

# MINLP Problems and Explanation-based Constraint Programming

Guillaume Rochart<sup>1,2</sup>, Eric Monfroy<sup>1</sup>, and Narendra Jussien<sup>2</sup>

<sup>1</sup> LINA, FRE CNRS 2729

2, rue de la Houssinière – B.P. 92208 – F-44322 Nantes Cedex 3

<sup>2</sup> Département Informatique, École des Mines de Nantes

4, rue Alfred Kastler – B.P. 20722 – F-44307 Nantes Cedex 3

**Abstract.** Numerous industrial problems can be modelled as MINLP problems combining both numeric and integer variables. Several methods were proposed to solve these problems. But industrial applications need more than solving problems: dynamic problems, over-constrained problems, or explaining solver behaviour are features required by industrial applications. Explanation-based constraint programming offers such tools. In this paper, we show how to apply explanation-based mechanisms for mixed problems thanks to a generic framework. Last, some first experimental results are exposed: the overhead due to explanation managing is acceptable and can even speed up some resolutions.

## Introduction

Numerous industrial problems can be modelled as MINLP (*Mixed Integer Non-Linear Programming*) problems combining both numeric and integer variables: design of water or gas networks, automobile, aircraft, etc. [4]. These problems are really hard to solve: they combine the combinatorial nature of mixed integer programming and the intrinsic difficulty of nonlinear programs. Several methods were proposed to solve such problems [4]: branch-and-bound, extended cutting plane methods, generalised Bender's decomposition, etc.

But industrial applications need more than solving problems. Problems can be dynamic, this implies that constraints may be added or removed dynamically. Moreover, if no solution is found, the user often needs to know why the problem is over-constrained, or why the expected solution is inconsistent.

Constraint programming offers generic models and tools to solve combinatorial problems. Furthermore, explanation-based constraint programming provides tools to solve dynamically such problems and maintain explanations about the resolution: *why* a problem has no solution, *why* the optimum bound is reached, or *how* to improve a solution are informations that explanation-based constraint programming can provide. Such features are now well known for constraint programming over integer variables [6].

However only few works proposed solutions to extend it to real variables. [8] proposed to extend `mac-dbt` to solve numeric problems thanks to a dynamic domain splitting mechanism. But these works solve separately integer problems

and numeric problems. Moreover, no solution is proposed for the main drawback about explanations for numeric problems: slow convergence of propagation may need to store a huge amount of explanations. This may make prohibitive using explanations with these problems.

Here, we propose both a generic framework to solve MINLP problems with explanation-based constraint programming and some ideas to decrease the number of explanations to store, by filtering redundant or useless explanations.

In the following, we first define MINLP problems, explanations and explanation-based constraint programming. Then, we introduce and extend propositions of [8]. Some propositions are made in order to reduce the number of explanations the solver must store. Last, we present some experimentations about these propositions and explanation-based resolution of MINLP problems.

## 1 Context

### 1.1 MINLP problems

**MINLP** *Mixed Integer Non Linear Programming* consists in finding solutions to problems combining the combinatorial nature of mixed integer programming (MIP) with nonlinear program (NLP) difficulties [4]. These problems basically contain:

- discrete and numeric variables;
- linear and nonlinear constraints.

Since both MIP and NLP are NP-hard problems, solving such problems can be really complex. [4] lists several solutions to solve such problems. However, in order to use constraint programming features like generic models, dynamic solving or explanations about the resolutions, it may be useful to consider such problems as constraint satisfaction problems.

Indeed constraint programming provides tools for solving integer combinatorial problems and solving numeric problems [2]. In this paper, we show how to solve such problems modelled as CSP (with integer, numeric or mixed constraints) with explanation-based constraint programming.

**Constraint Programming** A CSP (*Constraint Satisfaction Problems*) is modelled as a set of variables with their domains and a set of constraints over these variables. Thus CSP modelling can be used to model most of the MINLP problems.

Constraint programming offers a generic framework to solve CSP thanks to both enumerating algorithm to search a solution through the search tree and filtering algorithm to reduce this search space:

- *filtering algorithms* are based on reduction operators: given the current domains of the variables in a constraint, those operators can deduce inconsistent values. For discrete constraints, they can be based on graph algorithms like the `all_different` [12] constraint for instance. With numeric

constraints, these operators are usually based on interval arithmetic (cf section 2.1): given an interval an expression must be in, some values are not consistent. For instance if  $x + y$  must be in  $[1, 2]$  and  $x \in [0, 2], y \in [-2, 3]$ , then  $y \in [-1, 2]$ : indeed suppose  $y = 3$  then  $x + y \in [3, 5]$  which has no common values with the constrained interval  $[1, 2]$ .

- *enumerating algorithms* consist in making hypothetical choices. If these choices are not good (no solution can be found), a backtrack mechanism undoes these choices and offers new ones. These hypothesis are called *decision constraints* and can be:
  - *instantiation constraints* for discrete variables: each consistent value is tried for each variable,
  - *splitting constraints* for numeric variables: a solution is searched in the first half of the domain then in the second half.

However, classical algorithms cannot explain why no solution is found, or why a value is not kept for a variable. Some industrial applications now need to give some valuable information back to the user. In the next section, we present how to combine such search algorithms with explanations.

## 1.2 Explanations and constraints

**Explanations** Explanations are useful features to dynamically solve a problem and to give some valuable information to the user about the resolution of the CSP. An explanation contains information about both the search state and the constraints implying the deduced variable domain modification. We can define an explanation as follows:

**Definition 1 (Explanation).** *An explanation of an inference ( $\mathcal{X}$ ) is a subset of original constraints ( $\mathcal{C}' \subset \mathcal{C}$ ) and decision constraints (choices made during the search:  $d_1, d_2, \dots, d_k$ ) such that:  $\mathcal{C}' \wedge d_1 \wedge \dots \wedge d_n \Rightarrow \mathcal{X}$ .*

**Explanation Usage** Such explanations are useful to improve solution search: they avoid thrashing by indicating precisely the past choices responsible of a contradiction. Indeed, several algorithms are based on explanations like **backjumping** or **mac-dbt**[7]. When a contradiction occurs, **backjumping** algorithms retracts all decisions taken since the responsible decision was made. **mac-dbt** is more powerful since it only removes the responsible decision without retracting intermediary decisions. This is possible thanks to incremental constraints able to update their data structures. However, this is not the only one usage of such explanations nor the more interesting.

*Dynamic resolution* Explanations make dynamic problems easier to solve [13], since removing constraints is quite easy as soon as we know exactly which domain reductions are due to this constraint. Indeed, since the algorithm stores the constraints responsible for each domain modification, it can precisely determine which value withdrawals should be undone after removing a given constraint.

For instance, consider domains for  $x$  an integer variable in  $\llbracket 1, 5 \rrbracket$  and  $y$  a numeric variable in  $[-10, 10]$  explained with the following explanations:

- $x \neq 1$  because of  $\{c_2, c_3\}$  and  $x \neq 2$  because of  $\{c_1, c_2, c_3\}$ ,
- $y \geq 2.3$  because of  $\{c_3, c_4\}$  and  $y \leq 5.1$  because of  $\{c_1, c_3\}$ .

In this case, if the constraint  $c_1$  is removed, the domains become  $\llbracket 2, 5 \rrbracket$  and  $[2.3, 10]$ . This basically means that the domains do not need to be computed from scratch after each constraint removal.

*Resolution explanations* Explanations are also particularly useful for user interaction. Indeed when a problem has no solution, the user is not given any further information. Explanations can point out constraints responsible for such a contradiction, helping the user to debug the model or to find data to be modified in order to make the problem feasible.

All users do not need the same information: a final user does not need to know all implementing constraints whereas the developer can require as much information as possible to debug the model. This implies that these explanations must be customised depending on the profile of the user. [9] proposed tools offering such features to users of explanation-based constraint programming.

Last, several works were led to offer user-friendly tools based on these explanations. For instance, thanks to the OADYMPPAC trace, some tools were provided to display and help to interpret explanations during the problem resolution (see <http://contraintes.inria.fr/OADymPPaC/> for details about this trace): for instance, [5] proposed a visualisation tool based on matrices to show the constraints the most used in explanations. Those constraints are basically the hardest part of the problem to solve. So if the user see that one of these constraints can be removed, the problem can be made easier to solve dynamically.

## 2 Numeric CSP and explanations

Numeric CSP are CSP with numeric variables modelled as real intervals. To solve such problems, algorithms are very similar to those for integer CSP. Indeed, instead of simply assigning variables, search algorithms cut intervals to lead a dichotomic search. With respect to propagation, filtering algorithms are mainly based on interval arithmetic [2, 1]. This arithmetic determines all values an expression can be equal to, according to the variable domains. For instance, given  $x \in [1, 2]$  and  $y \in [4, 5]$ , interval arithmetic ensures that  $x * y \in [4, 10]$ .

### 2.1 Explained interval arithmetic

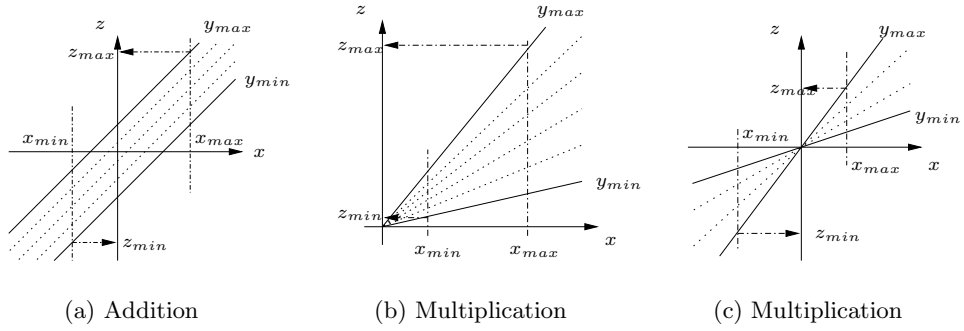
**Explaining Interval Arithmetic Operators** We could use trivial explanations (composed of domain explanations of all variables involved in the constraints) but those explanations are not precise enough for improving problem solving nor user interaction. This is why an explanation-based solver must be able to explain as precisely as possible why a domain is modified.

With respect to interval arithmetic, this means that each operator should be able to know which bounds are responsible for the returned value.

## Application to Classical Binary Operators

*The Addition Operator* To illustrate how we explain interval operators, we consider in this section the addition operator.

Interval arithmetic implies that  $x + y = [\underline{x} + \underline{y}, \bar{x} + \bar{y}]$  (see figure 1(a)) where  $\underline{x}$  (resp.  $\bar{x}$ ) is the lower bound (resp. upper bound) of  $x$ . This means that lower bound of  $x + y$  depends only on the lower bounds of  $x$  and  $y$ . Thus if a constraint  $z = x + y$  is posted, when  $x$  or  $y$  bounds are modified, the following explanation is generated for justifying the new lower bound (the upper bound is similar):  $expl(z \geq \underline{x} + \underline{y}) = expl(x \geq \underline{x}) \cup expl(y \geq \underline{y})$ . Since these latter explanations are stored for all variables, the algorithm can inductively build this new explanation.



**Fig. 1.** Interval operators

In this case, generated explanations are always the same, but this property is not universal: the multiplication operator explanations are a bit more complex.

*The Multiplication Operator* Multiplication depends on almost all bounds:  $x * y = [\min(\underline{x} * \underline{y}, \underline{x} * \bar{y}, \bar{x} * \underline{y}, \bar{x} * \bar{y}), \max(\underline{x} * \underline{y}, \underline{x} * \bar{y}, \bar{x} * \underline{y}, \bar{x} * \bar{y})]$ .

However, as illustrated on figure 1(b), if  $x \geq 0$  and  $y \geq 0$ , the rule is simplified into  $x * y = [\underline{x} * \underline{y}, \bar{x} * \bar{y}]$

This means, that, if  $x \geq 0 \wedge y \geq 0$ , the explanations can be optimised:

$$expl(z \geq \underline{x} * \underline{y}) = expl(x \geq \underline{x}) \cup expl(y \geq \underline{y})$$

$$expl(z \leq \bar{x} * \bar{y}) = \underbrace{expl(x \geq 0) \cup expl(y \geq 0)}_{\text{rule justification}} \cup \underbrace{expl(x \leq \bar{x}) \cup expl(y \leq \bar{y})}_{\text{value justification}}$$

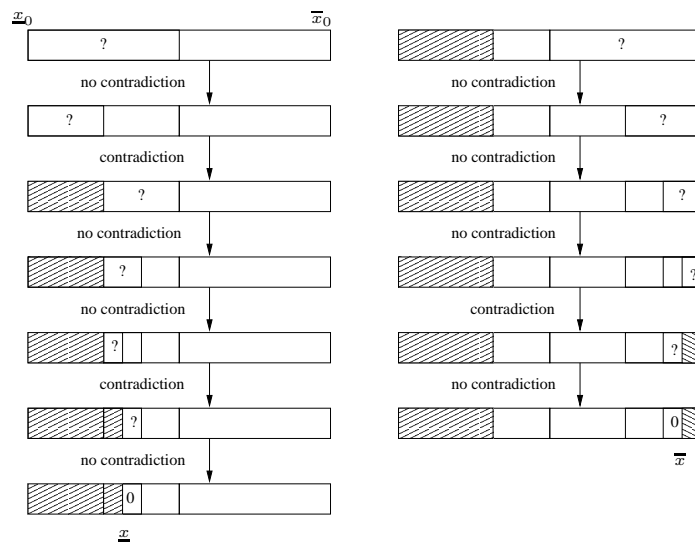
In the last equation, the first part justifies why the rule is valid (indeed, figure 1(c) shows that this is not true if  $x$  or  $y$  can be less than 0), while the second part explains the new bound of  $z$ . This idea can be generalised for the different cases (nine cases for the multiplication operator), in order to optimise the explanation computation (here the lower bound depends only on two bound).

This example shows that all operators of interval arithmetic must be studied so as to determine as precisely as possible which bounds are responsible of the interval value of the expression: subtraction is similar to addition, division similar to multiplication (except the case of null denominator), unary monotonic function bounds are explained by single bound explanation, and so on.

## 2.2 Box consistency

The box consistency algorithm [2, 1] consists in finding the right-most and left-most quasi-zeros<sup>3</sup> with respect to one variable of the constraint: like the shaving technique [11], box consistency avoids exploring parts of the search tree where *a priori* no solution can be found. To find these reduced intervals, the variable interval is recursively cut to check if a solution can be found in the left-most and the right-most subintervals. The algorithm stops as soon as the domain has been explored or that the maximal depth is reached (the depth is 4 on figure 2).

[8] introduced how to use explained interval arithmetic as described in previous section in order to implement a dynamic splitting algorithm. This can be directly applied to hull consistency. Actually, it can be used for box consistency as well as soon as we can explain why box-consistency deduces that a sub-interval of the domain is inconsistent.



**Fig. 2.** Box consistency: impossible subintervals are filtered through a dichotomic algorithm (a first subinterval is tested, since nothing can be proven, a subsubinterval is tested...) with a given maximum depth (here 4)

<sup>3</sup> *i.e.* intervals where the algorithm cannot prove there is no solution.

Several algorithms can be used to check if a solution can be found in an interval (like the Newton method for instance[1], or just narrowing algorithms). If this algorithm is explanation-friendly (like a narrowing algorithm using explained interval arithmetic), generating bound explanations is quite easy. Indeed, the bound explanations are composed of the explanation of the lack of solution in the filtered subinterval (that is contradiction explanations for narrowing algorithm) and the explanation of the previous value of the bound:

$$\text{expl}(x \geq \underline{x}) = \text{expl}(x \geq \underline{x}_0) \cup \bigcup_{I \in \text{inconsistent subintervals}} \text{expl}(x \notin I)$$

### 3 Generic framework for numeric and integer resolution

Previous works focused on solving problems with explanations either on pure integer problems [6] or on pure numeric problems [8]. Actually, a generic framework can easily be derived from these two resolution frameworks.

In order to make discrete and numeric parts of the problem cooperate, the same algorithm can be used. Indeed the algorithm proposed by [8] is very close to the generic dynamic backtracking algorithm implemented in PaLM[6]. This means that a common search algorithm can solve a MINLP problems thanks to appropriate branching algorithms.

#### 3.1 Notions

To introduce this algorithm, we need to define two new notions: *dependent* and *entailed* constraints.

**Dependent constraints** Dependency notion is necessary to ensure that if an unary constraint on a variable is removed (like a splitting constraint during enumeration), all the unary constraints that semantically depend on this constraint are removed too.

**Definition 2 (Dependent constraint).** *A constraint  $c$  is dependent w.r.t. a constraint  $c'$  if and only if  $c$  is irrelevant if  $c'$  is not consistent anymore.*

For instance, if we consider two consecutive splitting constraints  $c_1 \equiv v \in [1.0, 2.0]$  and  $c_2 \equiv v \in [1.0, 1.5]$ ,  $c_2$  is made dependent of  $c_1$  since if  $c_1$  is not consistent, it is irrelevant to try to maintain  $c_2$  consistent.

**Entailed constraints** Entailment is needed to describe constraints which are deduced by the search algorithm. For instance, if we know that  $\{c_1, c_2, c_3\}$  is inconsistent, then we can deduce that if  $c_1$  and  $c_2$  are respected, we must ensure that  $\neg c_3$ . But as soon as  $c_1$  or  $c_2$  is removed, this constraint is not implied anymore.

**Definition 3 (Entailed constraint).** *A constraint  $c$  is entailed w.r.t. a set of constraint  $\mathcal{S}$  if and only if  $c$  is logically implied by  $\mathcal{S}$ .*

For example, consider that a contradiction occurs because of a set  $\mathcal{S}$  of constraints (the explanation of the contradiction). If a decision constraint  $c \in \mathcal{S}$  is removed to repair the problem, the opposite constraint  $\neg c$  can be posted. However, the new constraint is logically implied only as long as the other constraints of  $\mathcal{S}$  are consistent. In this case,  $c$  is an entailed constraint for the set  $\mathcal{S} \setminus \{c\}$ .

One can note that an *entailed* constraint is a *dependent* constraint. The main difference is that an entailed constraint is directly explained by the constraints this constraint depends on. By this way, the opposite constraint  $\neg c$  is never present in an explanation and the algorithm will not try to propagate its opposite again.

### 3.2 Algorithms

The algorithms on Fig. 3 and 4 describe how to solve a mixed CSP over discrete and numeric variables. These algorithms are very close to the one provided by [6] for discrete variables.

```

(1) begin
(2)   while unassignedVars  $\neq \emptyset$  do
(3)      $c \leftarrow$  branching.getDecisionConstraint()   % Constraint to add
(4)     try
(5)       problem.post( $c$ )
(6)       problem.propagate()   % Filtering
(7)     catch
(8)       problem.handleContradiction()   % Decision repairing
(9)     endtry
(10)  endwhile
(11) end

```

**Fig. 3.** Generic explanation-based algorithm

Thanks to this generic algorithm, specific work is done directly by the branching mechanisms. For instance, the branching algorithms on Fig. 5 and 6 can be used. For the numeric variable branching, the generated splitting constraints are made dependent *w.r.t.* the previous splitting constraints on the same variables. This allows to remove a dependent constraint automatically as soon as one of these constraints this constraint depends on is removed.

### 3.3 Collaboration between numeric and integer parts

During resolution, the solver must choose branchings for the enumerating algorithm in order to choose the next variable to branch on. Here we chose to first instantiate integer variables since their domains are obviously smaller than numeric domains.

```

(12) begin
(13)   e ← problem.getContradictionExplanation()
(14)   if e.isEmpty() then
(15)     problem.raiseProblemContradiction()   % Over-constrained problem
(16)   else
(17)     ct ← e.selectConstraintToRemove()
(18)     try
(19)       problem.remove(ct)   % Incremental constraint removal
(20)       % Removes recursively all dependent constraints
(21)
(22)       e.delete(ct)
(23)       problem.post(opposite(ct), e)   % Contextual posting
(24)       % forbids ct to be reintroduced
(25)       % ¬ct is an entailed constraint w.r.t. e
(26)       problem.propagate()   % Re-propagation
(27)     catch
(28)       problem.handleContradiction()   % Possible recursive contradiction
(29)     endtry
(30)   endif
(31) end

```

**Fig. 4.** Contradiction handling for mac-dbt algorithm

```

(32) begin
(33)   % Integer variable branching
(34)   x ← problem.getMinDomVariable()
(35)   v ← x.getInf()
(36)   return (x = v)
(37) end

```

**Fig. 5.** Example of integer branching mechanism

Then the enumerating algorithm posts the constraints returned by the branching algorithms. Thanks to this mechanism, the search tree is explored for both integer variables and for numeric variables.

Moreover, explanations for all domains can contain both assigning constraints for the integer variables and splitting constraint for numeric variables. This implies that if a numeric domain becomes empty because of an integer variable affectation, only this assigning constraint will be removed without removing all work done on numeric variables. This makes collaboration between the integer and numeric parts transparent and efficient.

### 3.4 Sketch of a proof of completeness

It is well known that **dynamic backtracking** is a complete algorithm for solving discrete CSP. A similar proof can be used to prove that its extension to numeric problems is still complete. Intuitively, to prove that an algorithm is complete, we need to prove that the algorithm does not skip any solution and that it terminates. The following sketch of proof can be used:

- **All solutions are found:** instantiation or splitting constraints are locally removed if and only if the filtering algorithms can prove that there is no solution with such constraint given the other constraints. The main idea

```

(38) begin
(39)     % Numeric variable branching
(40)      $x \leftarrow \text{problem.nextNumericVariable}()$     % Cyclic variable choice
(41)      $i \leftarrow x.\text{firstHalf}()$ 
(42)      $(x \in i).\text{setDependent}(\text{previous splitting constraints on } x)$ 
(43)     return  $(x \in i)$ 
(44) end

```

**Fig. 6.** Example of numeric branching mechanism

here is that all skipped potential values of variables are logically implied by the constraints of the systems.

- **The algorithm terminates:** to prove that the algorithm terminates, we need to prove that the algorithm cannot go back to a past node in the search tree. Consider that the decision constraint  $c$  is locally inconsistent (given all constraints of the problem and other decision constraints).
  - If the contradiction explanation contains only this decision constraint, the constraint  $\neg c$  is posted for ever. This means that the algorithm ensures that the decision  $c$  will not be covered anymore.
  - If the contradiction explanation contains  $n$  other decision constraints, the constraint  $\neg c$  is posted as entailed constraint *w.r.t.* the  $n$  other decision constraints (all these constraints depend on less constraints than this new one which depends on the union). This means that as long as these constraints are not removed, the decision  $c$  is not covered anymore.

Inductively, we can deduce that the same affectation of the variables cannot be covered several times by the algorithm. Thus, since the search space is finite (this is obvious for discrete variables and this is due to float precision for numeric variables), the algorithm terminates.

## 4 Filtering useful explanations

The main drawback of using explanations is the overhead of computing and managing explanations. Moreover, due to slow convergence to the fix point for numeric problems [10], using explanations with numeric problems can be prohibitive. In this section, we present some ideas to decrease the number of explanations to store. We present two main kinds of filtering to select explanations to be kept: arbitrary filtering based on heuristics, and semantic filtering based on the semantics of the constraints the variable is involved in.

### 4.1 Semantic filtering

One way to filter explanations consists in selecting semantically useful explanations: when a variable is involved in a mixed constraint (that is a constraint which contains integer variables too), only few numeric values are really pertinent.

Indeed, even if several explanations are provided to explain a modification of the bound between two discrete values, only one of them is needed to justify the

new bound value: if the bound is changed from  $-5$  to  $-4.9$  and from  $-4.9$  to  $-4.86$  and if this variable must have integer values (due to an equality constraint for instance), only explanations from  $-5$  to  $-4.9$  is necessary to explain the new bound  $-4$  that will be obtained by the mixed constraint (the other explanation can be ignored).

For instance, consider the equality constraint between an integer variable and a numeric variable. In this case, to explain why the constraint is inconsistent, only integer bound modifications must be explained. This means that explanations must be stored only when integer value of the numeric bound is modified.

This can be extended to most of discrete mixed constraints. For instance, let us consider an **element** constraint extended to numeric variables: such constraint can be defined with a numeric variable  $v$ , an integer variable  $x$  and an array of numeric values  $t$ . The constraints ensures that  $x = i \Leftrightarrow v = t[i]$ . In this case, only domain modification implying inconsistent values in  $t$  would be kept.

## 4.2 Arbitrary filtering

Another way for filtering explanations to decrease the amount of explanations are arbitrary methods. They are based on heuristics in order to guess which explanations are useful and which ones should be merged to other explanations.

**Regular explanations** Some explanations can be useful only if the domain modification is great enough. Indeed, explaining why the lower bound is not  $\alpha$  but  $\alpha + \varepsilon$  is certainly useless for the user. This means that explanations could be stored only if the domain reduction is more than a constant  $\delta_{min}$ .

However, this selection heuristic may be hard to tune. Indeed, if  $\delta_{min}$  is too large, the search slows down dramatically, since the information kept is not accurate enough to improve search.

We could also use dynamic values of  $\delta_{min}$  depending on the current domain size for instance.

**Splitting-based explanations** Another heuristic can consist in storing one explanation for all reductions deduced between two decision constraint posts. In this case, before storing an explanation, the last explanation stored about the variable domain is checked in order to determine if it was stored before the last taken decision.

**Similar explanations** A more simple heuristic consists in checking if the last explanation of the involved variable bound contains the same constraints as the explanation given for the domain reduction. In this case no new explanation is stored.

## 5 Experimentations

As explained in previous sections, the main drawback of using explanations is the overhead due to maintaining and storing explanations as long as the problem is solved. However, as our experiments confirm, this overhead is acceptable given the information such explanations can give for dynamic resolution purpose or for valuable information to be returned to the user. With some structured problems, using `mac-dbt` can even speed up the search of solutions thanks to explanations [8].

First we present some results on a pure numeric problems to validate the filtering we introduced in the previous section. Then we solve a problem from the MINLP library of problems [3], to apply these techniques on more realistic problems. All these experimentations were launched on a Pentium 4 2GHz, 256 Mb, using Java version of `Choco` and `PaLM` (<http://choco.sourceforge.net>).

### 5.1 Filtering explanations

In this experimentation (this a problem from the `Elisa` framework available at <http://sourceforge.net/projects/elisa>), we solve a numeric CSP defined on three variables  $x$ ,  $y$  and  $z$ , such that:

$$\begin{cases} x^2(1 + y^2) + y(y - 24x) = -13 \\ y^2(1 + z^2) + z(z - 24y) = -13 \\ z^2(1 + x^2) + x(x - 24z) = -13 \end{cases}$$

All 16 solutions of this system of equations were searched using a basic implementation of box-consistency with the following explanation sets:

- no explanation,
- classical explanations,
- regular explanations (changes must be more than 1 to be explained in a new explanation),
- explanations filtered with respect to similar explanations: only different successive explanations are stored,
- explanations filtered according to the splitting constraint posts.

We used `mac` algorithm to solve experiments without explanation and `mac-dbt` otherwise. We found the results in Table 1. The evaluation of the memory usage is a gross approximation since it is based on the memory used by the JVM at the end of the resolution. However, it is hard to have more precise evaluation without performance alteration.

These results show that the overhead on a classical problem is acceptable if we consider the information explanations add to user interaction and dynamic solving. Moreover, filtering explanations in order to decrease the number of explanations to store seems to be useful since it can decrease the CPU time by up to 10% for the filtering over distinct explanations. Combining such a method with the filtering accelerating method proposed by [10] may make using explanations almost free in terms of time of resolution overhead.

**Table 1.** CPU time (average on ten runs) to solve a pure numeric problem with explanations or not, and an evaluation of the necessary memory

Type of explanation	CPU time (ms)	Evaluation of memory (kbytes)
No explanation	13666	3480
All explanations	41497	15242
Regular explanations	47313	7590
Only distinct explanations	36995	7275
Splitting-based explanations	41748	10335

## 5.2 Solving MINLP problems with e-CP

In this experimentation, the problem consists in affecting pipe diameter in a water network. This problem is proposed in the MINLP library [3]. In this problem, several parts of a city are modelled as vertices in a graph, and several edges are provided as potential pipes to provide water distribution. On each edge, we must determine: whether there is pipe, what the kind of pipe is, what the direction of the flow is (if there is a pipe), the amount of flow (constrained by flow conservation and flow supplies or demands).

Moreover, the pressure must respect pressure loss equations and must be enough in each points according to the altitude of the involved part of the city.

To make it more simple to implement, we only consider two diameters of pipes: the problem must choose between one pipe (diameter of value 2) or no pipe at all (diameter of value 0) on the different arcs of the network.

This problem is interesting to solve since it is composed of:

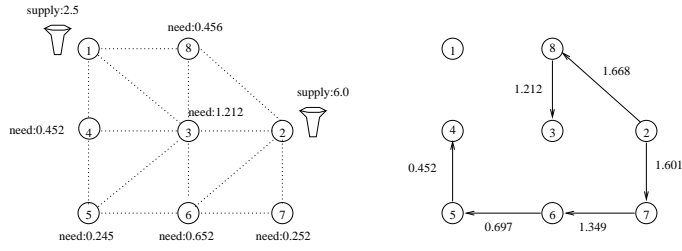
- *integer constraints*: for instance, the link between the diameter (0 or 2) and the boolean variable stating whether there is a pipe or not;
- *mixed constraint*: linking integer variables and numeric variables (equality and `element` constraints);
- *numeric constraints*: linking flow variables and pressure variables to ensure flow conservation and pressure loss due to pipe diameters, for instance.

If we consider the first decision solution found, Table 2 show results very similar to those of the previous section: filtering explanations decreases the time needed to solve the problem for most of the heuristics. Moreover the difference between solving with explanation and solving without explanation is less important. This may be due to the structure of this problem which can be exploited by explanations, as shown in the next section.

Similar results were found on other experiments like a problem of spring design minimising the volume of metal necessary to build it given some specifications.

**Table 2.** CPU time (average on ten runs) to solve a MINLP problem with explanations or not

Type of explanation	CPU time
No explanation	2814
All explanations	4659
Regular explanations	4418
Only distinct explanations	4351
Splitting-based explanations	4434



**Fig. 7.** The problem restricted to the flow assigning and a solution (obviously not an optimal one with respect to the pressure loss and cost)

### 5.3 Structured problems

In the previous examples, explanations are useful only for user information and interaction. But in some particular cases, where the problem is well structured, explanations can be useful to decrease the CPU time to find a solution too.

Indeed, our experiments confirm results found in [8]: with the following equation system, the explanation-based algorithms (with box consistency) find a solution in few seconds (less than 6 seconds) with an approximation of  $10^{-4}$  whereas with classical approach, no solution is found before more than ten minutes ([8] let it run during several hours without any solution).

$$\begin{aligned}
 x_1 + x_2 + x_3 + x_4 &= x_l & y_1 + y_2 + y_3 + y_4 &= y_l \\
 x_1 + x_2 - x_3 + x_4 &= 3 & y_1 + y_2 - y_3 + y_4 &= 3 \\
 x_1^2 + x_2^2 + x_3^2 + x_4^2 &= 4 & y_1^2 + y_2^2 + y_3^2 + y_4^2 &= 4 \\
 x_1^2 + x_2^2 + x_3^2 + x_4^2 - 2 * x_1 &= 3 & y_1^2 + y_2^2 + y_3^2 + y_4^2 - 2 * y_1 &= 3 \\
 x_i \in [-10^{10}, 10^{10}], x_l \in [-0.2, 0.2] & & y_i \in [-10^{10}, 10^{10}], y_l \in [-0.2, 0.2] &
 \end{aligned}$$

Of course, in this pathological case, symbolic transformation would have detected the redundancy and would certainly have divided this problem in two subproblems. But in more subtle situations, such feature can be very useful to improve resolution.

## 6 Conclusion

In this paper, we presented results about using explanations with MINLP resolutions. Indeed, giving explanations to the user to explain the result is a feature that many industrials need. These first results are encouraging since the overhead is quite acceptable, all the more that heuristics for filtering explanations seems to be efficient to decrease this overhead.

Further works should be led both to test as precisely as possible the efficiency of the different heuristics and to improve the first implementation used in these experimentations. Moreover, this work could be extended to apply it to solver collaboration and not only for intra-solver cooperation. Indeed, explanations (or at least nogoods) could be valuable information to exchange between different kinds of solvers since they give information about past computations of each solver.

## References

1. Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *ICLP'99*, pages 230–244, 1999.
2. Frédéric Benhamou, David McAllester, and Pascal Van Hentenryck. CLP(Intervals revisited). In *International Logic Programming Symposium*. MIT Press, 1994.
3. Michael R. Bussieck, Arne Stolbjerg Drud, and Alexander Meeraus. Minlplib - a collection of test models for mixed-integer nonlinear programming. In *INFORMS J. Comput.*, 2003.
4. Michael R. Bussieck and Armin Pruessner. Mixed-integer nonlinear programming. In *SIAG/OPT Newsletter: Views & News*, 2003.
5. Mohammad Ghoniem, Narendra Jussien, and Jean-Daniel Fekete. VISEXP: visualizing constraint solver dynamics using explanations. In *FLAIRS'04: Seventeenth international Florida Artificial Intelligence Research Society conference*, Miami, Florida, USA, 2004.
6. Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, pages 118–133, Singapore, 2000.
7. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'00*, pages 249–261, 2000.
8. Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric CSP. In *European Conference on Artificial Intelligence*, pages 224–228, 1998.
9. Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.
10. Yahia Lebbah and Olivier Lhomme. Accelerating filtering techniques for numeric csps. *Artificial Intelligence*, pages 109–132, 2002.
11. Paul Martin and David B. Shmoys. A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. In W. H. Cunningham, S. T. McCormick, and M. Queyranne, editors, *Proceedings of IPCO'96*, pages 389–403, Vancouver, British Columbia, Canada, 1996.
12. Jean-Charles Régim. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelfth National Conference on AI*, pages 362–367, Seattle, 1994.
13. Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. Technical report, EMN, 2004.