilities to a constraint solver requires addressing several aspects:

– computing explanations: domain reductions are usually associated with a *cause*: the propagator that actually performed the modification. This information can be used to compute an explanation. This can be done synchronously during propagation (by intrusive modification of the propagation algorithm) or asynchronously post propagation (by accessing an explanation service provided by propagators).
– storing explanations: a data structure needs to be defined to be able to store decisions made by the solver, domain reductions and their associated explanations. There exist several ways for storing explanations: a *flattened* storage of the domain modifications and their explanations composed of propagators and previously made decisions, or a *unflattened* storage of the domain modifications and their explanations expressed through previous domain modifications [6]. The data structure is referred to as *explanation store* in the following.
– accessing explanations: the data structure used to store explanations needs to provide access not only to domain modification explanations but also to current upper and lower bounds of the domains, current domain as a whole, etc.

In [13], the authors give an overview of techniques used to compute explanations and to handle them in a constraint solver. Despite being possibly very efficient, explanations suffer from several drawbacks:

– memory: storing explanations requires storing a way or another, variable modifications;
– cpu: computing explanations usually comes with a cost even though the propagation algorithm can be partially used for that;
– software engineering: implementing explanations can be quite intrusive within a constraint solver.

Finally, explanations were initially designed to deal with satisfaction problems. Generally, an optimization problem is processed as a sequence of satisfaction problems, in which cuts are added along with resolution to handle the optimization criterion. Regarding the explanation store, addition of cuts renders some explanations obsolete. Indeed, cuts cause modifications over domains that were previously achieved thanks to propagators.

3 Explanation-based LNS

In this section we introduce two new neighborhood computation techniques based on explanations for LNS. Those neighborhoods are referred to as `exp-cft` and `exp-obj`. Basically, we introduce implementations of the RELAX(S) method of Algorithm 1. For sake of simplicity, the following descriptions are stated in a minimization context; but, straightforward modifications adapt them to a maximization context.

3.1 Explaining the cut: `exp-cft`

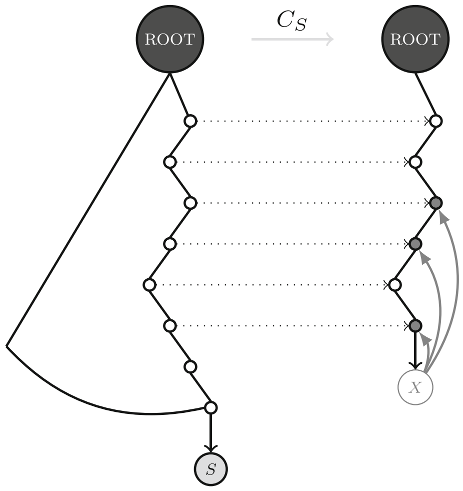
Historically, explanations have been used to explicate and repair a conflict. It is therefore only natural that we suggest a first neighborhood based on conflicts. But, instead of explaining each conflict occurring during the resolution process, we will force a conflict to be thrown when a solution is found. Indeed, we assume that there are far less solutions than failures and, thus, such a choice limits the overhead induces by plugging explanations in. A solution never leads to a conflict, though, so it needs to be prompted. When solving a COP, every new solution should be better than the previous one, until the optimum is reached. This is dynamically achieved by posting cuts. Given a solution S, P_S the decision path that leads to S and C_S the cut induces by S, it is not pertinent to entirely impose P_S together with the C_S in a standard resolution, because it necessarily leads to a useless and trivial conflict. However, provoking the generation of such a conflict with explanations plugged-in will enable to point out which decisions of P_S cannot be applied together with the cut. And then, the variables associated with these decisions may be helpful to compute partial solutions more able to be repaired. This serves as a basis for the first neighborhood, named `exp-cft`. Figure 1 depicts the main operations driven by `exp-cft`.

First, the conflict is provoked by *imposing* anew the decisions of P together with the cut. Once the conflict occurs, conflict-related decisions are retrieved from the explanation store. Second, neighborhoods are built on the basis of variables associated with the decisions explaining the cut. Since the application of the entire decision path led to build a solution e.g., an assignment of all its variables, a relaxation of the decision path will necessarily leave some variables uninstantiated, thereby building a partial solution. The complete method is described in Algorithm 2; The entry method is RELAX_EXP-CFT.

Each time a new solution is found (line 3), the method EXPLAINCUT (lines 4) is called. It returns the explanation of the conflict from which conflict-related decisions are extracted (line 5). Then, some of these decisions are randomly selected (line 7) and removed from the original decision path P. Finally, the relaxed path is imposed (line 8).

The EXPLAINCUT method (line 10-19) works as follow: first the decision path P of a solution S is retrieved (line 11), and the cut is posted (line 12). Then, all decisions of P are imposed and propagated one by one, with respect to the original (i.e., chronological) order (line 14-17). When the problem becomes unsatisfiable (line 17), the loop ends and the set

Fig. 1 Illustration of `exp-cft`



Algorithm 2 Cut-guided neighborhood (in a minimization context)

```
Require: $o$: the objective variable
Require: $k$: an integer
Require: $D_c$: set of decisions related to the conflict provoked by the cut
 1: procedure RELAX_EXP-CFT($S$)
 2:     $P$ ←PATHTO(S)                                   ▷ Retrieves the decision path to $S$
 3:     if a new solution has been found then
 4:         $\mathcal{E} \leftarrow$ EXPLAINCUT($S, o$)
 5:         $D_c$ ←EXTRACTDECISION($\mathcal{E}$)         ▷ Extracts decisions from explanations
 6:     end if
 7:     $R$ ←RANDOM($D_c$)                               ▷ Randomly selects decisions in $D_c$
 8:     APPLY($P \setminus R$)                           ▷ Applies $P$ minus $R$
 9: end procedure
10: procedure EXPLAINCUT($S, o$)
11:     $P \leftarrow$ PATHTO($S$)
12:     $Dom_o \setminus o^*$                            ▷ Posts the cut
13:     $\mathcal{F} \leftarrow \emptyset$               ▷ Propagation return status
14:     repeat
15:         $\delta$ ←POLLFIRST(P)
16:         $\mathcal{F} \leftarrow$ APPLY($\delta$)     ▷ $\delta$ is applied and propagated
17:     until $\mathcal{F} \neq \emptyset$               ▷ A conflict is thrown by propagation
18:     return EXPLAINCONFLICT($\mathcal{F}$)
19: end procedure
20: procedure EXPLAINCONFLICT($\mathcal{F}$)
21:     $\mathcal{E} \leftarrow \emptyset$               ▷ Explanation of the conflict
22:     if $\mathcal{F}$ is domain wipe out then
23:         for $x \in d$ do
24:             $\mathcal{E} \leftarrow \mathcal{E} \bigcup$EXPLAINREMOVAL($d, x$)   ▷ Explains the removal of $x$ from $d$.
25:         end for
26:     else if $\mathcal{F}$ is constraint inconsistency then
27:         $\mathcal{E}$ ←EXPLAINCONSTRAINT(c)          ▷ Constraint-specific method
28:     end if
29:     return $\mathcal{E}$
30: end procedure
```

of decisions related to the conflict is queried from the explanation store (line 18) by a call to the EXPLAINCONFLICT method.

The EXPLAINCONFLICT method (line 20-30) queries the explanation store and returns the set of decisions and constraints that explains the conflict. The type of conflict thrown conditions the way the explanation is built. If a domain wipe out occurs (line 22), the resulting explanation is the conjunction of the explanation of each value removal (a call to the EXPLAINREMOVAL method, line 24). If a constraint inconstency is detected (line 26), the EXPLAINCONSTRAINT method is called to retrieve an explanation (line 27). The default implementation calls the EXPLAINREMOVAL method for all values removed from the variables of the constraint; but constraint-specific implementations of the EXPLAINCONSTRAINT method provide more accurate explanations. At the end, both EXPLAINREMOVAL and EXPLAINCONSTRAINT query the explanation store .

Let D_c be the set of decisions related to the conflict provoked by the cut , there are $2^{|D_c|-1}$ subsets of D_c, each of them corresponding to a possible relaxation of P. Enumerating all the subsets of D_c is not polynomial, and there is no guarantee that small neighborhoods are more able to build better solutions than bigger ones. Hence, to enforce the diversification of `exp-cft` and to test neighborhoods of various sizes, we choose to randomly select α decisions to relax, where α is also randomly chosen in $[1, |D_c| - 1]$.

Example 2 On a call to EXPLAINCUT of the `exp-cft` neighborhood (Algorithm 2, Line 4), the first instruction is to post the cut (here, $C_{S_1} \equiv o < 6$), then P_{S_1} is imposed. The trace of the execution is reported in Table 3.

The application of the cut removes values greater than 5 from the domain of o (Line 1'). The application of δ_1 (Line 2'.a and Line 2'.b) and δ_2 (Line 3'.a and Line 3'.b) have the same effect with or without the cut. However, the application of δ_3 (step 4'.a), together with C_{S_1}, triggers more reductions and a conflict is detected by the propagator of $C1$ (Line 4'.f). The domains of x_i and o are such that it is impossible to find a valid assignment for o. Indeed, in [11], the authors established that if $\sum_{i=1}^{n} upp_{x_i} - low_o > 0$ then the constraint is unsatisfiable. In our case, $\sum_{i=1}^{n} upp_{x_i} - low_o = 2$, hence, the inconsistency is explained by the current upper bounds of x_i and the lower bound of o which, themselves, are explained by the decisions δ_1, δ_2 and δ_3. The conflict-related decisions are: $D_c = \{\delta_1, \delta_2, \delta_3\}$. The decisions δ_4 and δ_5 have not been applied before the conflict occurs. Thus, those decisions cannot be part of the conflict-related decisions set.

On a call to RELAX_EXP-CFT, δ_4 and δ_5 will be imposed by default, together with two or less randomly selected decisions from D_c.

The explanation of a conflict may not be unique nor minimal [13]. There may be no neighborhood of `exp-cft` that leads to a new solution. However, when the application of the cut directly throws a conflict, then D_c is empty and the resolution can be interrupted: the optimal solution has been found and proven (Algorithm 2, Line 2). This explains why such a method can be complete [12, 26].

Table 3 The trace of the search of the COP defined in Example 1 with the `exp-cft` neighborhood

Step	Cause	Consequences
1'	C_{S_1}	$o = [0, \mathbf{5}]$
2'.a	δ_1	$x_1 = [0, \mathbf{0}]$
2'.b	$x_1 \wedge C2$	$x_2 = [0, \mathbf{0}]$
3'.a	δ_2	$x_4 = [0, \mathbf{0}]$
3'.b	$x_4 \wedge C3$	$x_3 = [\mathbf{0}, 3]$
4'.a	δ_3	$x_3 = [\mathbf{2}, \mathbf{2}]$
4'.b	$x_3 \wedge C1$	$x_5 = [0, \mathbf{3}], x_6 = [0, \mathbf{3}], o = [\mathbf{2}, 5]$
4'.c	$x_5 \wedge x_6 \wedge C4$	$x_5 = [\mathbf{1}, 3], x_6 = [\mathbf{1}, 3]$
4'.d	$x_5 \wedge x_6 \wedge C1$	$x_5 = [1, \mathbf{2}], x_6 = [1, \mathbf{2}], o = [\mathbf{4}, 5]$
4'.e	$x_5 \wedge x_6 \wedge C4$	$x_5 = [\mathbf{2}, 2], x_6 = [\mathbf{2}, 2]$
4'.f	$x_5 \wedge x_6 \wedge C1$	inconsistency

Bold values highlight direct Consequences

3.2 Explaining the domain of the objective variable: `exp-obj`

The previous method defines neighborhoods based on conflicts, which is a common way to exploit explanations. We now present an alternative based on the *non optimal nature* of the best solution found so far. At a certain point of a COP resolution, one may wonder what prevents the solver from finding a better solution than a given one. The explanation store helps to determine which decisions prevent the objective variable from taking better values (in a minimization context, values strictly smaller than the current one, o^*). The variables involved in these decisions may be helpful to compute partial solutions more able to improve the best solution known so far. This serves as a basis for the second neighborhood, named `exp-obj`. Figure 2 depicts the main operations driven by `exp-obj`.

First, the explanation store is queried to retrieve decisions which are related to the removals of values below o^*. Second, neighborhoods are built on the basis of the variables associated with those decisions. Since, we aim at finding the optimal solution, it starts by relaxing decisions related to the smallest values first. The complete method is described in Algorithm 3; The entry point is the RELAX_EXP-OBJ method.

Each time a new solution is found (line 3), the method EXPLAINDOMAIN (line 4) is called. It returns the set of decisions related to the removal of values smaller than o^* from the objective variable. Then, some decisions are removed from the decision path P (lines 8-13), and the relaxed decision path is imposed (line 14). The way decisions are selected to be relaxed is conditioned by the number of relaxation tries. First of all, some decisions are selected to be relaxed according to the removal order (lines 9-10): those implying the removal of i from o are relaxed prior to the ones implying the removal of $i + 1$. When there is no more decisions to remove (line 11), the decisions are randomly selected (line 12). The RANDOM method is the same method as the one called in Algorithm 2 (line 6).

The method EXPLAINDOMAIN (lines 16-24) works as follows: first, the total number of removed values is computed (line 17). Then, an iteration over the values, from Low_o to $o^* - 1$, is achieved in order to explain each value removal (lines 19-23). Explanations of the value removals are queried from the explanation store (call to EXPLAINREMOVAL, line 20), and the decisions are extracted (line 21). Each value removal is explained by one or more decisions thanks to D_d and I (lines 19-22). D_d is an ordered set of decisions; On a value

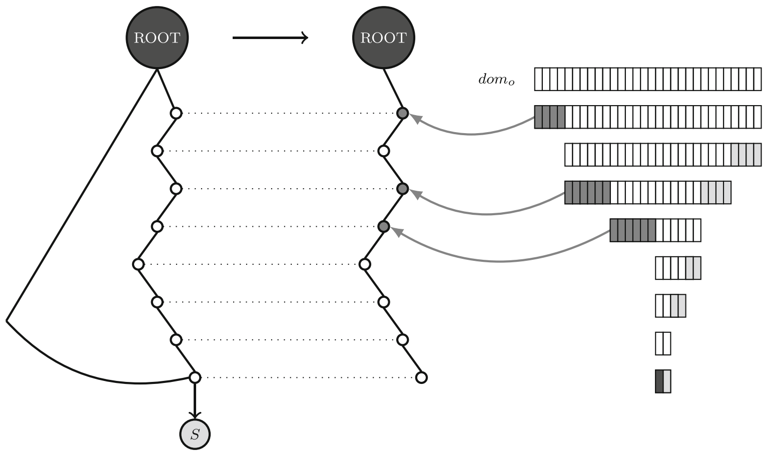


Fig. 2 Illustration of `exp-obj`

Algorithm 3 Domain-guided neighborhood (in a minimization context)

```
Require: $o$: the objective variable
Require: $k$: an integer
Require: $D_d$: ordered set of decisions related to the domain of $o$
Require: $I$: array of integers
 1: procedure RELAX_EXP-OBJ($S$)
 2:     $P \leftarrow$PATHTO(S)                         ▷ Retrieves the decision path to $S$
 3:     if a new solution has been found then
 4:         EXPLAINDOMAIN($S, o$)
 5:         $k \leftarrow 0$
 6:     end if
 7:     $k \leftarrow k + 1$
 8:     $R \leftarrow \emptyset$
 9:     if $k \leq length(I)$ then
10:         $R \leftarrow \bigcup_{j=1}^{I[k]} D_d[j]$
11:     else
12:         $R \leftarrow$RANDOM($D_d$)                  ▷ Randomly selects decisions in $D_d$
13:     end if
14:     APPLY($P \setminus R$)                          ▷ Applies $P$ minus $R$
15: end procedure
16: procedure EXPLAINDOMAIN($S, o$)
17:     $n \leftarrow (o^* - Low_o)$                    ▷ Gets the number of removed values
18:     $D_d \leftarrow []; I \leftarrow [];$
19:     for $k \in [1, n]$ do
20:         $\mathcal{E} \leftarrow$ EXPLAINREMOVAL($d_o, Low_o + k - 1$)
21:         $D_d \leftarrow D_d \bigcup$ EXTRACTDECISION($\mathcal{E}$)   ▷ Queries the explanation store
22:         $I[k] \leftarrow |D_d|$
23:     end for
24: end procedure
```

removal, decisions not already related to previous value removals are added into it. I is an array of indices; When the k^{th} value removed from o is explained, the size of D_d is stored in I. The relaxation operations (line 10) state that, as long as k is less than $length(I)$ (line 9), only decisions related to the removal of values less or equal to $Low_o + k$ are removed from the decision path. Such a guarantee is afforded by the way D_d and I are computed. In [15], it has been shown that, for a given variable, the explanations related to a value removal depends on, by construction, the explanations of previously removed values. In our context, this observation provides the following property on the objective variable.

Property 1 Given a solution S of cost $o^* \in [Low_o, Upp_o]$ and a decision path P associated with S. For all $x, x' \in [Low_o, o^*[$, $x \leq x'$, let $D_{\{x\}}$ be the set of decisions that explains the removal of x from o, we have:

$$D_{\{x\}} \subseteq D_{\{x'\}}$$

The *for-loop* depicted in Algorithm 3, line 19-23, relies on Property 1; It incrementally builds the set of decisions candidate for relaxation together with the array of indices, by iterating over removed values from Low_o to $o^* - 1$. Hence, when decisions related to the removals of the k first values of o have to be relaxed (line 10), one only has to remove the $I[k]$ first decisions of the ordered set D_d from P. Note that the novelty of the approach remains on the fact that explanation are accessed not on a conflict, as it is commonly done, but on a solution.

Example 3 In the Example 1, the objective variable o is instantiated to 6 in the solution. The explanations of each value removed from o during the resolution of the COP are reported in Table 2. On the one hand, values below o^* are explained by δ_2, δ_3 and δ_4. The removal of the values 0 and 1 from the domain of o are explained by the application of δ_3, through the propagation of $C1$ (Table 1, line 3.b). The removal of values from 2 to 5 is related to the execution of the constraint $C1$: the lower bound of the variables x_3, x_5 and x_6 enable to deduce that o cannot take a value lower than 6 (Table 1, line 4.c). The decisions δ_2 and δ_3 are related to the current lower bound of x_3; The decision δ_4 is related to the current lower bounds of x_5 and x_6.

On the other hand, removed values above o^* are explained by δ_4 and δ_5. The removal of the values 9 and 10 from the domain of o stems from the application of δ_4. The application of δ_5 triggers the removal of values 7 and 8, which reduce the domain of o to a singleton. As we are interested in improving the value of the best solution found so far, we only care about removal of values below $o^* = 6$. Hence, δ_5 will not considered as being part of D_d, and the execution of the EXPLAINDOMAIN method ends with $D_d = \{\delta_3, \delta_2, \delta_4\}$ and $I = \{1, 1, 3, 3, 3, 3\}$. Actually, I can be more compact, this is discussed after.

On a call to RELAX_EXP-OBJ, δ_1 and δ_5 have to be imposed, because they do not explained any removal of value below o^* from the domain of o. Then, the two first neighborhoods impose δ_3 and relax δ_2 and δ_4; the four following ones relax all decisions from D_d. Finally, any new call to RELAX_EXP-OBJ will randomly select decisions from D_d to be imposed with δ_1 and δ_5, until a new solution is found.

In the Example 3, δ_3 (respectively, $\delta_2 \wedge \delta_4$) manages to remove two (respectively, four) consecutive values from o. As a consequence, even though D_d is properly sized, 1 appears twice in I, and 3 appears four times, and some of the first neighborhoods of `exp-obj` will be redundant. By considering removals of consecutive values (*intervals*), instead of value removals, we can easily reduce the size of I and get a more efficient relaxation of the decisions path. In the previous example, the loop will consider the removal of two intervals, [0, 1] and [2, 5], instead of six value removals; The resulting decisions array and indices array will be $D_d = \{\delta_3, \delta_2, \delta_4\}$ and $I = \{1, 3\}$; The two first neighborhoods will be: $\{\delta_1, \delta_3; \delta_5\}$ and $\{\delta_1, \delta_5\}$, and then it switches to the random selection. Managing interval removals is used in our implementation.

These neighborhoods could be seen as an unnecessary complicated version of the *round-robin* process in which one or more top-level decisions would be questioned from the decision path. As shown in Example 3, not all (top-)decisions are related to (lower) value removals from the objective variable, some of them do not question the quality of the best solution found so far. Moreover, the choice of decisions to relax is not made aimlessly and is directed by the objective variable though explanations.

3.3 Additional information and further improvements

This section details the method to relax the decision path and techniques adopted to improve the efficiency of the approaches described in this paper.

Relaxing the decision path In this paragraph, we describe how the decision path is effectively updated (Algorithm 2, line 7 and Algorithm 3, line 14). The method APPLY($P \setminus R$) aims at removing a set of decisions R from a decision path P. First, all decisions from R are removed from P. Then, negated decisions must be considered too, even though they do not appear in R. Indeed, a decision δ is negated when the search process closes the sub-tree induced by δ, i.e., the entire sub-tree has been explored. A decision which becomes negated is explained by higher decisions in the search tree. So, if one decision which explains the negation of another one is removed, then keeping the negated decision is not justified anymore, and it should be removed too. For example, let $P = \{\delta_1, \delta_2, \delta_3, \neg\delta_4\}$ be a decision path and $R = \{\delta_2\}$ be the set of decisions to remove from P; The relaxed decision path P' is equal to $\{\delta_1, \delta_3\}$. $\neg\delta_4$ is automatically removed because it is explained by $\delta_1 \wedge \delta_2 \wedge \delta_3$. More details about the explanation of negated decisions are given in [13].

Lazy explanation recording Conflict-based searches access to explanations on each conflict [9, 12, 27]. Our approaches, on the contrary, only requires access on solutions. Consequently, it is not worth computing and storing explanations while solving. To avoid computing and storing useless information related to domain reduction, we implement a *lazy* and asynchronous fashion way to compute and store domain modifications, like described in [8]. Minimal data related to events generated during the resolution (i.e., the variable, the modification and the cause) is stored into a queue all resolution long. This queue is *backtrackable* to store relevant information and to reduce non relevant one upon backtracking. When a solution is found, the explanation store have to be queried, but is empty at that point. A *computation routine* is then executed: datas stored in the queue are popped one by one (w.r.t. the chronological order), the explanations are computed and stored. Once the queue is empty, the explanation store is up to date and ready for queries. Even though storing minimal data in the queue comes with a cost, it is negligible in comparison with maintaining the explanation store during the search and it significantly reduces both memory and cpu consumptions. Plugging lazy and asynchronous explanations in without querying the explanation store (nor computing explanations) only slows down the resolution process by less than 10 %.

Explaining interval removals Most of the time, domain reduction is treated as a sequence of value removals. For instance, a lower bound modification from i to $k-1$ is explained by the removal of all the values j from i to $k-1$. Such behavior becomes pathological when variables have large domains, which is often the case for the objective variable. Thus, it is mandatory to explain interval removals instead of value removals: it prevents from storing and computing a large amount of information and saves both and memory consumption. Our approach adapts the technique described in [15], originally proposed for numeric CSP, to integer domains represented as intervals. Note that Lazy Clause Generation solvers handle interval removals natively [32]. In LCG solvers, every value of the domain of an integer variable is represented using boolean variables $[\![v = x]\!]$ and $[\![v \leq x]\!]$. For instance, the variable $[\![v = x]\!]$ is true if v takes the value x and false if v takes a value different from x.

Dealing with enumerated domain objective variables Generally, the objective variable is *bounded*, that is, its domain is an interval of integers. An alternative is to define the domain as on ordered set of integers, the domain is said to be *enumerated*. Due to the size of the objective variable domain, which is generally very large, a bounded domain is often preferred to an enumerated one. In some cases, it is worth representing all values, though.

Such a choice does not call into question the validity of the Property 1 nor the behavior of `exp-obj`. But, it may build different neighborhoods.

Example 4 Given a COP with an objective variable o, its enumerated domain $dom_o = \{0, 1, 2, 3, 4\}$, a solution S with $o^* = 4$ and $P_S = (\delta_1, \delta_2, \delta_3, \delta_4)$, a decision path to S; Table 4 reports explanations for objective variable domain modifications.

An enumerated domain not only enables to update bounds but also to make holes (i.e., to remove a value in the middle of the domain). On S, the application of the method RELAX_EXP-OBJ will compute $D_d = \{\delta_4, \delta_2, \delta_3, \delta_1\}$, $I = \{2, 3, 4, 4, 4\}$. Then, the first partial solution will relax δ_4 and δ_2, which will restore the values 0 and 4 from dom_o. If this partial solution cannot lead to a new solution, a second partial solution will relax δ_4, δ_2 and δ_3, which restore the values 0, 1, 3 and 4 from dom_o.

When dealing with an objective variable with an enumerated domain, the execution of RELAX_EXP-OBJ may end with a *weaker* relaxed decision path, that is, values from o will be relaxed without necessarily respecting the lexical ordering of the domain.

Reconsidering the number of selected decisions In Section 3.1, we explained how the random selection of decisions to relax works: we select randomly α decisions to relax, where α is also randomly selected. However, the parameter α is not computed on each call to the RANDOM method, but every $\theta = minimum(\binom{|D|-1}{\alpha}, 200)$ calls, where D is either equal to D_c or D_d. This gives the opportunity to test a wide range of the possible combinations when $\binom{|D|-1}{\alpha}$ is small enough, and to test only a small subpart of them when the number is big. Various neighborhoods of the same size are tested before a new value for α is picked. We evaluate other approaches, and this one brings more robustness and improves the overall resolution process.

4 Evaluation

The central objective of the `exp-cft` and `exp-obj` algorithms are to build better neighborhoods in order to explore more appropriate parts of the search space and to speed up the LNS process. This section demonstrates the benefits of combining `exp-obj` and `exp-cft` together.

4.1 Implementation of LNS

Propagation-guided large neighborhood search Neither instance dependent but relying on global parameters, this combination of three neighborhoods has been proven to be very efficient on a modified version of the Car Sequencing Problem [23]. On each call to the

Table 4 Explanations of an enumerated domain objective variable

Variable	{removed value} ← Explanation
o	$\{0\} \leftarrow (\delta_4 \wedge \delta_2)$,
	$\{1\} \leftarrow \delta_3$,
	$\{2\} \leftarrow \delta_1$,
	$\{3\} \leftarrow (\delta_2 \wedge \delta_3)$,
	$\{4\} \leftarrow \delta_2$

propagation-guided neighborhood (or pgn), a first variable is selected randomly to be part of the partial solution and is assigned to its value in the previous solution. This assignment is propagated through the constraint network and the graph of dependencies is build: variables modified by propagation are marked. Every marked variable not instantiated is then stored in *priority list*, where variables are sorted by the domain reduction that occurred on their domain. The top priority variable in the list is selected to be fixed. The selection stops when the sum of the logarithm of the domain of all variables is below a given constant. The desired partial solution size is updated by adding a multiplicative correction factor epsilon. Two alternatives have been defined: The *reverse propagation-guided neighborhood* (or repgn) is built by expansion instead of by reduction; The *random propagation-guided neighborhood* (or rapgn) is implemented with a list of size 0. Hence, the first contender is made of a sequential application of pgn, repgn and rapgn. We use the default parameters defined in [23]: the size of the list is set to 10, the constant is valued to 30 and the way epsilon evolves is dynamic. As we consider LNS as a black-box strategy here, we do not try to adapt the parameters to the problem treated and we focus on the genericity of the approach. This contender is called *PGLNS*.

Explanation-based LNS We evaluate various combinations of the explanation-based neighborhoods presented in Section 3. The first combination is made of exp-obj and a random neighborhood, named ran. The latter will have to bring diversification by providing neighborhoods which are not related to the problem structure. Indeed, in [23], the authors "*obtained better results by interleaving purely random neighborhoods with more advanced ones*". Thus, ran relaxes ζ variables randomly selected on each call to the RELAX method; the remaining ones are obviously instantiated to their value in the best solution known so far. ζ is set to $\frac{|\mathcal{V}|}{3}$ on a solution; and is incremented once every 200 calls to the neighborhood computation. Such parameters enable a strong diversification. The first contender is named *objLNS*. A second combination, named *cftLNS*, groups together exp-cft and ran. As exp-obj and exp-cft exploit explanations in two different ways, we suggest a third combination, named EBLNS, made of exp-obj, exp-cft and ran. We hope EBLNS will improve the overall behavior of each neighborhood used individually.

In every contenders, each neighborhood is then applied fairly, in sequence, until a new solution is found.

Fast restarts All contenders are evaluated with a fast restart strategy [22] plugged in: we limit the reparation step to 30 fails. Such a strategy is commonly associated with LNS and has been proven to improve its efficiency.

Random LNS A purely random neighborhood, made exclusively of ran, easy to implement and configuration-free has been evaluated too. Due to its poor efficiency in practice (it was never competitive with any other approaches evaluated here), the results are not reported here, though.[2]

4.2 Benchmark protocol

Propagation-Guided LNS and Explanation-based LNS were implemented in Choco-3.1.0 [28], a Java library for constraint programming. All the experiments were done on a

[2]The complete results of purely random contender are available on request.

Macbook Pro with a 6–core Intel Xeon at 2.93Ghz running on MacOS 10.6.8, and Java 1.7. Each run had one core and a 15 minutes time limit. Due to randomness, the evaluations were run ten times and the arithmetic mean of the objective value (obj), its range[3] (rng) and the standard deviation (stddev) are reported. The range gives indication about the stability of an approach. Note that the first solution of each run of the same problem is always the same one, whatever versions of LNS is plugged in.

4.3 Benchmark description

The evaluation proposed here is based on ten problems composed of 49 instances. There are nine optimization problems extracted from the MiniZinc distribution and one additional problem, the Optimized Car Sequencing Problem.[4] The latter one has been added to facilitate the comparison with PGLNS.

We kept instances for which classic backtrack algorithm finds at least one solution within a 15 minutes time limit: LNS needs an initial solution to be activated.

There are five minimization problems: the Modified Car Sequencing Problem (`car_cars`), the Restaurant Assignment Problem (`fastfood`), the League Model Problem (`league_model`), the Resource-Constrained Project Scheduling Problem (`rcpsp`) and the Vehicle Routing Problem (`vrp`). There are five maximization problems: the Maximum Profit Subpath Problem (`mario`), the Itemset Mining Problem (`pattern_set_mining`), the Prize Collecting Problem (`pc`), the Ship Scheduling Problem (`ship_schedule`) and the Still Life Problem (`still_life`). Thereafter, the results will be presented by type. Obviously, there is no fundamental differences between the two classes of problems, and we did not except the behavior of LNS to be different. The details of the models are reported in Table 5.

As global constraints require implementation of specific explanation schemas, whose evaluation is not the purpose of the paper, the problems are modeled with built-in constraints, which are natively explained. The following problems were initially modeled with global constraints: the Modified Car Sequencing Problem, the League Model Problem, the Resource-Constrained Project Scheduling Problem, the Maximum Profit Subpath Problem and the Itemset Mining Problem. Moreover, in half of the classes of problem, the search strategies are static: `input_order`.

4.4 Evaluation of objLNS, cftLNS and EBLNS

The main motivation of this paper is twofold. First, we suggest two generic neighborhoods based on explanations for the LNS framework. They require neither accurate parameterization nor need to be adapted to the instance treated. Second, we show that they define neighborhoods more able to build new solutions, and thus improve the resolution of optimization problems. In this section, we compare various contenders based on explanations, objLNS, cftLNS and EBLNS, with the propagation-guided one, PGLNS. Various pairwise comparisons between contenders are done. The results are presented in tables which report the arithmetic mean of the objective variable and its range of the approaches evaluated.

[3]The *range*: $100 * \frac{highest - lowest}{mean}$.

[4]The Optimized Car Sequencing Problem (5 instances) is a modified version of the Car Sequencing Problem in which an additional option-free configuration has been added (as described in [23]). The objective is to schedule the cars requiring this configuration at the end, in such a way that a solution to the original satisfaction problem is found.

Table 5 Descriptions of the problems treated

Problem	Constraints	Search
car_cars	array_int_element, bool2int, int_eq, int_eq_reif, int_lin_eq, int_lin_le	{input_order, indomain_max}
fastfood	int_abs, int_lin_eq, int_lt, int_min, set_in	{input_order, indomain_min}
league_model	array_bool_or, bool2int, bool_le, int_eq_reif, int_le,	{first_fail, indomain_max} ∧ {first_fail, indomain_min}
rcpsp	array_bool_and, bool2int, bool_eq_reif, bool_le, int_le_reif, int_lin_le, int_lin_le_reif indomain_min}	{smallest, indomain_min} ∧ {input_order,
vrp	int_le, int_lin_eq, int_lin_le indomain_min}	{first_fail,
mario	array_bool_and, array_bool_or, array_int_element, array_var_bool_element, array_var_int_element, bool2int, bool_eq_reif, bool_le, bool_le_reif, int_eq, int_eq_reif, int_lin_eq, int_lt_reif, int_min, int_ne_reif	{first_fail, indomain_min} ∧ {input_order, indomain_max}
pattern_set_mining	bool2int, bool_eq, int_lin_eq, int_lin_le_reif	{input_order, indomain_max}
pc	array_bool_and, array_int_element, array_var_int_element, bool2int, bool_le, int_eq, int_eq_reif, int_lin_eq, int_lin_le, int_lt_reif {input_order, indomain_max}	{largest, indomain_max} ∧ {largest, indomain_max} ∧
ship_schedule	array_bool_and, array_bool_or, array_int_element, bool2int, bool_le, int_eq, int_eq_reif, int_le_reif, int_lin_eq, int_lin_le, int_lin_le_reif, int_lt_reif, int_times	{input_order, indomain_max} ∧ {input_order, indomain_min}
still_life	array_bool_and, bool_le, int_eq, int_eq_reif, int_le_reif, int_lin_eq, int_lin_le, int_lin_le_reif, int_lin_ne_reif, int_times	{input_order, indomain_max}

Bold number highlights the best objective value per instances; Italic numbers denote equality. We also use plots to display the multiplying factor over the objective value obtained by using one approach instead of the other. The horizontal axis represents the instances treated, sorted with respect of the difference between the solution found using the first approach h_1 and the one using the second one h_2, in increasing order. The vertical axis reports the multiplying factors, $\rho(h_i, h_j) = max(\frac{h_i}{h_j}, 1)$. A mark on the left side of the plot (light gray area) reports $\rho(h_1, h_2)$ for an instance better solved with h_1, it measures the loss of using h_2 instead of h_1. A mark on the right side of the plot (dark gray area) reports $\rho(h_2, h_1)$ for an instance better solved with h_2, it measures the gain of using h_2 instead of h_1. The plots also report the approximated area A[5] of the gain and loss. The larger the dark gray (respectively light gray) area is, the bigger the improvement (respectively, the loss) related to EBLNS.

4.4.1 Comparative evaluation of PGLNS and objLNS

Table 6 reports the results obtained with PGLNS and objLNS on the ten problems.

On 19 out of 49 instances, PGLNS is the best approach, whereas, on 26 instances objLNS is the more efficient. In about one-third of the cases (6 out of 9 for PGLNS and 8 out of 26 for objLNS) the range is almost 0. That means all resolutions treated by the same contender lead to in the same last solution. Besides, there are four instances where the two approaches are equivalent.

On the one hand, objLNS finds the best solutions for the Restaurant Assignment problem (`fastfood`), the League Model problem (`league`), the Prize Collecting problem (`pc`) and the Still Life problem (`still_life`). objLNS is more stable than PGLNS on instances of the Restaurant Assignment problem and the Prize Collecting problem. That is not true for the other instances. This contender seems also appropriate to solve the Vehicle Routing problem (`vrp`), but the two approaches provide unstable results on these instances. One can suppose that random neighborhoods play an important role in the resolution of those instances. On the other hand, PGLNS is the best approach to solve the Resouce-Constrained Project Scheduling problem (`rcpsp`), it also more stable than objLNS on these instances. This contender is also appropriate to treat the Modified Car Sequencing problem (`car_cars`), the Itemset Mining problem (`pattern_set_mining`) and the Ship Scheduling problem (`ship_schedule`), where it is also more stable. More generally, PGLNS is more stable (16.4 %), in average, than objLNS (24.54 %).

Every approach seems more adapted to solve some kinds of problems, but objLNS appears to be the one with the broadest range of resolutions. Besides, the contribution of one approach over the other is quite hard to evaluate exactly just by reading the table. That is why we also plot the results, in Fig. 3.

On minimization problems (Fig. 3a), objLNS is more interesting: the dark gray area ($A \approx 8.3743$) is larger than the light gray one ($A \approx 0.3451$). This means that objLNS enables to find better solutions than PGLNS, up to 6.34 times. Such a gap is due to the instances of the Restaurant Assignment problem (`fastfood`) where objLNS find far superior results.

On maximization problems (Fig. 3b), PGLNS is more interesting. The light gray area ($A \approx 2.1993$) is larger than the dark gray one ($A \approx 0.8617$). The gap is less important than on minimization problems, and comes from the results in favor of PGLNS

[5]The approximation is computed with the Trapezoidal rule [1].

Table 6 Comparative evaluation of PGLNS and objLNS

Instance	PGLNS			objLNS		
	obj	rng(%)	stddev	obj	rng(%)	stddev
Minimization Problems						
cars_cars_4_72.fzn	**109.3**	5.49	1.85	116.9	4.28	1.51
cars_cars_16_81.fzn	121.2	2.48	1.17	**119.5**	6.69	3.41
cars_cars_26_82.fzn	**127.5**	5.49	2.20	136.6	6.59	2.29
cars_cars_41_66.fzn	**108.8**	1.84	0.98	109.8	5.46	1.40
cars_cars_90_05.fzn	**300.8**	11.97	9.93	319.3	2.82	3.41
fastfood_ff3.fzn	3883.4	90.64	967.10	**1331**	0.45	2.05
fastfood_ff58.fzn	8882.7	101.78	2955.56	**1399.2**	10.58	46.12
fastfood_ff59.fzn	871	0.00	0.00	**294.6**	32.59	25.00
fastfood_ff61.fzn	221.7	24.81	16.41	**189.6**	29.54	16.50
fastfood_ff63.fzn	146.9	40.16	17.70	**120.1**	47.46	16.69
league_model55-3-12.fzn	139949	35.73	18439.09	**109949.7**	0.00	0.46
league_model90-18-20.fzn	1922916	7.28	35228.00	**1117921**	39.36	137823.82
league_model100-21-12.fzn	3139911.9	0.00	2.17	**2486918.6**	85.65	818388.45
rcpsp_11.fzn	**81**	3.70	1.10	83.6	2.39	0.66
rcpsp_12.fzn	**38.3**	2.61	0.46	39.9	5.01	0.54
rcpsp_13.fzn	**78**	0.00	0.00	79	0.00	0.00
rcpsp_14.fzn	**140.9**	0.71	0.30	141	0.00	0.00
vrp_A-n38-k5.vrp.fzn	2341.6	22.16	155.59	**2322.2**	18.30	116.17
vrp_A-n62-k8.vrp.fzn	6318.1	6.66	127.92	**6104**	29.26	454.38
vrp_B-n51-k7.vrp.fzn	**4291.8**	15.87	189.12	4355.8	19.74	246.86
vrp_P-n20-k2.vrp.fzn	402.1	41.28	43.70	**358.9**	21.73	22.50
vrp_P-n60-k15.vrp.fzn	**2271.9**	12.28	91.15	2424	20.30	135.45

Table 6 (continued)

Instance	PGLNS			objLNS		
	obj	rng(%)	stddev	obj	rng(%)	stddev
Maximization Problems						
`mario_mario_easy_2.fzn`	*628*	0.00	0.00	*628*	0.00	0.00
`mario_mario_easy_4.fzn`	506	8.70	13.05	**517.6**	8.50	19.05
`mario_mario_n_medium_2.fzn`	719.9	31.95	61.85	**774.7**	22.85	59.38
`mario_mario_n_medium_4.fzn`	**576.6**	52.90	110.30	555.1	37.29	72.76
`mario_mario_t_hard_1.fzn`	**4783**	0.00	0.00	3199.9	148.00	1947.09
`pattern_set_mining_k1_ionosphere.fzn`	**16.9**	130.18	7.12	11	100.00	3.46
`pattern_set_mining_k2_audiology.fzn`	**53.6**	1.87	0.49	52.7	18.98	3.03
`pattern_set_mining_k2_german-credit.fzn`	3	0.00	0.00	**3.8**	131.58	1.54
`pattern_set_mining_k2_segment.fzn`	**11.8**	8.47	0.40	11	0.00	0.00
`pc_25-5-5-9.fzn`	62.8	3.18	0.98	**64**	0.00	0.00
`pc_28-4-7-4.fzn`	47.7	12.58	2.72	**58**	0.00	0.00
`pc_30-5-6-7.fzn`	60	0.00	0.00	**60.9**	6.57	1.14
`pc_32-4-8-0.fzn`	104.7	14.33	4.38	**109.7**	8.20	3.41
`pc_32-4-8-2.fzn`	84.9	15.31	6.14	**89.8**	6.68	2.75
`ship-schedule.cp_5Ships.fzn`	483645.5	0.01	13.50	**452157**	19.40	37368.74
`ship-schedule.cp_6ShipsMixed.fzn`	**300185**	4.86	4378.36	248822.5	43.45	41741.68
`ship-schedule.cp_7ShipsMixed.fzn`	**396137**	20.40	23645.31	279396	89.34	84364.24
`ship-schedule.cp_7ShipsMixedUnconst.fzn`	**364141.5**	21.01	26324.16	285652.5	87.13	80181.40
`ship-schedule.cp_8ShipsUnconst.fzn`	**782516**	20.51	53704.49	584308.5	48.27	89265.47
`still-life_09.fzn`	37.6	13.30	1.50	**42**	0.00	0.00
`still-life_10.fzn`	49.8	4.02	0.87	**51.9**	5.78	0.94
`still-life_11.fzn`	59	0.00	0.00	**61.3**	1.63	0.46
`still-life_12.fzn`	63.3	4.74	0.90	**66.5**	21.05	4.57
`still-life_13.fzn`	79.7	2.51	0.78	**81.7**	9.79	2.79

Bold numbers highlight the best objective value per instances. Italic numbers denote equality

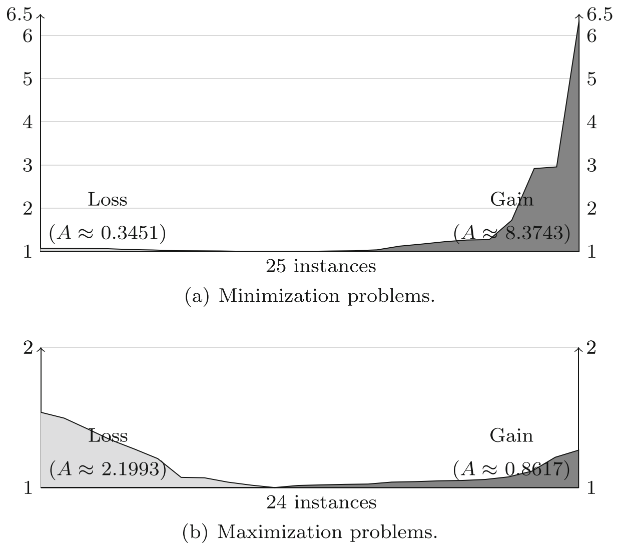


Fig. 3 Multiplying factor between PGLNS and objLNS, per instance

on the Ship Scheduling problem (`ship_schedule`) and the Itemset Mining problem (`pattern_set_mining`).

In conclusion, objLNS is more attractive in average, its contribution to good quality solutions is valuable concerning minimization problems. On maximization problems, the gain does not seem to be interesting, but objLNS better solves more instances than PGLNS (15 out of 24).

4.4.2 Comparative evaluation of PGLNS and cftLNS

Table 7 reports results obtained with PGLNS and cftLNS on the 49 instances.

In comparison with PGLNS, cftLNS better treats 29 out of 49 instances, whereas PGLNS is the most efficient on 15 cases. There are 6 equalities. In about one-third of the cases (5 out of 15 for PGLNS, 8 out of 29 for cftLNS), the range is equal to 0, which indicates a good stability of the approaches.

The distribution is very similar to the one observed with objLNS: cftLNS is suitable for the Restaurant Assignment problem (`fastfood`), the League Model problem (`league`), the Prize Collecting problem (`pc`) and the Still Life problem (`still_life`). cftLNS does not bring more stability in comparison with objLNS, though. Besides, cftLNS seems to be appropriate to deal with the Vehicle Routing problem (`vrp`), the Maximum Profit Subpath (`mario`) and the Itemset Mining problem (`pattern_set_mining`) with respect to PGLNS. But, it is less obvious to conclude regarding its stability. We observe larger gaps, in particular on instances of the Itemset Mining problem (`pattern_set_mining`). PGLNS remains the best approach to solve efficiently instances of the Ship Scheduling problem (`ship_schedule`) and the Modified Car Sequencing problem (`car_cars`). On this last problem, one may see that one instance is better solved with cftLNS. In general, PGLNS is a little more stable than cftLNS (16.4 % for PGLNS, 17.94 % for cftLNS).

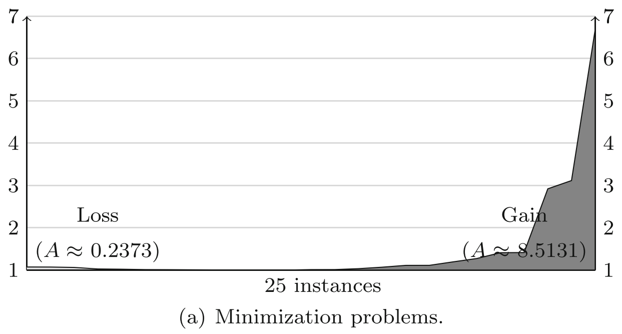
Table 7 Comparative evaluation of PGLNS and cftLNS

	PGLNS			cftLNS		
Instance	obj	rng(%)	stddev	obj	rng(%)	stddev
Minimization Problems						
`cars_cars_4_72.fzn`	**109.3**	5.49	1.85	117.1	4.27	1.51
`cars_cars_16_81.fzn`	121.2	2.48	1.17	**119.9**	6.67	3.30
`cars_cars_26_82.fzn`	**127.5**	5.49	2.20	136.8	5.85	2.64
`cars_cars_41_66.fzn`	**108.8**	1.84	0.98	109.8	5.46	1.40
`cars_cars_90_05.fzn`	**300.8**	11.97	9.93	319.5	2.50	3.01
`fastfood_ff3.fzn`	3883.4	90.64	967.10	**1330**	0.00	0.00
`fastfood_ff58.fzn`	8882.7	101.78	2955.56	**1329.2**	16.48	87.60
`fastfood_ff59.fzn`	871	0.00	0.00	**279.6**	16.81	18.80
`fastfood_ff61.fzn`	221.7	24.81	16.41	**157.2**	18.45	10.49
`fastfood_ff63.fzn`	146.9	40.16	17.70	**103.9**	0.96	0.30
`league_model20-3-5.fzn`	*49984*	0.00	0.00	*49984*	0.00	0.00
`league_model30-4-6.fzn`	*79973*	0.00	0.00	*79973*	0.00	0.00
`league_model50-4-4.fzn`	*99971*	0.00	0.00	*99971*	0.00	0.00
`league_model55-3-12.fzn`	139949	35.73	18439.09	**109949.9**	0.00	0.30
`league_model90-18-20.fzn`	1922916	7.28	35228.00	**1730925.5**	26.58	132319.46
`league_model100-21-12.fzn`	3139911.9	0.00	2.17	**3103922.1**	2.58	23747.70
`rcpsp_11.fzn`	**81**	3.70	1.10	81.9	1.22	0.30
`rcpsp_12.fzn`	**38.3**	2.61	0.46	39.3	5.09	0.64
`rcpsp_13.fzn`	*78*	0.00	0.00	*78*	0.00	0.00
`rcpsp_14.fzn`	**140.9**	0.71	0.30	141	0.00	0.00
`vrp_A-n38-k5.vrp.fzn`	2341.6	22.16	155.59	**2106.2**	44.44	261.66
`vrp_A-n62-k8.vrp.fzn`	6318.1	6.66	127.92	**6117.2**	16.30	310.67
`vrp_B-n51-k7.vrp.fzn`	4291.8	15.87	189.12	**4019.1**	29.83	353.93
`vrp_P-n20-k2.vrp.fzn`	402.1	41.28	43.70	**336.8**	23.46	26.14
`vrp_P-n60-k15.vrp.fzn`	**2271.9**	12.28	91.15	2317.8	29.99	224.26

Table 7 (continued)

Instance	*PGLNS*			*cftLNS*		
	obj	rng(%)	stddev	obj	rng(%)	stddev
Maximization Problems						
`mario_mario_easy_2.fzn`	*628*	0.00	0.00	*628*	0.00	0.00
`mario_mario_easy_4.fzn`	506	8.70	13.05	**510.1**	8.63	17.47
`mario_mario_n_medium_2.fzn`	719.9	31.95	61.85	**889**	25.20	63.14
`mario_mario_n_medium_4.fzn`	576.6	52.90	110.30	**723.9**	28.60	74.47
`mario_mario_t_hard_1.fzn`	**4783**	0.00	0.00	3039.4	157.37	1679.16
`pattern_set_mining_k1_ionosphere.fzn`	16.9	130.18	7.12	**26.6**	150.38	15.94
`pattern_set_mining_k2_audiology.fzn`	53.6	1.87	0.49	**54**	0.00	0.00
`pattern_set_mining_k2_german-credit.fzn`	3	0.00	0.00	**4.9**	81.63	1.30
`pattern_set_mining_k2_segment.fzn`	**11.8**	8.47	0.40	11	0.00	0.00
`pc_25-5-5-9.fzn`	62.8	3.18	0.98	**64.2**	1.56	0.40
`pc_28-4-7-4.fzn`	47.7	12.58	2.72	**57.2**	13.99	2.40
`pc_30-5-6-7.fzn`	60	0.00	0.00	**62.7**	6.38	1.62
`pc_32-4-8-0.fzn`	104.7	14.33	4.38	**111.2**	8.09	3.60
`pc_32-4-8-2.fzn`	84.9	15.31	6.14	**91.6**	4.37	1.20
`ship-schedule.cp_5Ships.fzn`	**483645.5**	0.01	13.50	483602.5	0.02	47.50
`ship-schedule.cp_6ShipsMixed.fzn`	**300185**	4.86	4378.36	251359.5	68.71	63750.63
`ship-schedule.cp_7ShipsMixed.fzn`	**396137**	20.40	23645.31	302325.5	10.34	10123.47
`ship-schedule.cp_7ShipsMixedUnconst.fzn`	**364141.5**	21.01	26324.16	318492.5	23.42	27619.44
`ship-schedule.cp_8ShipsUnconst.fzn`	**782516**	20.51	53704.49	470266.5	22.39	31509.21
`still-life_09.fzn`	37.6	13.30	1.50	**42**	0.00	0.00
`still-life_10.fzn`	49.8	4.02	0.87	**53.6**	3.73	0.66
`still-life_11.fzn`	59	0.00	0.00	**61.2**	1.63	0.40
`still-life_12.fzn`	63.3	4.74	0.90	**75.1**	2.66	0.54
`still-life_13.fzn`	79.7	2.51	0.78	**87.2**	3.44	0.87

Bold numbers highlight the best objective value per instances. Italic numbers denote equality



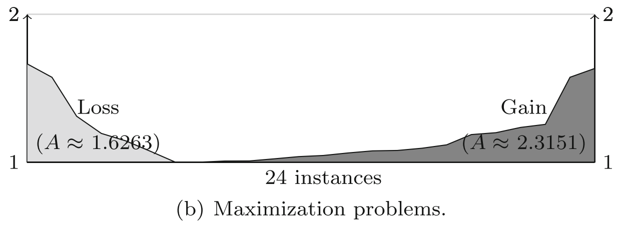


Fig. 4 Multiplying factor between PGLNS and cftLNS, per instance

Figure 4 reports the loss and gain of using cftLNS instead of PGLNS.On minimization problems (Fig. 4a), the plot is comparable with one about objLNS: the dark gray area ($A \approx 8.5131$) is larger than the light gray one ($A \approx 0.2373$). cftLNS brings few loss and significant gain on those instances. Once again, the instances of the Restaurant Assignment problem (`fastfood`) are much better treated with cftLNS (corresponding to left peak). On maximization problems (Fig. 4b), the trend continues: cftLNS seems to be more efficient. One may note that not only the dark gray area ($A \approx 2.31.51$) is larger than the light gray one ($A \approx 1.6263$), which means that cftLNS builds better quality solutions in average, but also that it betters more instances (16 out 24 for cftLNS, 7 for PGLNS).

In conclusion, cftLNS appears to be a good alternative to PGLNS, not only in term of the number of instances better solved, but also in term of gain regarding the neighborhoods guided by propagation.

4.4.3 Comparative evaluation of objLNS and cftLNS

Results of cftLNS and objLNS seem to be very similar, in comparison with PGLNS. Now, we compare these two approaches and report the results in the Table 8. The cftLNS approach dominates: it betters 28 out of 49 instances, whereas objLNS finds better solution in 13 cases. There are eight equalities. In addition, cftLNS is more stable in average (17.94 % for cftLNS, 24.54 % for objLNS). objLNS is the best approach to solve instances of the Modified Car Sequencing problem (`car_cars`) and the League Model problem (`league_model`), though. Regarding cftLNS, it is more suitable to solve instances of the Restaurant Assignment problem (`fastfood`), the Resource-Constrained Project Scheduling problem (`rcpsp`), and, even less so, the Vehicle Routing problem (`vrp`), the Itemset Mining (`pattern_set_mining`) and the Still Life problem (`still_life`). Only a few instances of the Maximum Profit Subpath problem (`mario`) are hard to decide between the two contenders.

Table 8 Comparative evaluation of objLNS and cftLNS

	objLNS			cftLNS		
Instance	obj	rng(%)	stddev	obj	rng(%)	stddev
Problèmes de minimisation						
`cars_cars_4_72.fzn`	**116.9**	4.28	1.51	117.1	4.27	1.51
`cars_cars_16_81.fzn`	**119.5**	6.69	3.41	119.9	6.67	3.30
`cars_cars_26_82.fzn`	**136.6**	6.59	2.29	136.8	5.85	2.64
`cars_cars_41_66.fzn`	*109.8*	5.46	1.40	*109.8*	5.46	1.40
`cars_cars_90_05.fzn`	**319.3**	2.82	3.41	319.5	2.50	3.01
`fastfood_ff3.fzn`	1331	0.45	2.05	**1330**	0.00	0.00
`fastfood_ff58.fzn`	1399.2	10.58	46.12	**1329.2**	16.48	87.60
`fastfood_ff59.fzn`	294.6	32.59	25.00	**279.6**	16.81	18.80
`fastfood_ff61.fzn`	189.6	29.54	16.50	**157.2**	18.45	10.49
`fastfood_ff63.fzn`	120.1	47.46	16.69	**103.9**	0.96	0.30
`league_model20-3-5.fzn`	*49984*	0.00	0.00	*49984*	0.00	0.00
`league_model30-4-6.fzn`	*79973*	0.00	0.00	*79973*	0.00	0.00
`league_model50-4-4.fzn`	*99971*	0.00	0.00	*99971*	0.00	0.00
`league_model55-3-12.fzn`	**109949.7**	0.00	0.46	109949.9	0.00	0.30
`league_model90-18-20.fzn`	**1117921**	39.36	137823.82	1730925.5	26.58	132319.46
`league_model100-21-12.fzn`	**2486918.6**	85.65	818388.45	3103922.1	2.58	23747.70
`rcpsp_11.fzn`	83.6	2.39	0.66	**81.9**	1.22	0.30
`rcpsp_12.fzn`	39.9	5.01	0.54	**39.3**	5.09	0.64
`rcpsp_13.fzn`	79	0.00	0.00	**78**	0.00	0.00
`rcpsp_14.fzn`	*141*	0.00	0.00	*141*	0.00	0.00
`vrp_A-n38-k5.vrp.fzn`	2322.2	18.30	116.17	**2106.2**	44.44	261.66
`vrp_A-n62-k8.vrp.fzn`	**6104**	29.26	454.38	6117.2	16.30	310.67
`vrp_B-n51-k7.vrp.fzn`	4355.8	19.74	246.86	**4019.1**	29.83	353.93
`vrp_P-n20-k2.vrp.fzn`	358.9	21.73	22.50	**336.8**	23.46	26.14
`vrp_P-n60-k15.vrp.fzn`	2424	20.30	135.45	**2317.8**	29.99	224.26

Table 8 (continued)

	objLNS			cftLNS		
Instance	obj	rng(%)	stddev	obj	rng(%)	stddev
Problèmes de maximisation						
`mario_mario_easy_2.fzn`	*628*	0.00	0.00	*628*	0.00	0.00
`mario_mario_easy_4.fzn`	**517.6**	8.50	19.05	510.1	8.63	17.47
`mario_mario_n_medium_2.fzn`	774.7	22.85	59.38	**889**	25.20	63.14
`mario_mario_n_medium_4.fzn`	555.1	37.29	72.76	**723.9**	28.60	74.47
`mario_mario_t_hard_1.fzn`	**3199.9**	148.00	1947.09	3039.4	157.37	1679.16
`pattern_set_mining_k1_ionosphere.fzn`	11	100.00	3.46	**26.6**	150.38	15.94
`pattern_set_mining_k2_audiology.fzn`	52.7	18.98	3.03	**54**	0.00	0.00
`pattern_set_mining_k2_german-credit.fzn`	3.8	131.58	1.54	**4.9**	81.63	1.30
`pattern_set_mining_k2_segment.fzn`	*11*	0.00	0.00	*11*	0.00	0.00
`pc_25-5-5-9.fzn`	64	0.00	0.00	**64.2**	1.56	0.40
`pc_28-4-7-4.fzn`	58	0.00	0.00	**57.2**	13.99	2.40
`pc_30-5-6-7.fzn`	60.9	6.57	1.14	**62.7**	6.38	1.62
`pc_32-4-8-0.fzn`	109.7	8.20	3.41	**111.2**	8.09	3.60
`pc_32-4-8-2.fzn`	89.8	6.68	2.75	**91.6**	4.37	1.20
`ship-schedule.cp_5Ships.fzn`	452157	19.40	37368.74	**483602.5**	0.02	47.50
`ship-schedule.cp_6ShipsMixed.fzn`	248822.5	43.45	41741.68	**251359.5**	68.71	63750.63
`ship-schedule.cp_7ShipsMixed.fzn`	279396	89.34	84364.24	**302325.5**	10.34	10123.47
`ship-schedule.cp_7ShipsMixedUnconst.fzn`	285652.5	87.13	80181.40	**318492.5**	23.42	27619.44
`ship-schedule.cp_8ShipsUnconst.fzn`	**584308.5**	48.27	89265.47	470266.5	22.39	31509.21
`still-life_09.fzn`	*42*	0.00	0.00	*42*	0.00	0.00
`still-life_10.fzn`	51.9	5.78	0.94	**53.6**	3.73	0.66
`still-life_11.fzn`	**61.3**	1.63	0.46	61.2	1.63	0.40
`still-life_12.fzn`	66.5	21.05	4.57	**75.1**	2.66	0.54
`still-life_13.fzn`	81.7	9.79	2.79	**87.2**	3.44	0.87

Bold numbers highlight the best objective value per instances. Italic numbers denote equality

Nevertheless, this must be balanced with the gain cftLNS brings. Figure 5 reports this information.

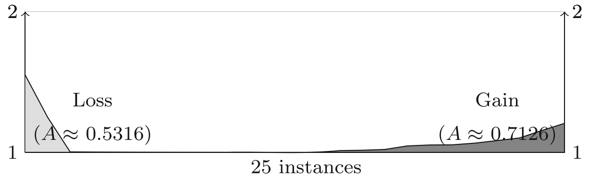
On minimization problems (Fig. 5a), the gain and loss are equivalent, both approaches are comparable. The light gray area ($A \approx 0.5316$) is smaller than the dark gray one ($A \approx 0.7126$), but the latter is wider than it is tall. This confirms that cftLNS better solves more instances than objLNS but that the gain is low. By contrast, objLNS brings more gain but on fewer instances. On maximization problems (Fig. 5b), the results are clearly in favor of cftLNS.

4.4.4 Comparative evaluation of EBLNS and PGLNS

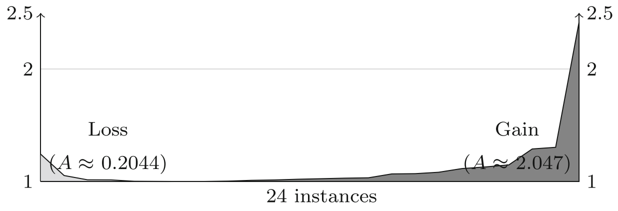
One strength of LNS is the capacity it offers to combine various neighborhoods together. We now evaluate PGLNS and EBLNS, a contender which combines the two explanation-based neighborhoods together with a purely random one (Fig. 6). The results are reported in Table 9.

The results are well distributed among PGLNS and EBLNS: in 22 out of 49 examples, PGLNS found the best solutions, whereas EBLNS found the best solutions on 24 examples. In about one third of the cases (7 out of 22 for PGLNS, 7 out of 24 for EBLNS), the range is equal to 0, which means that all runs meet the same last solution.

On the one hand, EBLNS finds the best results for the Still Life Problem (`still_life`), and is more stable than PGLNS, on these instances. It is also very appropriate to treat the Vehicle Routing Problem (`vrp`), the Itemset Mining Problem (`pattern_set_minning`) and the Price Collecting Problem (`pc`). However, the two approaches provide unstable results on theses instances, particularly on the Itemset Mining Problem ones where we can observe the largest ranges. On the Resource-Constrained Project Scheduling Problem (`rcpsp`), the results are very comparable, even though they are mildly in favor of PGLNS, and both approaches are very stable. On the Restaurant Assignment Problem (`fast_food`) and the League Model Problem (`league_model`), EBLNS finds equivalent or better solutions in more cases (7 out of 11 instances), and it tends to be more stable on average (27.3 % for PGLNS vs. 19.5 % for EBLNS).

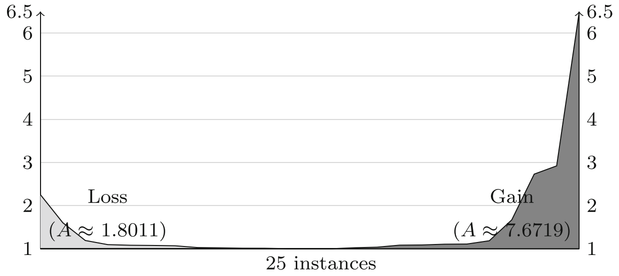


(a) Minimization problems.

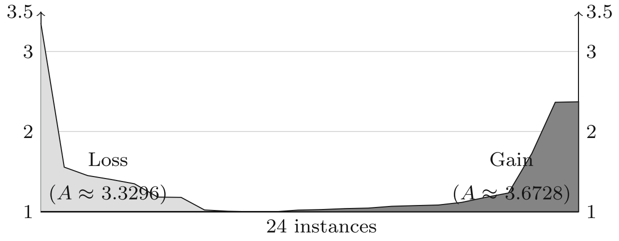


(b) Maximization problems.

Fig. 5 Multiplying factor between cftLNS and objLNS, per instance



(a) Minimization instances.



(b) Maximization instances.

Fig. 6 Multiplying factor between PGLNS and EBLNS, per instances

On the other hand, PGLNS finds the best results for the Ship Scheduling Problem (`ship_schedule`), and is more stable than EBLNS, on these instances. PGLNS is also a good approach to solve the Maximum Profit Subpath Problem (`mario`) and the Modified Car Sequencing Problem (`car_cars`), the problem it has originally been designed for. On those instances, the stability is again in favor of PGLNS; EBLNS is not stable at all on the Maximum Profit Subpath Problem instances.

Generally, each approach seems to be appropriate to some classes of problems. Such a trend is questioned by the stability: it varies from one instance to the other of the same class of problems. From now on, it is almost impossible to conclude on the quality of the neighborhoods build per approach. Because EBLNS relies on explanations, the overall process is slowed down, and this approach certainly suffers from that point of view, even if explanations help building good neighborhoods.

We now measure the gain of using EBLNS instead of PGLNS. The plots on Fig. 7 displays the multiplying factor over the objective value by using one approach instead of the other.

On minimization instances, each approach beats the other one in almost half of the instances treated, but the gain of using EBLNS instead of PGLNS is considerable. The dark gray area ($A \approx 7.6719$) is clearly greater than the light gray one ($A \approx 1.8011$). EBLNS significantly improves the objective value, and degrades it in a lesser extent. On maximization instances, the gain is less marked and is mildly in favor of EBLNS. The area are comparable, but EBLNS betters objective values, sometimes by a short head, on more instances than PGLNS: the dark gray area ($A \approx 3.6728$) is wider than it is tall.

By combining explanation-based neighborhoods together in EBLNS we were expecting to get the most from each of them and to globally improve the results and the stability. But, the results are mixed. On the one hand, we observe a poor gain regarding PGLNS on minimization problems: the dark gray area is smaller than the ones observed previously with objLNS and cftLNS. We even observe a bigger loss. On maximization problems, we

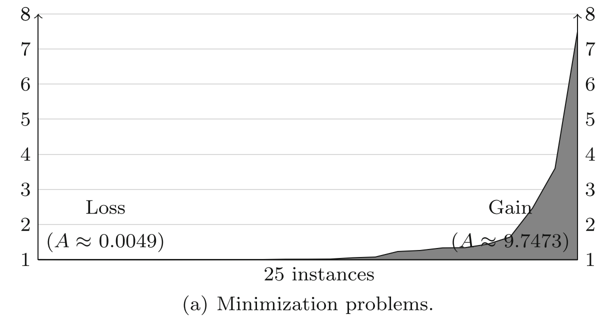
Table 9 Comparative evaluation of PGLNS and EBLNS

	PGLNS			EBLNS		
Example	obj	rng(%)	*v*stddev	obj	rng(%)	stddev
Minimization Problems						
`cars_cars_4_72.fzn`	**109.3**	5.49	1.85	118	2.54	0.77
`cars_cars_16_81.fzn`	121.2	2.48	1.17	**117.1**	11.10	5.84
`cars_cars_26_82.fzn`	**127.5**	5.49	2.20	139.5	3.58	1.96
`cars_cars_41_66.fzn`	**108.8**	1.84	0.98	110	0.00	0.00
`cars_cars_90_05.fzn`	**300.8**	11.97	9.93	321	0.00	0.00
`fastfood_ff3.fzn`	3883.4	90.64	967.10	**1330**	0.00	0.00
`fastfood_ff58.fzn`	8882.7	101.78	2955.56	**1373**	0.00	0.00
`fastfood_ff59.fzn`	871	0.00	0.00	**319.2**	21.30	28.09
`fastfood_ff61.fzn`	**221.7**	24.81	16.41	355	0.00	0.00
`fastfood_ff63.fzn`	**146.9**	40.16	17.70	331	0.00	0.00
`league_model20-3-5.fzn`	*49984*	0.00	0.00	*49984*	0.00	0.00
`league_model30-4-6.fzn`	**79973**	0.00	0.00	85974	23.26	7999.63
`league_model50-4-4.fzn`	**99971**	0.00	0.00	118971.7	33.62	11357.44
`league_model55-3-12.fzn`	139949	35.73	18439.09	**136950.5**	29.21	14865.70
`league_model90-18-20.fzn`	1922916	7.28	35228.00	**1152924.8**	26.89	88209.56
`league_model100-21-12.fzn`	3139911.9	0.00	2.17	**2651923.1**	80.70	716448.11
`rcpsp_11.fzn`	**81**	3.70	1.10	81.8	1.22	0.40
`rcpsp_12.fzn`	**38.3**	2.61	0.46	39.3	2.54	0.46
`rcpsp_13.fzn`	*78*	0.00	0.00	*78*	0.00	0.00
`rcpsp_14.fzn`	**140.9**	0.71	0.30	141	0.00	0.00
`vrp_A-n38-k5.vrp.fzn`	2341.6	22.16	155.59	**2114**	36.94	225.73
`vrp_A-n62-k8.vrp.fzn`	6318.1	6.66	127.92	**5840.4**	27.87	497.94
`vrp_B-n51-k7.vrp.fzn`	4291.8	15.87	189.12	**3950.8**	30.42	301.98
`vrp_P-n20-k2.vrp.fzn`	402.1	41.28	43.70	**364.2**	17.57	21.89
`vrp_P-n60-k15.vrp.fzn`	**2271.9**	12.28	91.15	2316.3	24.87	175.33

Table 9 (continued)

Example	PGLNS obj	PGLNS rng(%)	PGLNS stddev	EBLNS obj	EBLNS rng(%)	EBLNS stddev
Maximization Problems						
`mario_mario_easy_2.fzn`	*628*	0.00	0.00	*628*	0.00	0.00
`mario_mario_easy_4.fzn`	**506**	8.70	13.05	502.7	3.38	5.10
`mario_mario_n_medium_2.fzn`	**719.9**	31.95	61.85	610.5	125.31	274.10
`mario_mario_n_medium_4.fzn`	576.6	52.90	110.30	**711.9**	30.90	87.26
`mario_mario_t_hard_1.fzn`	**4783**	0.00	0.00	1426.1	282.87	1507.46
`pattern_set_mining_k1_ionosphere.fzn`	16.9	130.18	7.12	**29.2**	89.04	11.91
`pattern_set_mining_k2_audiology.fzn`	**53.6**	1.87	0.49	38.2	2.62	0.40
`pattern_set_mining_k2_german-credit.fzn`	3	0.00	0.00	**7.1**	169.01	4.61
`pattern_set_mining_k2_segment.fzn`	11.8	8.47	0.40	**28**	0.00	0.00
`pc_25-5-5-9.fzn`	62.8	3.18	0.98	**64.1**	3.12	0.54
`pc_28-4-7-4.fzn`	47.7	12.58	2.72	**49**	46.94	5.53
`pc_30-5-6-7.fzn`	**60**	0.00	0.00	58.7	13.63	2.65
`pc_32-4-8-0.fzn`	104.7	14.33	4.38	**108.8**	10.11	2.96
`pc_32-4-8-2.fzn`	84.9	15.31	6.14	**91.4**	6.56	1.80
`ship-schedule.cp_5Ships.fzn`	**483645.5**	0.01	13.50	483239	0.24	442.84
`ship-schedule.cp_6ShipsMixed.fzn`	**300185**	4.86	4378.36	253749	41.92	38561.83
`ship-schedule.cp_7ShipsMixed.fzn`	**396137**	20.40	23645.31	254441.5	102.45	79454.31
`ship-schedule.cp_7ShipsMixedUnconst.fzn`	**364141.5**	21.01	26324.16	269851	41.39	44658.25
`ship-schedule.cp_8ShipsUnconst.fzn`	**782516**	20.51	53704.49	539091	35.92	77158.01
`still-life_09.fzn`	37.6	13.30	1.50	**42**	0.00	0.00
`still-life_10.fzn`	49.8	4.02	0.87	**53.2**	3.76	0.60
`still-life_11.fzn`	59	0.00	0.00	**61.7**	1.62	0.46
`still-life_12.fzn`	63.3	4.74	0.90	**74.6**	5.36	1.20
`still-life_13.fzn`	79.7	2.51	0.78	**86.4**	6.94	2.01

Bold numbers highlight the best objective value per instances. Italic numbers denote equality



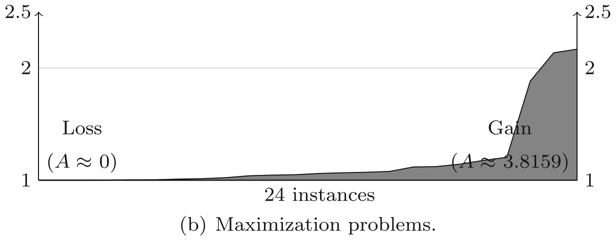


Fig. 7 Multiplying factor between PGLNS and PaEGLNS, per instance

observe a significant improvement concerning the gain, though. One may conclude that there might be too many explanation-based neighborhoods with respect to the total number of neighborhoods, which certainly slows down the overall process.

Thus, the two techniques are not comparable but they certainly are compatible. It is therefore natural to combine the neighborhoods of PGLNS and EBLNS together to address the defects of both approaches, and to improve the overall stability: we evaluate such approach in the next section.

4.5 Combining EBLNS and PGLNS

One of the strength of LNS is its ability to combine various neighborhoods together. In this Section, we combine the neighborhoods of EBLNS and PGLNS in a new contender and evaluate their efficiency.

Propagation and explanation guided LNS This contender is a combination of (1) `exp-obj`, (2) `exp-cft`, (3) `pgn`, (4) `repgn` and (5) `rapgn`. We simply concatenate the approaches and `rapgn` is preferred to `ran` because it brings robustness [23]. Each of the neighborhood is applied sequentially until a new solution is found. This contender is called *PaEGLNS*. We have also evaluated an adaptive version of PaEGLNS, which applies a neighborhood with respect to its ability to build new solution, but we have not reported its evaluation here as it was not competitive with the sequential approach.

Table 10 shows the evaluation of PaEGLNS. It reports the arithmetic mean of the objective value (obj), its range (rng) and the standard deviation (stddev) for PaEGLNS, PGLNS and EBLNS. PaEGLNS finds equivalent or better solutions in 69.4 % of the problems treated (34 out of 49). On the 16 other problems, PaEGLNS is always ranked second and EBLNS is almost always ranked first. Moreover, PaEGLNS overcomes PGLNS in about 77.5 % of the instances treated, and outperforms EBLNS in about 65.3 % of the instances

Table 10 Evaluation of PaEGLNS in comparison with PGLNS and EBLNS

	PaEGLNS			PGLNS			EBLNS		
Example	obj	rng(%)	stddev	obj	rng(%)	stddev	obj	rng(%)	stddev
Minimization									
`cars_cars_4_72.fzn`	*109.3*	4.57	1.42	*109.3*	5.49	1.85	118	2.54	0.77
`cars_cars_16_81.fzn`	119.6	9.20	3.72	121.2	2.48	1.17	**117.1**	11.10	5.84
`cars_cars_26_82.fzn`	127.8	4.69	2.48	**127.5**	5.49	2.20	139.5	3.58	1.96
`cars_cars_41_66.fzn`	*108.8*	1.84	0.98	*108.8*	1.84	0.98	110	0.00	0.00
`cars_cars_90_05.fzn`	**299.9**	11.34	9.16	300.8	11.97	9.93	321	0.00	0.00
`fastfood_ff3.fzn`	1573.4	31.78	207.55	3883.4	90.64	967.10	**1330**	0.00	0.00
`fastfood_ff58.fzn`	**1185**	9.28	30.19	8882.7	101.78	2955.56	1373	0.00	0.00
`fastfood_ff59.fzn`	**242**	0.00	0.00	871	0.00	0.00	319.2	21.30	28.09
`fastfood_ff61.fzn`	**153.9**	12.35	5.70	221.7	24.81	16.41	355	0.00	0.00
`fastfood_ff63.fzn`	**110**	34.55	10.74	146.9	40.16	17.70	331	0.00	0.00
`league_model20-3-5.fzn`	**49984**	0.00	0.00	49984	0.00	0.00	49984	0.00	0.00
`league_model30-4-6.fzn`	**79973**	0.00	0.00	79973	0.00	0.00	85974	23.26	7999.63
`league_model50-4-4.fzn`	**99971**	0.00	0.00	99971	0.00	0.00	118971.7	33.62	11357.44
`league_model55-3-12.fzn`	**110949.2**	9.01	2999.93	139949	35.73	18439.09	136950.5	29.21	14865.70
`league_model90-18-20.fzn`	1169916.1	8.55	30000.07	1922916	7.28	35228.00	**1152924.8**	26.89	88209.56
`league_model100-21-12.fzn`	3086911.2	4.54	36069.51	3139911.9	0.00	2.17	**2651923.1**	80.70	716448.11
`rcpsp_11.fzn`	**80**	3.75	1.00	81	3.70	1.10	81.8	1.22	0.40
`rcpsp_12.fzn`	38.5	2.60	0.50	**38.3**	2.61	0.46	39.3	2.54	0.46
`rcpsp_13.fzn`	*78*	0.00	0.00	*78*	0.00	0.00	*78*	0.00	0.00
`rcpsp_14.fzn`	*140.9*	0.71	0.30	*140.9*	0.71	0.30	141	0.00	0.00
`vrp_A-n38-k5.vrp.fzn`	**1901.1**	22.36	102.44	2341.6	22.16	155.59	2114	36.94	225.73
`vrp_A-n62-k8.vrp.fzn`	5879.8	15.36	232.56	6318.1	6.66	127.92	**5840.4**	27.87	497.94
`vrp_B-n51-k7.vrp.fzn`	4072.4	30.13	330.49	4291.8	15.87	189.12	**3950.8**	30.42	301.98
`vrp_P-n20-k2.vrp.fzn`	**299.9**	38.68	33.94	402.1	41.28	43.70	364.2	17.57	21.89
`vrp_P-n60-k15.vrp.fzn`	**2262.7**	13.66	77.94	2271.9	12.28	91.15	2316.3	24.87	175.33

Table 10 (continued)

Example	PaEGLNS obj	PaEGLNS rng(%)	PaEGLNS stddev	PGLNS obj	PGLNS rng(%)	PGLNS stddev	EBLNS obj	EBLNS rng(%)	EBLNS stddev
Maximization									
`mario_mario_easy_2.fzn`	*628*	0.00	0.00	*628*	0.00	0.00	*628*	0.00	0.00
`mario_mario_easy_4.fzn`	**507.6**	15.96	22.58	506	8.70	13.05	502.7	3.38	5.10
`mario_mario_n_medium_2.fzn`	**806.9**	42.76	104.25	719.9	31.95	61.85	610.5	125.31	274.10
`mario_mario_n_medium_4.fzn`	693.9	31.56	67.92	576.6	52.90	110.30	**711.9**	30.90	87.26
`mario_mario_t_hard_1.fzn`	*4783*	0.00	0.00	*4783*	0.00	0.00	1426.1	282.87	1507.46
`pattern_set_mining_k1_ionosphere.fzn`	**36.6**	87.43	8.89	16.9	130.18	7.12	29.2	89.04	11.91
`pattern_set_mining_k2_audiology.fzn`	**53.8**	1.86	0.40	53.6	1.87	0.49	38.2	2.62	0.40
`pattern_set_mining_k2_german-credit.fzn`	6.4	93.75	1.69	3	0.00	0.00	**7.1**	169.01	4.61
`pattern_set_mining_k2_segment.fzn`	22.2	72.07	5.08	11.8	8.47	0.40	**28**	0.00	0.00
`pc_25-5-5-9.fzn`	**64.2**	1.56	0.40	62.8	3.18	0.98	64.1	3.12	0.54
`pc_28-4-7-4.fzn`	**54.5**	23.85	4.39	47.7	12.58	2.72	49	46.94	5.53
`pc_30-5-6-7.fzn`	**60.8**	13.16	2.14	60	0.00	0.00	58.7	13.63	2.65
`pc_32-4-8-0.fzn`	**109.8**	20.04	6.19	104.7	14.33	4.38	108.8	10.11	2.96
`pc_32-4-8-2.fzn`	88.7	23.68	6.42	84.9	15.31	6.14	**91.4**	6.56	1.80
`ship-schedule.cp_5Ships.fzn`	**483650**	0.00	0.00	483645.5	0.01	13.50	483239	0.24	442.84
`ship-schedule.cp_6ShipsMixed.fzn`	**300664**	1.47	1737.49	300185	4.86	4378.36	253749	41.92	38561.83
`ship-schedule.cp_7ShipsMixed.fzn`	**399832**	3.15	5710.21	396137	20.40	23645.31	254441.5	102.45	79454.31
`ship-schedule.cp_7ShipsMixedUnconst.fzn`	**385896**	3.11	3492.69	364141.5	21.01	26324.16	269851	41.39	44658.25
`ship-schedule.cp_8ShipsUnconst.fzn`	**833724**	0.19	590.87	782516	20.51	53704.49	539091	35.92	77158.01
`still-life_09.fzn`	*42*	0.00	0.00	37.6	13.30	1.50	*42*	0.00	0.00
`still-life_10.fzn`	**53.3**	3.75	0.64	49.8	4.02	0.87	53.2	3.76	0.60
`still-life_11.fzn`	61.3	4.89	0.90	59	0.00	0.00	**61.7**	1.62	0.46
`still-life_12.fzn`	74.4	5.38	1.28	63.3	4.74	0.90	**74.6**	5.36	1.20
`still-life_13.fzn`	85.9	4.66	1.37	79.7	2.51	0.78	**86.4**	6.94	2.01

Bold numbers highlight the best objective value per instances. Italic numbers denote equality

treated. In general, combining the neighborhoods of PGLNS and EBLNS is profitable in term of quality of the solutions but also in stability. In 58,8 % of the instances better solved with PaEGLNS the range is less than 5 %, which means that almost all runs find the same last solution.

We now measure the gains and losses of using PaEGLNS instead of PGLNS or EBLNS. Figure 7 reports multiplying factor between PGLNS and PaEGLNS. The findings are unquestionable: PaEGLNS is clearly the most interesting approach with respect to PGLNS. There is almost no degradation, only those linked to `rcpsp_12` and `car_cars_26_82`, reflected in the plot by the non-zero light gray area (Fig. 7a). The comparison with EBLNS (Fig. 8) is mildly less advantageous, even though always widely in favor of PaEGLNS. There is no question that PaEGLNS is the best choice.

These two generic approaches are complementary: on the one hand, PGLNS builds the graph of dependencies between variables and detects closely linked subparts of the problem treated. On the other hand, EBLNS helps to focus on easy-to-repair and easy-to-improve neighborhoods by revealing the relationships existing between the objective variable and the decision variables. Combining PGLNS and EBLNS benefits from the advantages of the two approaches: exploiting the structure of the problems solved thank to both propagation and explanations, and bring more stability in the solutions found.

4.6 A deeper analysis

In the Section, we highlight some results that corroborate the previous ones found so far. A figure is made of a plot and an histogram. The plot depicts the evolution of the objective value along with the resolution time (in log scale) for the ten runs of the same approach (PGLNS, objLNS, cftLNS, EBLNS and PaEGLNS). The histogram reports the average

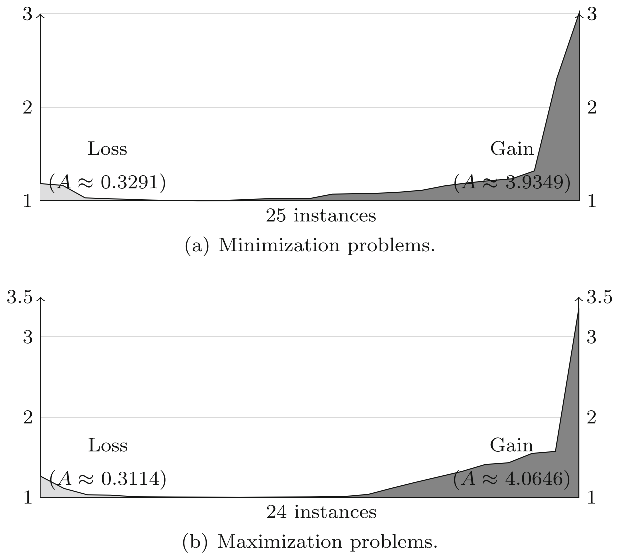


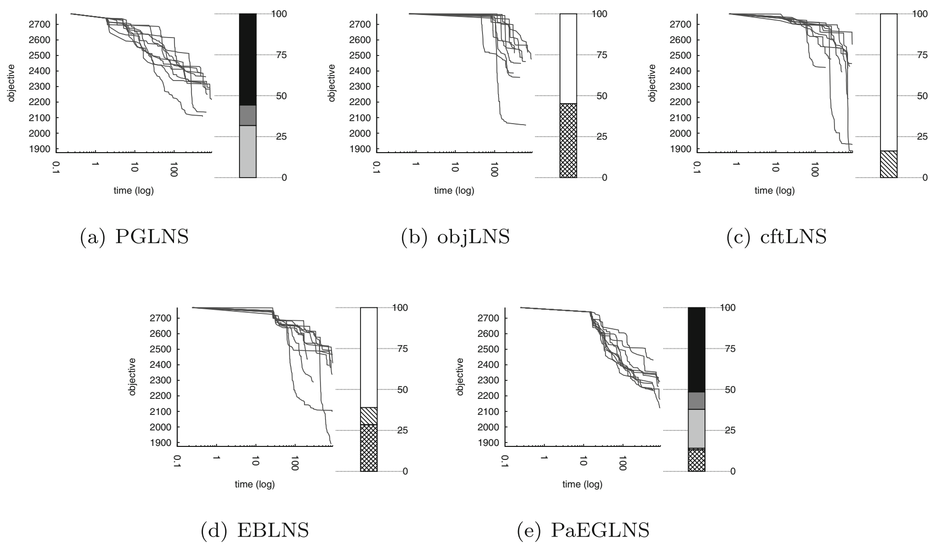
Fig. 8 Multiplying factor between EBLNS and PaEGLNS, per instance

repartition of the neighborhoods used to solve a given instance. The color coding is the following: `exp-obj` is in cross-patterned style, `exp-cft` is in slash-patterned style, `ran` is in plain white, `pgn` is in light gray, `repgn` is in dark gray and `rapgn` is in black.

On the Vehicle Routing Problem (Fig. 9), the histograms indicate that a great part of the intermediary solutions are found with the help of a pure random neighborhood. Its proportion varies from 54 % up to 83 %. Regarding PaEGLNS, which finds the better solution, we observe the same phenomenon: even if its contribution is reduced, the random neighborhood helps finding almost half of the new solutions. A comment is that random neighborhoods bring strong diversification and help solving the Vehicle Routing Problem but other neighborhoods play also a role, such as `exp-obj` and `pgn`.

On the Restaurant Assignment Problem (Fig. 10), where EBLNS works fine, the portion of `ran` (< 1 %) is negligible in comparison with `exp-obj` (34.2 %) and `exp-cft` (65.7 %). The stability of EBLNS on this instance is visible on the plot (Fig. 10d). We roughly observe a similar phenomenon about cftLNS. objLNS, on the other hand, relies much more on randomness (54.7 %). PaEGLNS not only betters the objective in comparison with PGLNS and EBLNS but also finds it quickly (see Tables 9 and 10), and the distribution of neighborhoods is well balanced. Surprisingly, the contribution of `exp-cft` is significantly reduced in PaEGLNS in comparison with EBLNS, which is not the case for `exp-obj` whose portion remains stable (about 35 %). Another comment is that the best solutions are found faster when combining various neighborhoods.

On the Prize Collecting Problem (Fig. 11), we observe that the resolution relies only on `rapgn` for PGLNS. Concerning the other combinations, on the contrary, the contribution of purely random neighborhoods to the solution discovery is lower. Another remarkable point is the proportion of `exp-obj` and `exp-cft` in EBLNS and PaEGLNS: more than 60 % of the solutions found by the two contenders relies on these neighborhoods. However, the combination of different neighborhoods of PaEGLNS overcomes both EBLNS and PGLNS, and confirms the importance of combining various neighborhoods in the same contender.



(a) PGLNS (b) objLNS (c) cftLNS

(d) EBLNS (e) PaEGLNS

Fig. 9 Solving `vrp_P-n60-k15.vrp` instance with the five approaches

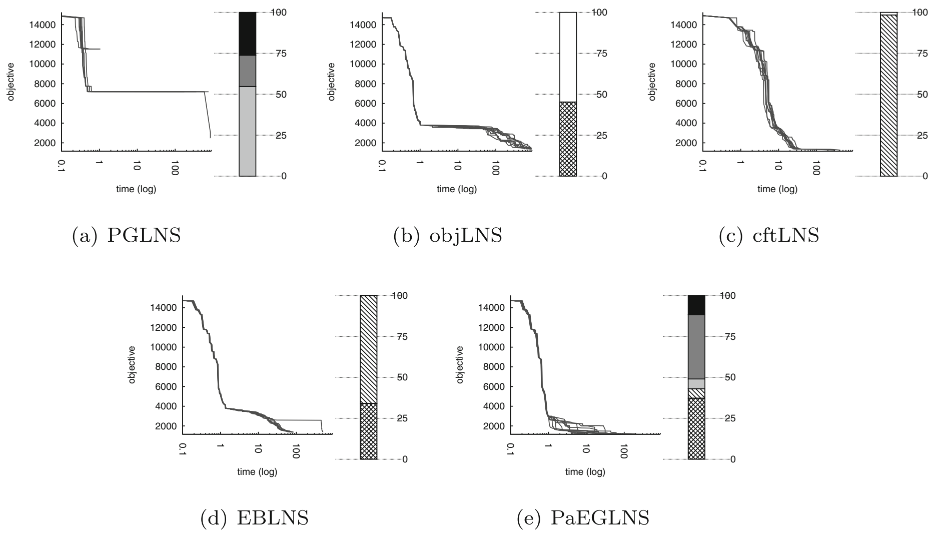


Fig. 10 Solving `fastfood_ff58` instance with the five approaches

Finally, on the Still Life Problem (Fig. 12), using `exp-obj` and `exp-cft` is the key to success. However, an alteration of the performances can be observed when other neighborhoods are added (Fig. 12e). This shows the risk of combining various and different neighborhoods together, it may also break the semantic of each of them. Such a result appears to be marginal in our evaluation, though, and does not enable to disconfirm the benefit of combining neighborhoods together.

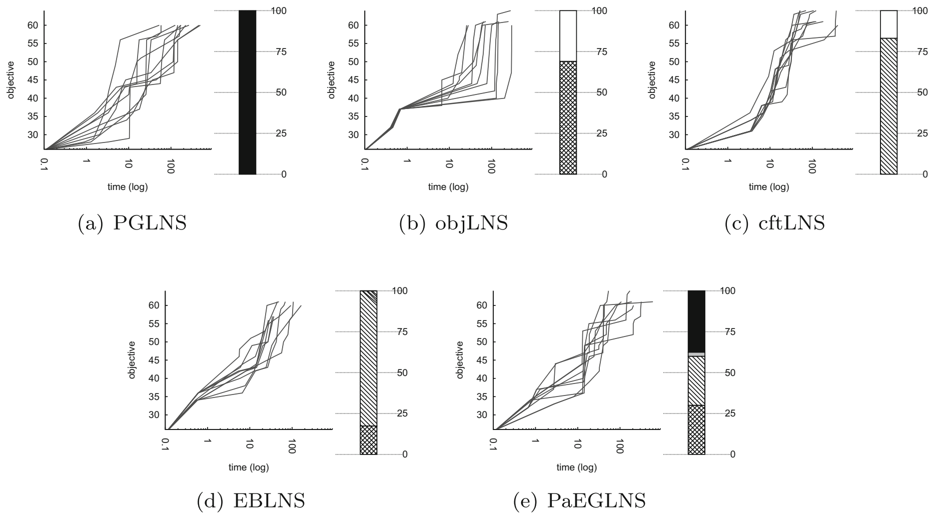


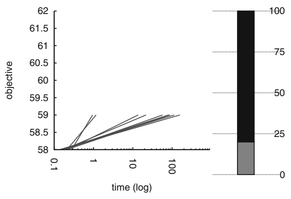
Fig. 11 Solving `pc_30-5-6-7` instance with the five approaches

All the plots are given in the online Appendix. One can see that, on the Restaurant Assignment Problem and the Itemset Mining Problem and, albeit to a lesser extent, on the Price Collecting Problem and the Still Life Problem, EBLNS is very stable. Moreover, these are problems that are well solved with our approach. Except the Vehicle Routing Problem, the portion of RLNS in EBLNS is marginal.

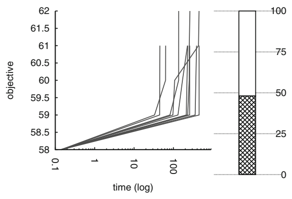
In this Section, we evaluated the `exp-obj` and `exp-cft`, two neighborhoods for LNS based on explanations. We limited the evaluation to models without global constraints, and none of the approaches evaluated here have benefited from possibly better filtering rules and explanations schemas. We show that objLNS, cftLNS and EBLNS are competitive with PGLNS on a wide range of optimization problems. Even if they are globally less stable and mildly slower, they enable to tackle some class of problems PGLNS poorly solved. In general, those combinations bring more gain than loss. In addition, we confront those first results with a combination of the propagation-guided neighborhoods and explanation-based ones, and show that such a combination improves the results, brings more stability and exploits advantageous every neighborhoods. Such a conclusion confirms previous studies: a strong diversification contributes to LNS and combining different neighborhoods is the key to efficiency.

5 Conclusion and future work

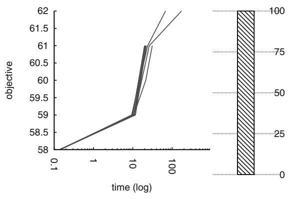
In this paper, we show that the issue of finding a problem independent neighborhood generation technique for LNS can be addressed using a lazy variation of explanations. The contributions are twofold. Firstly, we propose generic, configuration-free approaches to compute neighborhoods in a LNS, based on explanations. The first neighborhood is



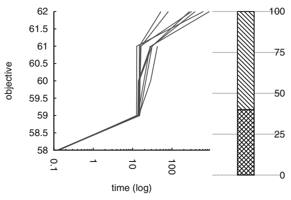
(a) PGLNS



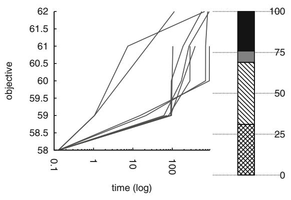
(b) objLNS



(c) cftLNS



(d) EBLNS



(e) PaEGLNS

Fig. 12 Solving `still-life_11` instance with the five approaches

based on the conflict implied by the application of the cut, the second one is based on the non optimal nature of the current solution. We address the diversification issues thanks to a set of heuristics for further selecting variables mixing random approaches and explanation-based ones and show that our approach is competitive with or even better than state-of-the-art generic neighborhoods, on a set of optimization problems. Secondly, we scheme explanation-based neighborhoods and propagation-guided ones, hoping that their behavior would be complementary. Finally, we assess the new contender on the same range of problems, and validate the combination to be very efficient and more stable.

Our results are encouraging and should be validated on a larger set of problems. Moreover, we have to put things in perspective and analyze the influence of the model on the results, specifically on the Modified Car Sequencing Problem, where the results of PGLNS are quite surprising. Both the search strategy (how decisions are selected) and the propagation engine (how events are relayed in the constraint network) influence explanation-based neighborhoods and propagation-guided-ones. But, since the explanation of a domain reduction is not unique, our approach would benefit from more concise explanations, e.g., by enabling global constraints. Future works should focus on the diversification procedure, more particularly plugging NoGoods in should be the key combination to speed up the exploration of nested search spaces. Finally, the lazy and asynchronous way to store and compute explanations may be improved using incrementallity. It would also be interesting to implement and evaluate such neighborhoods within a LCG solver.

References

1. Atkins, K.E. (1989). *An introduction to numerical analysis*.
2. Cambazard, H., & Jussien, N. (2006). Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, *11*(4), 295–313.
3. Chabrier, A., Danna, E., Le Pape, C., Perron, L. (2004). Solving a network design problem. *Annals of Operations Research*, *130*(1–4), 217–239.
4. Copado-Méndez, P.J., Blum, C., Guillén-Gosálbez, G., Jiménez, L. (2013). Application of large neighborhood search to strategic supply chain management in the chemical industry. In *Hybrid metaheuristics* (pp. 335–352). Springer.
5. Danna, E., & Perron, L. (2003). Structured vs. unstructured large neighborhood search: a case study on job-shop scheduling problems with earliness and tardiness costs. In F. Rossi (Ed.)*, Principles and practice of constraint programming? Lecture notes in computer science, CP 2003* (Vol. 2833, pp. 817–821*)*. Berlin Heidelberg: Springer.
6. Debruyne, R., Ferrand, G., Jussien, N., Lesaint, W., Ouis, S., Tessier, A. (2003). Correctness of const raint retraction algorithms. In *FLAIRS'03: 16th international Florida artificial intelligence research society onference* (pp. 172–176). St. Augustine: AAAI press.
7. Demir, E., Bektaş, T., Laporte, G. (2012). An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research*.
8. Gent, I.P., Miguel, I., Moore, N.C.A. (2010). Lazy explanations for constraint propagators. In*,* M. Carro & R. Pea (Eds.)*, Practical aspects of declarative languages, lecture notes in computer science* (Vol. 5937, pp. 217–233). Berlin Heidelberg: Springer.
9. Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, *1*, 25–46.
10. Godard, D., Laborie, P., Nuijten, W. (2005). Randomized large neighborhood search for cumulative scheduling. In: *ICAPS* (Vol. 5, pp. 81–89).
11. Harvey, W., & Schimpf, J. (2002). *Bounds consistency techniques for long linear constraints*.
12. Jussien, N., & Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, *139*(1), 21–45.

13. Jussien, N. (2003). *The versatility of using explanations within constraint programming*. Technical Report 03-04-INFO.
14. Jussien, N., Debruyne, R., Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In l*ecture notes in computer science* (no. 1894, pp. 249–261). . Singapore: Springer-Verlag.
15. Jussien, N., & Lhomme, O. (1998). Dynamic domain splitting for numeric csps. In *European conference on artificial intelligence (ECAI'98)* (pp. 224–228).
16. Kovacs, A.A., Parragh, S.N., Doerner, K.F., Hartl, R.F. (2012). Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of Scheduling*, *15*(5), 579–600.
17. Laborie, P., & Godard, D. (2007). Self-adapting large neighborhood search:application to single-mode scheduling problems. In P. Baptiste, G. Kendall, A. Munier-Kordon, F. Sourd (Eds.), *Proceedings of the 3rd multidisciplinary international conference on scheduling: theory and applications (MISTA 2007)* (pp. 276–284). Paris. Paper.
18. Mairy, J.-B., Deville, Y., Hentenryck, P.V. (2011). Reinforced adaptive large neighborhood search. In *8th workshop on local search techniques in constraint satisfaction (LSCS2011).*
19. Mairy, J.-B., Schaus, P., Deville, Y. (2010). Generic adaptive heuristics for large neighborhood search. In *7th workshop on local search techniques in constraint satisfaction (LSCS2010).*
20. Malitsky, Y., Mehta, D., O'Sullivan, B., Simonis, H. (2013). Tuning parameters of large neighborhood search for the machine reassignment problem. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 176–192). Springer.
21. Ohrimenko, O., Stuckey, P.J., Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, *14*(3), 357–391.
22. Perron, L. (2003). Fast restart policies and large neighborhood search. In Rossi, F. (Ed.)*, Principles and practice of constraint programming at CP 2003, lecture notes in computer science* (Vol. 2833). Berlin Heidelberg: Springer.
23. Perron, L., Shaw, P., Rueher, M. (2004). Combining forces to solve the car sequencing problem. In J.-C. Régin (Ed.)*, Integration of AI and OR techniques in constraint programming for combinatorial optimization problems,ecture notes in computer science* (Vol. 3011, pp. 225–239). Berlin Heidelberg: Springer.
24. Perron, L., Shaw, P., Furnon, V. (2004). Propagation guided large neighborhood search. In *CP'04* (pp. 468–481).
25. Pisinger, D., & Ropke, S. (2010). Large neighborhood search. In *Handbook of metaheuristics* (pp. 399-419). Springer.
26. Pralet, C., & Verfaillie, G. (2004). Travelling in the world of local searches in the space of partial assignments. In J.-C. Régin, & M. Rueher (Eds.)*,lecture notes in computer science,* (Vol. 3011, pp. 240–255). Springer.
27. Prosser, P. (1995). *MAC-CBJ: maintaining arc consistency with conflict-directed backjumping*. Technical Report Research Report/95/177, Department of Computer Science, University of Strathclyde.
28. Prud'homme, C., & Fages, J.-G. (2013). Introduction to choco3. In *1st workshop on CPSolvers: modeling, applications, integration, and standardization, CP*. http://choco.emn.fr.
29. Roli, A., Benedettini, S., Stützle, T., Blum, C. (2012). Large neighbourhood search algorithms for the founder sequence reconstruction problem. *Computers & Operations Research*, *39*(2), 213–224.
30. Schiex, T., & Verfaillie, G. (1994). Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, *3*(2), 187–207.
31. Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M., & J.-F. Puget (Eds.)*, Principles and practice of constraint programming, CP98, volume lecture notes in computer science* (Vol. 1520, pp. 417–431). Berlin Heidelberg: Springer.
32. Stuckey, P.J. (2010). Lazy clause generation: combining the power of sat and cp (and mip?) solving. In *CPAIOR* (pp. 5–9).
33. Verfaillie, G., & Jussien, N. (2005). Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, *10*(3), 253–281.