Controlled Propagation in Continuous Numerical Constraint Networks

Frédéric Goualard
Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière – BP 92208
F-44322 Nantes cedex 3
Frederic.Goualard@lina.univ-nantes.fr

Laurent Granvilliers
Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière – BP 92208
F-44322 Nantes cedex 3
Laurent.Granvilliers@lina.univ-nantes.fr

ABSTRACT

Local consistency is usually enforced on continuous constraints by decomposing them beforehand into so-called primitive constraints. It has long been observed that such a decomposition drastically slows down the computation of solutions. Five years ago, Benhamou et al. introduced an algorithm that avoids formally decomposing constraints, and whose efficiency is often on a par with state-of-the-art methods. It is shown here that this algorithm implements a strategy to enforce on a continuous constraint a consistency akin to directional bounds consistency as introduced by Dechter and Pearl for discrete problems. The actual impact of decomposition is also thoroughly analyzed by means of new experimental results.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking*; G.1.0 [**Numerical Analysis**]: General—*interval arithmetic*

General Terms

Algorithms, Performance

Keywords

Local consistency, directional consistency, constraint propagation

1. INTRODUCTION

Waltz's seminal paper [11] promoted the idea of local consistency enforcement to solve constraints. A system of constraints was solved by considering each of them in turn, discarding the values in the domains of the variables involved that could not possibly be part of a solution. Montanari [9] and Mackworth [8] introduced the notion of a network of constraints in which a more involved scheme for propagating domain modifications can be used. Davis [3] later adapted these works to solve continuous problems by employing interval arithmetic [10] to handle the domains of the variables.

The first solvers to implement the ideas of Davis and others were enforcing on continuous constraints a relaxation of arc consistency, *bounds consistency* (aka 2B consistency), which is better suited to continuous domains. For practical reasons, bounds consistency can only be effectively enforced on binary and ternary constraints. More complex constraints then have to be decomposed into such simpler constraints, thereby augmenting the number of variables and constraints to eventually consider.

Five years ago, Benhamou et al. presented the HC4 algorithm [1], which is strongly related to the methods employed to enforce bounds consistency except for its ability to handle constraints without formal decomposition. HC4 was shown to exhibit performances on a par with state-of-the-art interval arithmetic-based methods on some large problems. In their paper, the authors did not analyze HC4 on a theoretical point-of-view. They claimed, however, that it would enforce bounds consistency on the system of primitive constraints stemming from the decomposition of a constraint containing no more than one occurrence of any variable.

The first contribution of this paper is the characterization of the consistency HC4 enforces on one constraint in terms of the equivalent on continuous domains of *directional bounds consistency* introduced by Dechter and Pearl [5]. We also disprove Benhamou et al.'s claim concerning it computing bounds consistency for constraints with variables occurring at most once, and we prove that HC4 enforces *union consistency* on such constraints. The actual impact of the decomposition of constraints is also investigated by analyzing new experimental results that make extensive use of S-boxes [7] ability to control the propagation in a constraint network.

2. LOCAL CONSISTENCY TECHNIQUES

Consider *variables* from an infinite countable set $\{x_1, x_2, \ldots\}$, with their associated domains of possible values $D[x_1]$, $D[x_2]$, ..., and *constraints*, which state some relation between variables. A *constraint* C on a finite set of variables $S[C]$—its *scope*—with domains D is a subset of the product of their domains.

A *constraint problem* P is formally characterized by a triplet (V, D, R), where V is a finite set of variables with domains D, and R is a set of constraints such that the union of their scopes is included in V.

The basic step to solve a constraint problem corresponds to the inspection of each of its constraints C in turn and to the removal of all the values in the domains of the variables in $S[C]$ that cannot be part of a solution of C. To this end, we need to be able to project C onto each of its variables. This notion of *projection* is formally stated below.

DEFINITION 1 (PROJECTION OF A CONSTRAINT). *Let C be a constraint, D a Cartesian product of domains, and x a variable in C. The* projection *of C on x w.r.t. D is the set $\Pi(C, D, x)$ of values in $D[x]$ that can be extended to a solution of C in D.*

A projection is then the set of *consistent* values of a variable relative to a constraint. The well-known *arc consistency* property [8] demands that all values in a domain be consistent. This property is clearly too strong for our purpose since it has no practical counterpart for continuous domains in the general case. We then only consider weaker consistency notions, depending on the *approximation function* we use to manipulate real relations: given $\mathbb{I}$ the set of intervals whose bounds are representable numbers (*floating-point numbers*), and $\mathbb{U}$ the set of finite unions of such intervals, let $\text{hull}(\rho) = \bigcap\{D \in \mathbb{I}^n \mid \rho \subseteq D\}$, and $\text{union}(\rho) = \bigcap\{D \in \mathbb{U}^n \mid \rho \subseteq D\}$, for all $\rho \subseteq \mathbb{R}^n$.

Using the *hull* (resp. *union*) approximation on intervals (resp. unions of intervals), we may define *bounds consistency* (resp. *union consistency*) for continuous problems. Both consistency notions may be abstracted to one *apx consistency* that only requires instantiation of *apx* by either *hull* over $\mathbb{I}$ or *union* over $\mathbb{U}$, to retrieve both consistency notions.

Table 1: Computing an apx consistent CSP

```
Algorithm: APX-CONSISTENCY
Input: a constraint problem $P = (V, D, R)$
Output: an apx consistent equivalent problem
1.  $\mathscr{S} \leftarrow R$
2.  while $\mathscr{S} \neq \varnothing$ do
3.      $C \leftarrow$ choose an element of $\mathscr{S}$
4.      $D' \leftarrow \text{Revise}(C, D)$
5.      foreach $x_i$ s.t. $D'[x_i] \subsetneq D[x_i]$ do
6.          $\mathscr{S} \leftarrow \mathscr{S} \cup \{C' \mid C' \in R \wedge x_i \in S[C']\}$
7.          $D[x_i] \leftarrow D'[x_i]$
8.      endfor
9.      $\mathscr{S} \leftarrow \mathscr{S} \setminus \{C\}$        % Revise is idempotent
10. endwhile
```

DEFINITION 2 (APX CONSISTENCY). *Given a constraint C, a Cartesian product of domains D, and a variable x in $S[C]$, x is said to be* apx consistent *w.r.t. C and $D[x]$ if and only if $D[x] = \text{apx}(\Pi(C, D, x))$.*

A constraint C is said to be apx consistent w.r.t. a Cartesian product of domains D if and only if every variable x in its scope is apx consistent relative to C and $D[x]$. A constraint system is apx consistent w.r.t. D if and only if each of its constraints is apx consistent w.r.t. D.

Solving a constraint problem P means computing all tuples of the product of domains that satisfy *all* the constraints. The general algorithm is a search strategy called *backtracking*. The computation state is a search tree whose nodes are labelled by a set of domains. The backtracking algorithm requires a time exponential in the number of variables in the worst case. Its performances can be improved with local consistency enforcement to reduce the domains prior to performing the search. For instance, the domain of a variable x that is not *bounds consistent* relative to a constraint C can be reduced by the following $\text{Revise}(C, D, x)$ operation:

$$[\min(D[x] \cap \Pi(C, D, x)), \max(D[x] \cap \Pi(C, D, x))]$$

The consistency of a constraint network is obtained by a constraint propagation algorithm. An AC3-like algorithm [8] is given in Table 1, where the "$\text{Revise}(C, D)$" operation applies $\text{Revise}(C, D, x)$ on each variable x in $S[C]$ in an arbitrary order.

As noted by Dechter [4], it may not be wise to spend too much time in trying to remove as many inconsistent values as possible by enforcing a "perfect local consistency" on each constraint with Alg. APX-CONSISTENCY. It may be indeed more efficient to defer part of the work to the search process.

Table 2: Computing a directional apx consistent CSP

```
Algorithm: DApxC
Input: – a constraint problem $P = (V, D, R)$
       – a strict partial ordering $\prec$ over $V$
       – an ordered partition $\Gamma_1, \ldots, \Gamma_q$ of $V$
         that is compatible with $\prec$
Output: a directional apx consistent equivalent problem
1. for $i = q$ downto 1 do
2.     foreach $C \in R$ such that $\Gamma_i \subseteq S[C]$ and
3.                           $S[C] \subseteq \Gamma_1 \cup \cdots \cup \Gamma_i$ do
4.         foreach $x \in S[C] - \Gamma_i$ do
5.             $D[x] \leftarrow \text{Revise}(C, D, x)$
6.         endfor
7.     endfor
8. endfor
```

The amount of work performed can be reduced by adopting the notion of *directional consistency* [5], where inferences are restricted according to a particular variable ordering.

DEFINITION 3 (DIRECTIONAL APX CONSISTENCY). *Let D be a Cartesian product of domains. A constraint system R is* directional apx consistent *relative to D and a strict partial ordering on variables if and only if for every variable x and for every constraint $C \in R$ on x such that no variable y of its scope is smaller than x, x is apx consistent relative to C and $D[x]$.*

A propagation algorithm for directional apx consistency, called DApxC, is presented in Table 2. It is adapted from Dechter's directional consistency algorithms. We introduce a partition $\Gamma_1, \ldots, \Gamma_q$ of the set of variables V that is compatible with the given partial ordering "$\prec$": two different variables x and y, such that x precedes y ($x \prec y$), must belong to two different sets Γ_i and Γ_j with $i < j$.

3. RELATING DAPXC AND HC4

In this section, we first present the practical aspects of enforcing either bounds or union consistencies; next, we show that the revising procedure for the HC4 algorithm enforces a directional apx consistency.

3.1 Revising Procedures for Apx Consistency

According to Def. 2, the enforcement of bounds and union consistencies on a real constraint requires the ability to project it on each of its variables and to intersect the projections with their domains.

In the general case, the accumulation of rounding errors and the difficulty to express one variable in terms of the others will preclude us from computing a precise projection of a constraint. However, such a computation may be performed for constraints involving no more than one operation ($+$, $\times$, cos, ...), which corresponds to binary and ternary constraints such as $x \times y = z$, $\cos(x) = y$, ...

As a consequence, complex constraints have to be decomposed into conjunctions of binary and ternary constraints (the *primitives*) prior to the solving process.

Enforcing apx consistency on a primitive is obtained by using interval arithmetic [10]. To be more specific, the revising procedure APXrevise(C, D) for a constraint like $C\colon x + y = z$ is implemented as follows:

$$\begin{cases} \text{Revise}(C, D, x)\colon & D[x] \leftarrow D[x] \cap (D[z] \ominus D[y]) \\ \text{Revise}(C, D, y)\colon & D[y] \leftarrow D[y] \cap (D[z] \ominus D[x]) \\ \text{Revise}(C, D, z)\colon & D[z] \leftarrow D[z] \cap (D[x] \oplus D[y]) \end{cases}$$

where $\ominus$ and $\oplus$ are interval extensions of the corresponding real arithmetic operators over either $\mathbb{I}$ or $\mathbb{U}$, depending on the instantiation of *apx* desired.

Enforcing apx consistency on a constraint system is performed in two steps: the original system is first decomposed into a conjunction of primitives, adding fresh variables in the process; the new system of primitives is then handled with the APX-CONSISTENCY algorithm described in Table 1, where the Revise procedure is performed by an APXrevise algorithm for each primitive. Let *HC3 (resp. UC3) be the APX-CONSISTENCY algorithm applied to primitive constraints only, and where* apx *is instantiated with the* hull *(resp.* union*) approximation.*

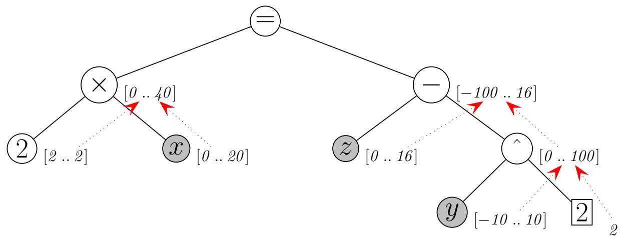
3.2 HC4revise: a Revising Procedure for Directional Apx Consistency

The HC4 algorithm was originally presented by its authors [1] as an efficient means to compute bounds consistency on complex constraints with no variable occurring more than once (called *admissible constraints* in the rest of the paper). It was demonstrated to be still more efficient than HC3 to solve constraints with variables occurring several times, though it was not clear at the time what consistency property is enforced on any single constraint in that case.

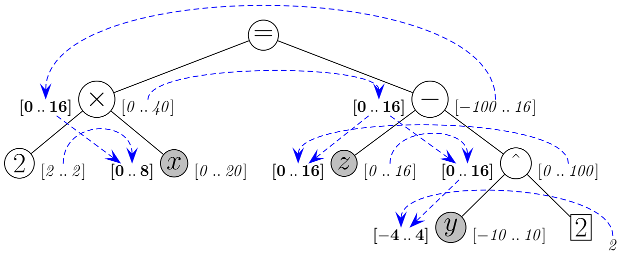
To answer that question, we first describe briefly below the revising procedure HC4revise of HC4 for one constraint C as it was originally presented, that is in terms of a two sweeps procedure over the expression tree of C. We will then relate this algorithm to the one presented in Table 2.

To keep the presentation short, the HC4revise algorithm will be described by means of a simple example. The reader is referred to the original paper [1] for an extended description.

Given the constraint $C\colon 2x = z - y^2$, HC4revise first evaluates the left-hand and right-hand parts of the equation using interval arithmetic, saving at each node the result of the local evaluation (see Fig. 1(a)). In a second sweep from top to bottom on the expression tree (see Fig. 1(b)), the domains computed during the first bottom-up sweep are used to project the relation at each node on the remaining variables.



(a) Forward sweep



(b) Backward sweep

Figure 1: HC4revise on the constraint $2x = z - y^2$

Given the constraint set $\Delta_C = \{2x = \alpha_1, y^2 = \alpha_2, z - \alpha_2 = \alpha_3, \alpha_1 = \alpha_3\}$ obtained by decomposing C into primitives, it is straightforward to show that HC4revise simply applies all the Revise procedures in Δ_C in a specific order—induced by the expression tree of C—noted ω_C.

To be more specific, HC4revise first applies the Revise procedure for the right-hand variable of all the primitives up to the root, and then the Revise procedures for the left-hand variables:

1. Revise$(2x = \alpha_1, D, \alpha_1)$
2. Revise$(y^2 = \alpha_2, D, \alpha_2)$
3. Revise$(z - \alpha_2 = \alpha_3, D, \alpha_3)$
4. Revise$(\alpha_1 = \alpha_3, D, \alpha_3)$
5. Revise$(\alpha_1 = \alpha_3, D, \alpha_1)$
6. Revise$(2x = \alpha_1, D, x)$
7. Revise$(z - \alpha_2 = \alpha_3, D, z)$
8. Revise$(z - \alpha_2 = \alpha_3, D, \alpha_2)$
9. Revise$(y^2 = \alpha_2, D, y)$

Note that, so doing, the domain of each fresh variable introduced by the decomposition process is set to a useful value before being used in the computation of the domain of any other variable.

For admissible constraints, the HC4revise algorithm can be implemented using the DApxC algorithm by considering two well-chosen partitions of the set of variables of the decomposed problem. Non-admissible constraints need being made admissible by adding new variables to replace multiple occurrences. The partitioning scheme is given by tree traversals as follows: The first partition Γ is obtained by a right-to-left preorder traversal of the tree where visiting a node has the side effect of computing the set of variables associated with its children. The underlying strict partial ordering of the variables is such that a child is greater than its parent. The second partition Γ' is obtained by inverting the partition computed by a left-to-right preorder traversal of the tree where the visit of a root node associated with a variable x just computes the set $\{x\}$. The underlying strict partial ordering is such that a child is smaller than its parent. HC4revise is equivalent to applying DApxC on Γ and then on Γ'.

Going back to our example, let us consider the CSP $P = (V, D, \Delta_C)$, with $V = \{x, y, z, \alpha_1, \alpha_2, \alpha_3\}$, $D = D[x] \times D[y] \times D[z] \times D[\alpha_1] \times D[\alpha_2] \times D[\alpha_3]$, and Δ_C defined as above. Let us also consider a dummy fresh variable α_0 supposed to be in the scope of the constraint represented by the root node and its children ($\alpha_1 = \alpha_3$), which is only introduced to initialize the computation of projections[1]. The partitions used to apply HC4revise on P by using Alg. DApxC are then as follows:

$$\begin{cases} \Gamma &= \{\alpha_0\}, \{\alpha_1, \alpha_3\}, \{z, \alpha_2\}, \{y\}, \{x\} \\ \Gamma' &= \{y\}, \{\alpha_2\}, \{z\}, \{\alpha_3\}, \{x\}, \{\alpha_1\}, \{\alpha_0\} \end{cases}$$

Let γ_C (resp. γ'_C) be the partial ordering induced by Γ (resp. Γ') on the variables. With its two sweeps on a tree-shaped constraint network, HC4revise is very similar to Dechter's *ADAPTIVE-TREE-CONSISTENCY* algorithm [4, p. 265]. More importantly, the constraint network processed by Alg. DApxC being a tree, we can state a result analogous to the one stated by Freuder [6] for arc consistency:

PROPOSITION 1. *Given a constraint C and a Cartesian product of domains D for the variables in $S[C]$, let Δ_C be the set of primitives obtained by decomposing C. We have:*

1. *Δ_C is directional bounds consistent w.r.t. γ'_C and $D'' = \mathrm{HC4revise}(C, D)$ for $D \in \mathbb{I}^n$;*
2. *if C is an admissible constraint, the constraint system represented by Δ_C is union consistent w.r.t. $D'' = \mathrm{HC4revise}(C, D)$ for $D \in \mathbb{U}^n$.*

PROOF. *Let Γ and Γ' be two partitions for the variables in $V = \bigcup_{C' \in \Delta_C} S[C']$ defined as described above. As stated previously, we have $\mathrm{HC4revise}(C, D) = \mathrm{DApxC}(\langle V, D', \Delta_C\rangle, \gamma'_C, \Gamma')$, where $D' = \mathrm{DApxC}(\langle V, D, \Delta_C\rangle, \gamma_C, \Gamma)$.*

The first point follows directly from this identity. To prove the second point, let us consider the set Π_C of projection operators implementing the Revise procedures for the primitives in Δ_C. The UC3 algorithm applied on Δ_C and D would compute the greatest common fixed-point $\top$ included in D of these operators, which is unique since they all are monotonous [2]. By design of UC3, Δ_C is union consistent w.r.t. $\top$.

Consider now HC4revise called on C and D, which applies each of the operators in Π_C once in the order ω_C:

- *either it computes a fixed-point of Π_C, which must be the greatest fixed-point $\top$, by unicity of the gfp and by contractance of the operators in Π_C,*
- *or, it is possible to narrow further the domains of the variables by applying one of the operators in Π_C. Let $\pi_1 \colon \beta_1 \leftarrow f_1(\beta_1, \ldots, \beta_k)$ be this operator. Consider the case where π_1 is an operator applied during the bottom-up sweep (the case where it is an operator applied during the top-down sweep may be handled in the same way): Let $D_1^{(0)}, \ldots, D_k^{(0)}$ be the domains of $\beta_1, \ldots, \beta_k$ just before applying π_1 for the first time; let $D_1^{(1)} = f_1(D_1^{(0)}, D_2^{(0)}, \ldots, D_k^{(0)})$; let $D_1^{(2)} \subseteq D_1^{(1)}$ be the new domain computed for β_1 during the top-down sweep, and $D_i^{(1)} = f_i(D_1^{(2)}, D_2^{(0)}, \ldots, D_k^{(0)})$ for $i \in \{2, \ldots, k\}$ the domains for $\beta_2, \ldots, \beta_k$ obtained by back-propagation of $D_1^{(2)}$. Let us show that $D_1^{(3)} = f_1(D_1^{(2)}, D_2^{(1)}, \ldots, D_k^{(1)})$ is equal to $D_1^{(2)}$, which will contradict the hypothesis that we have not reached a fixed-point: by definition of the UC3revise f_i operators used, we have: $D_1^{(1)} = \{b_1 \in D_1^{(0)} \mid \exists a_1 \in D_1^{(0)} \ldots \exists a_k \in D_k^{(0)} \colon c(a_1, \ldots, a_k) \wedge b_1 \in \mathrm{hull}(\{a_1\})\}$, $D_i^{(1)} = \{b_i \in D_i^{(0)} \mid \exists a_1 \in D_1^{(2)} \exists a_2 \in D_2^{(0)} \ldots \exists a_k \in D_k^{(0)} \colon c(a_1, \ldots, a_k) \wedge b_i \in \mathrm{hull}(\{a_i\})\}$ for $i \in \{2, \ldots, k\}$, and $D_1^{(3)} = \{b_1 \in D_1^{(2)} \mid \exists a_1 \in D_1^{(2)} \exists a_2 \in D_2^{(1)} \ldots \exists a_k \in D_k^{(1)} \colon c(a_1, \ldots, a_k) \wedge b_1 \in \mathrm{hull}(\{a_1\})\}$. Thus, for any $b_1 \in D_1^{(2)}$, we have $\exists a_1 \in D_1^{(2)} \exists a_2 \in D_2^{(0)} \ldots \exists a_k \in D_k^{(0)} \colon c(a_1, \ldots, a_k) \wedge b_i \in \mathrm{hull}(\{a_i\})$ since $D_1^{(2)} \subseteq D_1^{(1)}$ and the domains of $\beta_2, \ldots, \beta_k$ have not changed between the bottom-up and the top-down sweeps, the constraint being admissible (with a tree representation). However, by definition of $D_2^{(1)}, \ldots, D_k^{(1)}$, we have $a_2 \in D_2^{(1)}, \ldots, a_k \in D_k^{(1)}$ since $a_2 \in \mathrm{hull}(\{a_2\}), \ldots, a_k \in \mathrm{hull}(\{a_k\})$. Consequently, $b_1 \in D_1^{(3)}$, and then $D_1^{(2)} \subseteq D_1^{(3)}$. Since $D_1^{(3)} \subseteq D_1^{(2)}$ by contractance of the operators, we conclude that $D_1^{(2)} = D_1^{(3)}$.* $\square$

Note. Contrary to Benhamou *et al.* statement [1], the HC4-revise algorithm does not enforce hull consistency on an admissible constraint as the following example shows: take $c \colon 1/x = y$ with $x \in [-1, 1]$ and $y \in [0, +\infty]$. Applying HC4revise on c leads to $x \in [0, 1]$, and $y \in [0, +\infty]$, which are clearly not hull consistent domains (y should be narrowed down to $[1, +\infty]$).

4. ASSESSING THE IMPACT OF CONSTRAINT DECOMPOSITION

We present the results of both HC4 and HC3 on four standard benchmarks from the interval constraint community. They were chosen so as to be scalable at will and to exhibit various behaviors of the algorithms. As a side note, it is important to remember that these algorithms are often outperformed by other algorithms. Their study is still pertinent, however, since they serve as basic procedures in these more efficient algorithms.

It is important to note also that, originally, none of these problems is admissible. In order to show the impact of admissibility, we have factorized the constraints of one of them.

All the problems have been solved on an AMD Athlon 900 MHz under Linux, using a C++ interval constraint library written for our tests based on the gaol[2] interval arithmetic library. In order to avoid any interference, no optimization (e.g., *improvement factor*) was used.

[1]Alternatively, the constraint $\alpha_1 = \alpha_3$ could be replaced by the equivalent one $\alpha_1 - \alpha_3 = \alpha_0$, with α_0 constrained to be equal to 0.

[2]Interval C++ library available at `http://sf.net/projects/gaol/`

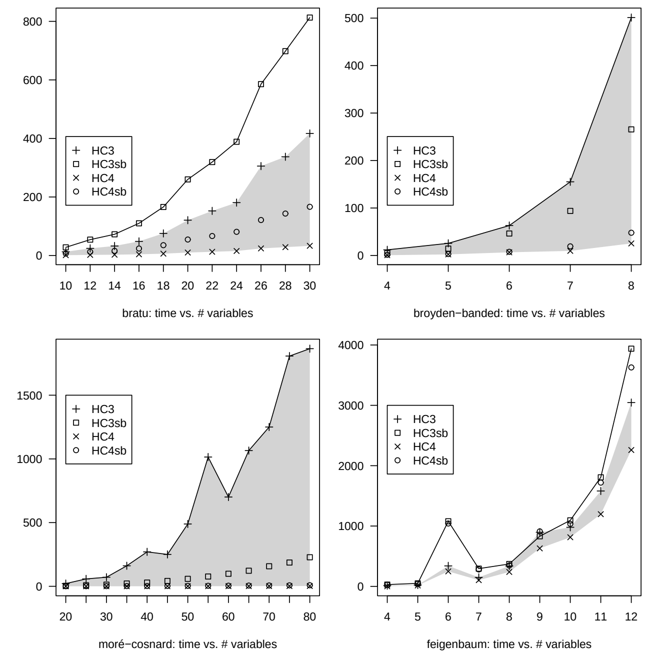


Figure 2: Solving times in seconds

For each benchmark, four different methods have been used:

- `HC3`, which enforces bounds consistency on the decomposed system: the propagation algorithm is APX-CONSISTENCY and the Revise method is implemented with HC3revise;
- `HC3sb`, which uses *S-boxes* [7], that is, each user constraint C is decomposed into a *separate* set of primitives and gives rise to a Revise procedure RB_C that enforces bounds consistency on this set by using HC3revise procedures for each primitive, and propagating the modifications with Alg. APX-CONSISTENCY. All the RB_C methods for the constraints in the user system are then handled themselves with Alg. APX-CONSISTENCY. This propagation scheme forces consistency to be *enforced locally* for each user constraint before reconsidering the others;
- `HC4`, which enforces directional bounds consistency on each user constraint using HC4revise, and which uses Alg. APX-CONSISTENCY for the propagation over the constraints in the system;
- `HC4sb`, which uses one S-box per user constraint. As a consequence, HC4revise is called as many times as necessary to reach a fixed-point for any non-admissible constraint before considering other constraints.

Each graphics provided (see Fig. 2) displays the computation time in seconds required to find all solutions up to an accuracy of 10^{-8} (difference between the lower and upper bounds of the intervals) for each method.

The `bratu` constraint system modelizes a problem in combustion theory. It is a square, sparse and quasi-linear system of $n+2$ equations (with $i, k \in \{1, \ldots, n\}$):

$$\begin{cases} \forall k\colon x_{k-1} - 2x_k + x_{k+1} + \exp(x_k)/(n+1)^2 = 0, \\ x_0 = x_{n+1} = 0, \quad \forall i\colon x_i \in [-10^8, 10^8] \end{cases}$$

The largest number of nodes per constraint is independent of the size of the problem and is equal to 12 in our implementation.

As already reported by Benhamou et al. [1], HC4 appears more efficient than HC3 to solve all instances of the problem, and its advantage grows with their size. Localizing the propagation does not seem a good strategy here, since HC3sb and HC4sb both perform poorly in terms of the number of projections computed[3]. Interestingly enough, HC4sb is faster than HC3 while it requires more projections. The first reason for this discrepancy that comes to mind is that the "anarchic" propagation in HC3 has a cost much higher in terms of management of the set of constraints to reinvoke than the controlled propagation achieved with HC4sb (see below for another analysis).

The `broyden-banded` problem is very difficult to solve with HC3, so that we could only consider small instances of it (with $k \in \{1, \ldots, n\}$):

$$\begin{aligned} \forall k\colon\ & x_k(2 + 5x_k^2) + 1 - \textstyle\sum_{j \in J_k} x_j(1 + x_j) = 0 \\ & J_k = \{j \mid j \neq k\ \wedge \max(1, i-5) \leqslant j \leqslant \min(n, i+1)\}, \\ & x_k \in [-10^8, +10^8] \end{aligned}$$

Contrary to `bratu`, the number of nodes in the constraints is not independent of the size of the problem. It follows however a simple pattern and it is bounded from below by 16 and from above by 46.

As with `bratu`, the efficiency of HC4 compared to HC3 is striking, even on the small number of instances considered. Note that, here, HC3sb is better than HC3. On the other hand, HC4 is still better than HC4sb.

The `moré-cosnard` problem is a nonlinear system obtained from the discretization of a nonlinear integral equation (with $k \in \{1, \ldots, n\}$):

$$\forall k\colon \begin{cases} x_k \in [-10^8, 0], \\ x_k + \frac{1}{2}[(1-t_k)\sum_{j=1}^{k} t_j(x_j + t_j + 1)^3 \\ \qquad + t_k \sum_{j=k+1}^{n} (1 - t_j)(x_j + t_j + 1)^2] = 0 \end{cases}$$

The largest number of nodes per constraint grows linearly with the number of variables in the problem.

HC4 allows to solve this problem up to 1000 times faster than HC3 on the instances we tested. An original aspect of this benchmark is that localizing the propagation by using S-boxes seems a good strategy: HC3sb solves all instances almost as fast as HC4 (see the analysis in the next section). Note that, once again, though the number of projections required for HC3sb is almost equal to the one for HC4, there is still a sizable difference in solving time, which again might be explained by higher propagation costs in HC3sb.

Lastly, the `Feigenbaum` problem is a quadratic system:

$$\begin{cases} \forall k \in \{1, \ldots, n\}\colon x_k \in [0, 100], \\ \forall k \in \{1, \ldots, n-1\}\colon\ -3.84x_k^2 + 3.84x_k - x_{k+1} = 0, \\ -3.84x_n^2 + 3.84x_n - x_1 = 0 \end{cases}$$

The largest number of nodes per constraint is independent of the size of the problem. It is equal to 10 in our implementation.

[3]Due to lack of space, all graphics corresponding to the number of projections have been omitted.

The advantage of HC4 over HC3 is not so striking on this problem. HC4sb and HC3sb do not fare well either, at least if we consider the computation time.

Parenthetically, the equations in the `feigenbaum` problem can easily be factorized so that the resulting problem is only composed of admissible constraints. Due to lack of space, the corresponding graphics is not presented here; however, we note that the solving time is reduced by a factor of more than 500 compared to the original version.

5. DISCUSSION

To sum up what has been observed in Section 4, it appears that it is more efficient to deal locally with the modification of a domain induced by some primitive by *first reinvoking the other primitives coming from the decomposition of the same user constraint* in problems with large constraints like `moré-cosnard`, while the opposite is true with systems of small constraints (even for systems with many constraints) such as `feigenbaum` or `bratu`. An intuitive understanding of that is that the information just obtained by a reduction is spread and lost in a large network of primitives while it could be efficiently used locally to compute projections on a user constraint.

Figure 3 relates the ratios—normalized so that they all lie in the range 0.0–1.0—of the number of projections required by HC3 and HC4, and by HC3sb and HC4sb to the number of nodes in a constraint. As one may see, the ratio is roughly constant for HC3 and HC4 when the number of nodes is independent of the size of the problem, while it increases sharply when the number of nodes increases with the size of the problem (e.g., `moré-cosnard`). On the other hand, the ratio between HC3sb and HC4sb stays constant for *all* problems, a fact particularly striking with `moré-cosnard`. This is a solid evidence that *localization of the information as obtained from using HC4 (or, to a lesser extent, HC3sb), is a winning strategy the larger the constraints in a problem are.*

The explanation for the loss of performances induced by the decomposition of user constraints is then twofold: for large systems made of small constraints, it is mainly due to the multiplication of constraints to consider, as it was conjectured by previous authors; for large systems with large constraints, an additional cause is the immediate spreading of information gained from one user constraint c to the primitives stemming from other user constraints, where it is not relevant, while deferring the reconsideration of more interesting primitive constraints stemming from c (*irrelevant-spreading effect*).

It is interesting to observe that HC4 is always more efficient than HC4sb on all the benchmarks considered. This is consistent with facts long known by numerical analysts: we show in a paper to come that HC4 may be considered as a free-steering nonlinear Gauss-Seidel procedure where the inner iteration is obtained as in the linear case. For this class of methods, it has been proved experimentally in the past that it is counterproductive to try to reach some fixed-point in the inner iteration.

For the interested reader, all the data used to prepare the figures in Section 4 and many more are available in tabulated text format at `http://www.sciences.univ-nantes.fr/info/perso/permanents/goualard/dbc-data/`.

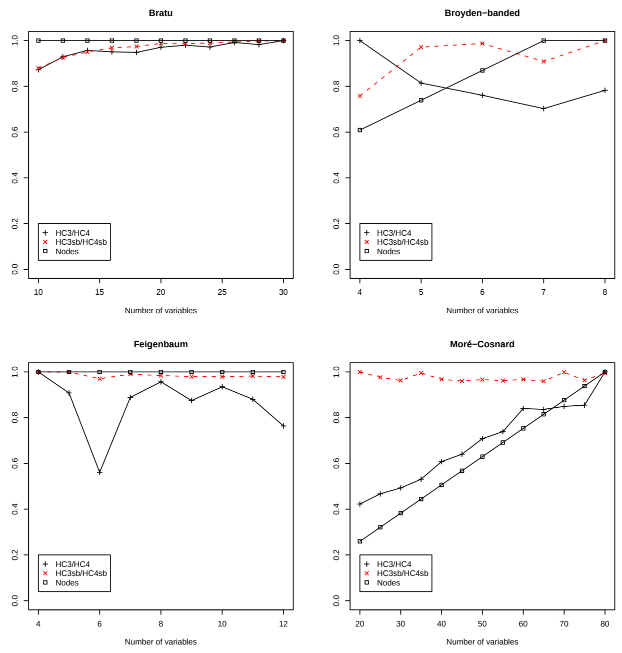


Figure 3: Impact of the number of nodes per constraint on the number of projections

Acknowledgements

The contribution of an anonymous reviewer in spotting a flaw in Prop. 1 as stated in an earlier version of this paper is gratefully acknowledged.

6. REFERENCES

[1] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Procs. of ICLP '99*, pages 230–244. The MIT Press, 1999.

[2] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *JLP*, 32(1):1–24, 1997.

[3] E. Davis. Constraint propagation with interval labels. *A.I.*, 32:281–331, 1987.

[4] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 1st edition, 2003.

[5] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *A.I.*, 34:1–38, 1987.

[6] E. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982.

[7] F. Goualard and F. Benhamou. A visualization tool for constraint program debugging. In *Procs. of ASE '99*, pages 110–117. IEEE Computer Society, 1999.

[8] A. Mackworth. Consistency in networks of relations. *A.I.*, 1(8):99–118, 1977.

[9] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[10] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[11] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, chapter 2, pages 19–91. McGraw-Hill, 1975.