ELSEVIER

Artificial Intelligence 139 (2002) 109–132

Artificial Intelligence

www.elsevier.com/locate/artint

Accelerating filtering techniques for numeric CSPs

Yahia Lebbah [a], Olivier Lhomme [b,*]

[a] *Département Informatique, Faculté des Sciences, Université d'Oran Es-Senia, B.P. 1524, El-M'Naouar Oran, Algeria*

[b] *ILOG, 1681 route des Dolines, F-06560 Valbonne, France*

Received 13 July 2001

Abstract

Search algorithms for solving Numeric CSPs (Constraint Satisfaction Problems) make an extensive use of filtering techniques. In this paper[1] we show how those filtering techniques can be accelerated by discovering and exploiting some regularities during the filtering process. Two kinds of regularities are discussed, cyclic phenomena in the propagation queue and numeric regularities of the domains of the variables. We also present in this paper an attempt to unify numeric CSPs solving methods from two distinct communities, that of CSP in artificial intelligence, and that of interval analysis. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Numeric constraint satisfaction problem; Filtering techniques; Propagation; Pruning; Acceleration methods; Nonlinear equations; Interval arithmetic; Interval analysis; Strong consistency; Extrapolation methods

1. Introduction

In several fields of human activity, like engineering, science or business, people are able to express their problems as constraint problems. The CSP (Constraint Satisfaction Problem) schema is an abstract framework to study algorithms for solving such constraint problems. A CSP is defined by a set of variables, each with an associated domain of possible values and a set of constraints on the variables. This paper deals more specifically with CSPs where the constraints are numeric nonlinear relations and where the domains are continuous domains (numeric CSPs).

* Corresponding author.
E-mail addresses: ylebbah@univ-oran.dz (Y. Lebbah), olhomme@ilog.fr (O. Lhomme).

[1] This paper is an extended version of [31].

0004-3702/02/$ – see front matter
PII: S0004-3702(02)00194-7

In general, numeric CSPs cannot be tackled with computer algebra systems: there is no algorithm for general nonlinear constraint systems. And most numeric algorithms cannot guarantee completeness: some solutions may be missed, a global optimum may never be found, and, sometimes a numeric algorithm even does not converge at all. The only numeric algorithms that can guarantee completeness—even when floating-point computations are used—are coming either from the interval analysis community or from the AI community (CSP). Unfortunately, those *safe* constraint-solving algorithms are often less efficient than non-safe numeric methods, and the challenge is to improve their efficiency.

The safe constraint-solving algorithms are typically a search-tree exploration where a filtering technique is applied at each node. Improvement in efficiency is possible by finding the best compromise between a filtering technique that achieves a strong pruning at a high computational cost and another one that achieves less pruning at a lower computational cost. And thus, a lot of filtering techniques have been developed. Some filtering techniques take their roots from numerical analysis: the main filtering technique used in interval analysis [37] is an interval variation of Newton iterations. (See [24,28] for an overview of such methods.) Other filtering techniques originate from artificial intelligence: the basic filtering technique is a kind of *arc*-consistency filtering [36] adapted to numeric CSPs [17, 26,32]. Higher-order consistencies similar to k-consistency [21] have also been defined for numeric CSPs [25,32]. Another technique from artificial intelligence [19,20] is to merge the constraints concerning the same variables, giving one "total" constraint (thanks to numerical analysis techniques) and to perform *arc*-consistency on the total constraints. Finally, [6,45] aim at expressing interval analysis pruning as partial consistencies, bridging the gap between the two families of filtering techniques.

All the above works address the issue of finding a new partial consistency property that can be computed by an associated filtering algorithm with a good efficiency (with respect to the domain reductions performed). Another direction, in the search of efficient safe algorithms, is to try to optimize the computation of already existing consistency techniques. Indeed, the aim of this paper is to study general methods for accelerating consistency techniques. The main idea is to identify some kinds of regularity in the dynamic behavior of a filtering algorithm, and then to exploit those regularities. A first kind of regularities we exploit is the existence of cyclic phenomena in the propagation queue of a filtering algorithm. A second kind of regularities is a numeric regularity: when the filtering process converges asymptotically, its fixed point often can be extrapolated. As we will see in the paper, such ideas, although quite general, may lead to drastic improvements in efficiency for solving numeric CSPs. The paper focus on numeric continuous problems, but the ideas are more general and may be of interest also for mixed discrete and continuous problems, or even for pure discrete problems.

The paper is organized in two main parts. The first part (Section 2) presents an overview of numeric CSPs; artificial intelligence works and interval analysis works are presented through a unifying framework. The second part consists of the next two sections, and presents the contribution of the paper. Section 3 introduces the concept of reliable transformation, and presents two reliable transformations that exploit two kinds of regularities occurring during the filtering process: cyclic phenomena in the propagation queue and numeric regularities of the domains of the variables. Section 4 discusses related works.

2. Numeric CSPs

This section presents numeric CSPs in a slightly non-standard form, which will be convenient for our purposes, and will unify works from interval analysis and constraint satisfaction communities.

A numeric CSP is a triplet $\langle \mathcal{X}, D, C \rangle$ where:

- $\mathcal{X}$ is a set of n variables $x_1, \ldots, x_n$.
- $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ denotes a vector of domains. The ith component of $\mathcal{D}$, D_i, is the domain containing all acceptable values for x_i.
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ denotes a set of numeric constraints. $var(C_j)$ denotes the variables appearing in C_j.

This paper focuses on CSPs where the domains are intervals: $\mathcal{D} \in \mathbb{I}(\mathbb{R})^n$ where $\mathbb{I}(\mathbb{R}) = \{[a, b] \mid a, b \in \mathbb{R} \cup \{-\infty, +\infty\}\}$.

The following notation is used throughout the paper. An interval $[a, b]$ such that $a > b$ is an empty interval. A vector of domains $\mathcal{D}$ such that a component D_i is an empty interval will be denoted by $\emptyset$. The lower bound, the upper bound and the midpoint of an interval D_i (respectively interval vector $\mathcal{D}$) are respectively denoted by $\underline{D_i}$, $\overline{D_i}$, and $m(D_i)$ (respectively $\underline{\mathcal{D}}$, $\overline{\mathcal{D}}$, and $m(\mathcal{D})$). The lower bound, the upper bound, the midpoint, the inclusion relation, the union operator and the intersection operator are defined over interval vectors; they have to be interpreted componentwise. For instance $\underline{\mathcal{D}}$ means $\langle \underline{D_1}, \ldots, \underline{D_n} \rangle$; $\mathcal{D}' \subset \mathcal{D}$ means $D'_i \subset D_i$ for all $i \in 1, \ldots, n$; $\mathcal{D}' \cap \mathcal{D}''$ means $\langle D'_1 \cap D''_1, \ldots, D'_n \cap D''_n \rangle$.

A k-ary constraint $C_j(x_{j_1}, \ldots, x_{j_k})$ denotes a k-ary relation over the real numbers, that is, a subset of $\mathbb{R}^k$.

2.1. Approximation of projection functions

The algorithms used over numeric CSPs typically work by narrowing domains and need to compute the projection—denoted $\Pi_{C_j, x_i}(\mathcal{D})$ or also $\Pi_{j,i}(\mathcal{D})$—of a constraint $C_j(x_{j_1}, \ldots, x_{j_k})$ over the variable x_i in the space delimited by $D_{j_1} \times \cdots \times D_{j_k}$. The projection $\Pi_{j,i}(\mathcal{D})$ is defined as follows.

- If $x_i \notin var(C_j)$, $\Pi_{j,i}(\mathcal{D}) = D_i$.
- If $x_i \in var(C_j)$ the projection is defined by the set of all elements $d_i \in D_i$ such that we can find elements $d_{j_1}, \ldots, d_{i-1}, d_{i+1}, \ldots, d_{j_k}$ for the $k-1$ remaining variables of $var(C_j)$ with $\langle d_{j_1}, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_{j_k} \rangle \in C_j$. Formally:

$$\begin{aligned} \Pi_{j,i}(\mathcal{D}) = \big\{ d_i \mid d_i \in D_i, \exists d_{j_1}, \ldots, d_{i-1}, d_{i+1}, \ldots, d_{j_k} \\ (d_{j_1} \in D_{j_1}, \ldots, d_{i-1} \in D_{i-1}, d_{i+1} \in D_{i+1}, \ldots, d_{j_k} \in D_{j_k}, \\ \langle d_{j_1}, \ldots, d_i, \ldots, d_{j_k} \rangle \in C_j) \big\}. \end{aligned} \tag{1}$$

Usually, such a projection cannot be computed exactly due to several reasons, such as: (1) the machine numbers are floating point numbers and not real numbers so round-off

errors occur; (2) the projection may not be representable as floating-point numbers; (3) the computations needed to have a close approximation of the projection of only one given constraint may be very expensive; (4) the projection may be discontinuous whereas it is much easier to handle only closed intervals for the domains of the variables.

Thus, what is usually done is that the projection of a constraint over a variable is approximated. Let $\pi_{C_j,x_i}(\mathcal{D})$ or also $\pi_{j,i}(\mathcal{D})$ denote such an approximation. In order to guarantee that all solutions of a numeric CSP can be found, a solving algorithm that uses $\pi_{j,i}(\mathcal{D})$ needs that $\pi_{j,i}(\mathcal{D})$ includes the exact projection. We will also assume in the rest of the paper that $\pi_{j,i}(\mathcal{D})$ satisfies a contractance property. Thus we have:

$$\Pi_{j,i}(\mathcal{D}) \subseteq \pi_{j,i}(\mathcal{D}) \subseteq \mathcal{D}.$$

$\pi_{j,i}(\mathcal{D})$ hides all the problems seen above. In particular, it allows us not to go into the details of the relationships between floating point and real numbers (see for example [2] for those relationships) and to consider only real numbers. It only remains to build such a $\pi_{j,i}$. Interval analysis [37] makes it possible.

2.1.1. Interval arithmetic

Interval arithmetic [37], on which interval analysis is built, is an extension of real arithmetic. It defines the arithmetic functions $\{+, -, *, /\}$ over the intervals with simple set extension semantics.

Notation. To present interval arithmetic, we will use the following convention to help the reading: x, y will denote real variables or vectors of real variables and X, Y will denote interval variables or vectors of interval variables. Distinction between a scalar variable and a vector of variables will be clear from the context.

With this notation, an arithmetic function $\odot \in \{+, -, *, /\}$ over the intervals is defined by:

$$X \odot Y = \{x \odot y \mid x \in X, y \in X\}.$$

Thanks to the monotonicity property of arithmetic operators $\odot$, $X \odot Y$ can be computed by considering the bounds of the intervals only. Let $X, Y \in \mathbb{I}(\mathbb{R})$, $X = [\underline{X}, \overline{X}]$, and $Y = [\underline{Y}, \overline{Y}]$, the arithmetic operators are computed on intervals as follows:

$$\begin{aligned}
X + Y &= [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}],\\
X - Y &= [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}],\\
X * Y &= \big[\min(\underline{X} * \underline{Y}, \underline{X} * \overline{Y}, \overline{X} * \underline{Y}, \overline{X} * \overline{Y}),\ \max(\underline{X} * \underline{Y}, \underline{X} * \overline{Y}, \overline{X} * \underline{Y}, \overline{X} * \overline{Y})\big],\\
X / Y &= \big[\min(\underline{X}/\underline{Y}, \underline{X}/\overline{Y}, \overline{X}/\underline{Y}, \overline{X}/\overline{Y}),\\
&\qquad \max(\underline{X}/\underline{Y}, \underline{X}/\overline{Y}, \overline{X}/\underline{Y}, \overline{X}/\overline{Y})\big] \quad \textbf{if } 0 \notin Y.
\end{aligned}$$

2.1.2. Interval extension of a real function

For an arbitrary function over the real numbers, it is not possible in general to compute the exact enclosure of the range of the function [29]. The concept of *interval extension* has been introduced by Moore: the interval extension of a function is an interval function

that computes outer approximations on the range of the function over a domain. Different interval extensions exist. Let f be a function over the real numbers defined over the variables $x_1, \ldots, x_n$, the following interval extensions are frequently used:

- *nat*(f): the natural interval extension of a real function f is defined by replacing each real operator by its interval counterpart. It is easy to see that *nat*(f) always contains the range of f, and is thus an interval extension.

  Example 1 (*Natural extension of* $x_1^2 + x_2^2 - 2$ *and* $x_1^2 + x_2$).

  The natural extension of $x_1^2 + x_2^2 - 2$ is $X_1^2 + X_2^2 - 2$.
  The natural extension of $x_1^2 + x_2$ is $X_1^2 + X_2$.

- *tay*(f): the Taylor interval extension of a real function f, over the interval vector X, is defined by the natural extension of a first-order Taylor development of f [42]:

$$f\big(m(X)\big) + \sum_{i=1}^{n} nat\left(\frac{\partial f}{\partial x_i}\right)(X) * \big(X_i - m(X_i)\big). \tag{2}$$

  The intuition why *tay*(f) is an interval extension is given in a footnote.[2]

  Example 2 (*Taylor extension of* $x - x^2$). Let

$$f(x) = x - x^2, \qquad f'(x) = 1 - 2x,$$
$$nat(f')(X) = 1 - 2X.$$

  The Taylor extension of $f(x)$ is:

$$tay(f)(X) = \big(c - c^2\big) + (1 - 2X)(X - c) \quad \text{where } c = m(X).$$

The Taylor extension gives generally a better enclosure than the natural extension on small intervals.[3] Nevertheless, in general neither *nat*(f) nor *tay*(f) give the exact range of f. For example, let $f(x) = 1 - x + x^2$, and $X = [0, 2]$, we have:

$$tay(f)([0, 1]) = [-1, 3], \qquad nat(f)([0, 1]) = [-1, 5],$$

whereas the range of f over $X = [0, 1]$ is $[3/4, 3]$.

[2] The Taylor interval extension comes from a direct application of the mean value theorem: Let f be a real function defined over $[a, b]$, let f be continuous and with a continuous derivative over $[a, b]$, let x_1, x_2 be two points in $[a, b]$. Then, there exists ξ between x_1 and x_2 such that $f(x_2) = f(x_1) + f'(\xi) * (x_2 - x_1)$.

ξ is unknown, but what can be done is to replace it by an interval that contains it, and to evaluate the natural extension of the resulting expression. Thus we know that $f(x_2) \in f(x_1) + f'([a, b]) * (x_2 - x_1)$. As this is true for every x_1 and x_2 in $[a, b]$, we can replace x_1 by the midpoint of $[a, b]$ and x_2 by an interval that contains it. This leads to $f([a, b]) \subseteq f(m([a, b])) + f'([a, b]) * ([a, b] - m([a, b]))$.

(2) is the generalization for vectors of the above result.

[3] The Taylor extension has a quadratic convergence, whereas the natural extension has a linear convergence; see for example [42].

2.1.3. Solution function of a constraint

To compute the projection $\pi_{j,i}(\mathcal{D})$ of the constraint C_j on the variable x_i, we need to introduce the concept of *solution function* that expresses the variable x_i in terms of the other variables of the constraint. For example, for the constraint $x + y = z$, the solution functions are: $f_x = z - y$, $f_y = z - x$, $f_z = x + y$.

Assume a solution function is known that expresses the variable x_i in terms of the other variables of the constraint. Thus an approximation of the projection of the constraint over x_i given a domain $\mathcal{D}$ can be computed thanks to any interval extension of this solution function. Thus we have a way to compute $\pi_{j,i}(\mathcal{D})$.

Nevertheless, for complex constraints, there may not exist such an analytic solution function; for example, consider $x + \log(x) = 0$. The interest of numeric methods as presented in this paper is precisely for those constraints that cannot be solved algebraically. Three main approaches have been proposed:

- The first one exploits the fact that analytic functions always exist when the variable to express in terms of the others appears only one time in the constraint. This approach simply considers that each occurrence of a variable is a different new variable. In the previous example this would give: $x_{(1)} + \log(x_{(2)}) = 0$. That way, it is trivial to compute a solution function: it suffices to know the inverse of basic operators. In our example, we obtain $f_{x_{(1)}} = -\log(x_{(2)})$ and $f_{x_{(2)}} = \exp^{-x_{(1)}}$.
  An approximation of the projection of the constraint over x_i can be computed by intersecting the natural interval extensions of the solution functions for all occurrences of x_i in C_j. For the last example, we could take $\pi_{x+\log(x)=0,x}(X) = -\log(X) \cap \exp^{-X}$. Projection functions obtained by this way will be called π^{nat} in this paper.
- The second idea uses the Taylor extension to transform the constraint into an interval linear constraint. The nonlinear equation $f(X) = 0$ becomes

$$f(c) + \sum_{i=1}^{n} nat\left(\frac{\partial f}{\partial x_i}\right)(X) * (X_i - c_i) = 0,$$

  where $c = m(X)$. Now consider that the derivatives are evaluated over a box $\mathcal{D}$ that contains X. $\mathcal{D}$ is considered as constant, and let $c = m(\mathcal{D})$. The equation becomes:

$$f(c) + \sum_{i=1}^{n} nat\left(\frac{\partial f}{\partial x_i}\right)(\mathcal{D}) * (X_i - c_i) = 0.$$

  This is an interval linear equation in X, which does not contain multiple occurrences. The solution functions could be extracted easily. But, instead of computing the solution functions of the constraint without taking into account the other constraints, we may prefer to group together several linear equations in a squared system. Solving the squared interval linear system allows much more precise approximations of projections to be computed. (See the following section.) Projection functions obtained by this way are called π^{Tay}. For example, consider the constraint $x + \log(x) = 0$; by using the Taylor form on the box $\mathcal{D}$, we obtain the following interval linear equation

$$c + \log(c) + (1 + 1/\mathcal{D})(X - c) = 0$$

that is:

$$AX + B = 0,$$

where $A = 1 + 1/\mathcal{D}$ and $B = \log(c) - c/\mathcal{D}$. The unique solution function of this 1-dimensional linear equation is straightforward: $X = -B/A$.

- A third approach [6] does not use any analytical solution function. Instead, it transforms the constraint $C_j(x_{j_1}, \ldots, x_{j_k})$ into k mono-variable constraints $C_{j,l}$, $l = 1, \ldots, k$. The mono-variable constraint $C_{j,l}$ on variable x_{j_l} is obtained by substituting their intervals for the other variables. The projection π_{j,j_l} is computed thanks to $C_{j,l}$. The smallest zero of $C_{j,l}$ in the interval under consideration is a lower bound for the projection of C_j over x_{j_l}. And the greatest zero of $C_{j,l}$ is an upper bound for that projection. Hence, an interval with those two zeros as bounds gives an approximation of the projection. Projection functions computed in that way are called π^{box}.
  In [6], the two extremal zeros of $C_{j,l}$ are found by a mono-variable version of the interval Newton method.[4]

Another problem is that the inverse of a nonmonotonic function is not a function over the intervals. For example the range of the inverse of the function $f(x) = x^2$ for an interval Y is the union of intervals $(-\sqrt{Y}) \cup (\sqrt{Y})$. It is possible to extend interval arithmetic in order to handle unions of intervals. A few systems have taken this approach [26,44]. Nevertheless, this approach may lead to a highly increasing number of intervals. The two other approaches more commonly used consist of computing the smallest interval encompassing a union of intervals: $\bigcup_{0 \leqslant i \leqslant n}([a_i, b_i]) = [\min_{0 \leqslant i \leqslant n}(a_i), \max_{0 \leqslant i \leqslant n}(b_i)]$, or to split the problem in several sub-problems in which only intervals appear.

2.2. Filtering algorithm as fixed point algorithms

A filtering algorithm can generally be seen as a fixed point algorithm. In the following, an abstraction of filtering algorithms will be used: the sequence $\{\mathcal{D}_k\}$ of domains generated by the iterative application of an operator $Op : \mathbb{I}(\mathbb{R})^n \longrightarrow \mathbb{I}(\mathbb{R})^n$ (see Fig. 1).

The operator Op of a filtering algorithm generally satisfies the following three properties:

- $Op(\mathcal{D}) \subseteq \mathcal{D}$ (contractance).
- Op is conservative; that is, it cannot remove solutions.
- $\mathcal{D}' \subseteq \mathcal{D} \Rightarrow Op(\mathcal{D}') \subseteq Op(\mathcal{D})$ (monotonicity).

Under those conditions, the limit of the sequence $\{\mathcal{D}_k\}$, which corresponds to the greatest fixed point of the operator Op, exists and is called a *closure*. We denote it by $\Phi_{Op}(\mathcal{D})$. A fixed point for Op may be characterized by a property *lc*-consistency, called a local consistency, and alternatively $\Phi_{Op}(\mathcal{D})$ will be denoted by $\Phi_{lc}(\mathcal{D})$. The algorithm achieving filtering by *lc*-consistency is denoted *lc*-filtering. A CSP is said to be *lc*-satisfiable if *lc*-filtering of this CSP does not produce an empty domain.

[4] The general (multi-variable) interval Newton method is briefly presented in Section 2.3.

$$\mathcal{D}_k = \begin{cases} \mathcal{D} & \text{if } k = 0 \\ Op(\mathcal{D}_{k-1}) & \text{if } k > 0 \end{cases}$$

Fig. 1. Filtering algorithms as fixed point algorithms.

Consistencies used in numeric CSPs solvers can be categorized in two main classes: *arc-consistency*-like consistencies and strong consistencies.

2.3. Arc-consistency-like consistencies

Most of the numeric CSP systems (e.g., BNR-prolog [40], Interlog [13,16], CLP(BNR) [5], PrologIV [15], UniCalc [3], Ilog Solver [27] and Numerica [46] compute an approximation of *arc*-consistency [36] which will be named $2B$-consistency in this paper.[5] $2B$-consistency states a local property on a constraint and on the bounds of the domains of its variables (B of $2B$-consistency stands for *bound*). Roughly speaking, a constraint C_j is $2B$-consistent if for any variable x_i in $var(C_j)$ the bounds $\underline{D_i}$ and $\overline{D_i}$ have a support in the domains of all other variables of C_j (w.r.t. the approximation given by π). $2B$-consistency can be defined in our notation as:

Definition 1 ($2B$-*consistency*). A CSP $\langle \mathcal{X}, D, C \rangle$ is $2B$-consistent if and only if $\forall C_j, \forall x_i (C_j \in \mathcal{C} \wedge x_i \in var(C_j) \Rightarrow \pi_{j,i}(\mathcal{D}) = D_i)$.

A filtering algorithm that achieves $2B$-consistency can be derived from Fig. 1 by instantiating Op as in Operator 1. Note the operator Op_{2B} applies on the *same* vector $\mathcal{D}$ all the $\pi_{j,i}(\mathcal{D})$ operators.

Operator 1 ($2B$-*consistency filtering operator*).

$$Op_{2B}(\mathcal{D}) = \bigcap_{C_j \in \mathcal{C}} \langle \pi_{j,1}(\mathcal{D}), \ldots, \pi_{j,n}(\mathcal{D}) \rangle.$$

Fig. 2 shows how projection functions are used by a $2B$-consistency filtering algorithm to reduce the domains of the variables.

Depending on the projection functions used, we obtain different $2B$-filtering algorithms.

Op_{nat} The operator Op_{nat} will denote Op_{2B} with π^{nat}. It abstracts the filtering algorithm presented in [5,17,32]. There are two main differences between our abstraction and the implementations.

(1) In classic implementations, projection functions are applied sequentially and not all on the same domain. In the abstraction (and in our non-classic

[5] We have a lot of freedom to choose $\pi_{j,i}(\mathcal{D})$, so the definition of $2B$-consistency here abstracts both $2B$-consistency in [32] and *box*-consistency in [6].

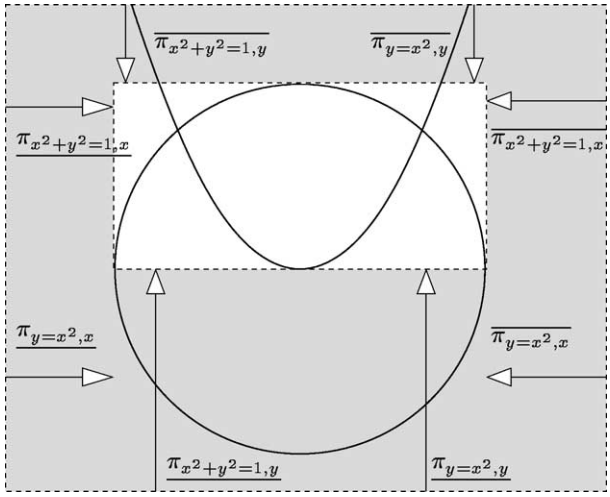

Fig. 2. $2B$-filtering on the constraint system $\{x^2 + y^2 = 1, y = x^2\}$.

implementations) they are applied on the same domain. This has the drawback of increasing the upper bound of the complexity, but has the advantage of generating much more "regular" sequences of domains. (See Section 3.2.)

(2) Implementations always applied an AC3-like optimization [36]. It consists of applying at each iteration only those projection functions that may reduce a domain: only the projection functions that have in their parameters a variable whose domain has changed are applied. For the sake of simplicity, AC3-like optimization does not appear explicitly in this algorithm schema.

Op_{box} This operator denotes Op_{2B} that uses π^{box}. It abstracts the filtering algorithm presented in [6,45]. Differences with our abstraction are the same as above.

Op_{Tay} This operator denotes Op_{2B} that uses π^{Tay}. It abstracts the interval Newton method [2,37]. The interval Newton method controls in a precise way the order in which projection functions are computed. It is used for solving squared nonlinear equation systems such as $\mathcal{C} = \{f_1(x_1, \ldots, x_n) = 0, \ldots, f_n(x_1, \ldots, x_n) = 0\}$. The interval Newton method replaces the solving of the nonlinear squared system by the solving of a sequence of interval linear squared systems. Each linear system is obtained by evaluating the interval Jacobi matrix over the current domains, and by considering the first-order Taylor approximation of the nonlinear system. The resulting interval linear system is typically solved by the interval Gauss–Seidel method. The Gauss–Seidel method associates each constraint C_i with the variable x_i (after a possible renaming of variables), and loops while applying only the projection functions $\pi_{i,i}$.

To summarize, the main differences with our abstraction are that, in an implementation, the partial derivatives are recomputed periodically and not at each step, and that the Gauss–Seidel method does not apply all the projection

functions. A more realistic implementation of the Interval Newton method would correspond to Operator 2 as follows.[6]

Operator 2 (*Interval Newton operator*).

$Op_{Tay}(\mathcal{D}) = \mathcal{D}'$ where
$\mathcal{D}' := \mathcal{D}$;
Let $A_{i,j} = \left[nat\left(\frac{\partial f_j}{\partial x_i}\right)(\mathcal{D})\right], \quad j = 1,\ldots,n,\ i = 1,\ldots,n$
for $i = 1,\ldots,n$
$GS_i := \dfrac{-f_i(mid(\mathcal{D})) - \sum_{j=1,j\neq i}^{n} A_{i,j}(D'_j - mid(D_j))}{A_{i,i}} + mid(D_i)$;
$D'_i := D_i \cap GS_i$;
endfor

Note also that, in general, the Gauss–Seidel method does not converge towards the solution of the interval linear system, but it has good convergence properties for diagonally-dominant matrices. So, in practice, before solving the linear system, a preconditioning step is achieved that transforms the Jacobi matrix into a diagonally dominant matrix. Preconditioning consists of multiplying the interval linear equation $A * X = B$ by a matrix M, giving the new linear system $A' * X = B'$ where $A' = MA$ and $B' = MB$. The matrix M is typically the inverse of the midpoint matrix of A.

A nice property of the interval Newton operator is that in some cases, it is able to prove the existence of a solution. When $Op_{Tay}(\mathcal{D})$ is a strict subset of $\mathcal{D}$, Brouwer's fixed-point theorem applies and states existence and unicity of a solution in $\mathcal{D}$ (cf. [38]).

2.4. Strong consistencies

The idea of constraint satisfaction is to tackle difficult problems by solving easy-to-solve sub-problems: the constraints taken individually. It is often worth to have a more global view, which generally leads to a better enclosure of the domains. This is why strong consistencies have been proposed for solving CSP [21,22]. Their adaptation to numeric CSPs is summarized in this section. Interval analysis methods such as Op^{Tay} extensively use another kind of global view: the preconditioning of the Jacobi matrix. Nevertheless, the need for strong consistencies, although less crucial with interval analysis methods, may appear for very hard problems such as [43].

Strong consistencies have first been introduced over discrete CSPs (e.g., *path*-consistency, k-consistency [21] and (i, j)-consistency [22]), and then over numeric CSPs

[6] The for-loop corresponds to only one iteration of the Gauss–Seidel method and not to the complete solving of the interval linear system, which in practice is not useful [24].

($3B$-consistency [32] and kB-consistency [33]). kB-consistency is the adaptation of $(1,k)$-consistency over numeric CSP. Filtering by $(1,k)$-consistency is done by removing from each domain values that can not be extended to k variables. kB-consistency ensures that when a variable is instantiated to one of its two bounds, then the CSP is $|k-1|B$-satisfiable. When $k=2$, we refer to Operator 1. More generally, as given in Definition 2, $kB(w)$-consistency ensures that when a variable is forced to be close to one of its two bounds (more precisely, at a distance less than w), then the CSP is $|k-1|B(w)$-satisfiable. For simplest presentation, $2B(w)$-consistency refers to $2B$-consistency.

Definition 2 ($kB(w)$-*consistency*). We say that a CSP $\langle \mathcal{X}, D, C\rangle$ is $kB(w)$-consistent if and only if:

$$\begin{aligned}&\forall i,\ i \in \{1,\dots,n\} \Rightarrow\\ &\quad \underline{\psi}(\mathcal{D}, i, w) \text{ is } |k-1|B(w)\text{-satisfiable}, \quad \text{and}\\ &\quad \overline{\psi}(\mathcal{D}, i, w) \text{ is } |k-1|B(w)\text{-satisfiable},\end{aligned}$$

where $\underline{\psi}(\mathcal{D}, i, w)$ (respectively $\overline{\psi}(\mathcal{D}, i, w)$) denotes $\langle \mathcal{X}, \mathcal{D}', \mathcal{C}\rangle$ where $\mathcal{D}'$ is the same domain as $\mathcal{D}$ except that D_i is replaced by $D_i \cap [\underline{D_i}, \underline{D_i} + w]$ (respectively D_i is replaced by $D_i \cap [\overline{D_i} - w, \overline{D_i}]$).

The direct filtering operator $Op_{kB(w)}$ underlying the $kB(w)$-consistency uses a kind of proof by contradiction: the algorithm tries to increase the lower bound $\underline{D_i}$ by proving that the closure by $|k-1|B(w)$-consistency of $\langle D_1, \dots, [\underline{D_i}, \underline{D_i} + w], \dots, D_n\rangle$ is not empty and tries to decrease the upper bound in a symmetric way.

$3B$-consistency filtering algorithms, used for example in Interlog, Ilog Solver or Numerica, can be derived from Fig. 1 by instantiating operator Op to Op_{3B} as defined in Operator 3.

Operator 3 ($kB(w)$-*consistency filtering operator*: $Op_{kB(w)}$). Let $\mathcal{P} = \langle \mathcal{X}, D, C\rangle$, the filtering operator $Op_{kB(w)}(\mathcal{P})$, with $k \geqslant 3$, is defined as follows:

$Op_{kB(w)}(\mathcal{P}) = \langle \mathcal{X}, D', C\rangle, \quad k \geqslant 3.$

$\mathcal{D}'$ being computed as follows:

$\mathcal{D}' := \mathcal{D}$;

for $i = 1, \dots, n$ **do**

  while $D'_i \neq \emptyset \wedge \Phi_{|k-1|B(w)}\big(\underline{\psi}(\mathcal{D}', i, w)\big) = \emptyset$ **do**

    $\underline{D'_i} := \underline{D'_i} + w$;

  while $D'_i \neq \emptyset \wedge \Phi_{|k-1|B(w)}\big(\overline{\psi}(\mathcal{D}', i, w)\big) = \emptyset$ **do**

    $\overline{D'_i} := \overline{D'_i} - w$;

endfor

Fig. 3 shows how $3B(w)$-filtering uses $2B$-filtering.

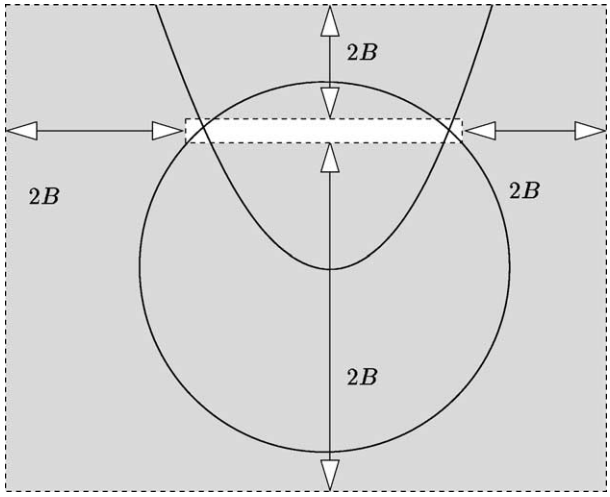

Fig. 3. $3B(w)$-filtering on the constraint system $\{x^2 + y^2 = 1, y = x^2\}$.

Implementations using this schema may be optimized considerably, but we do not need to go into details here. The reader is referred to [32] for the initial algorithm, and to [12] which studies the complexity of an unpublished implementation we used for years (see for example [30]) and that is more efficient than the algorithm published in [32].

The algorithm that achieves box-consistency is closely related to $3B$-consistency. Indeed, box-consistency can be seen as a kind of one-way $3B$-consistency limited to one constraint. The reader can found in [14] a theoretical comparison between box-consistency and $3B$-consistency.

3. Acceleration of filtering techniques

The question of choosing the best filtering algorithm for a given constraint system is an open problem. Some preliminary answers may come from the observation that the above fixed point algorithms suffer from two main drawbacks, which are tightly related:

- the existence of "slow convergences", leading to unacceptable response times for certain constraint systems;
- "early quiescence" [17], i.e., the algorithm stops before reaching a good approximation of the set of possible values.

The focus of this paper is on the first drawback. Its acuteness varies according to the *Op* operator:

Op_{nat} Due to its local view of constraints, Op_{nat} often suffers from early quiescence, but its simplicity makes it the most efficient operator to compute, and many problems

are best solved by this filtering operator (e.g., Moreaux problem [46]). At first sight, one could think that slow convergence phenomena do not occur very often with Op_{nat}. It is true that early quiescence of Op_{nat} is far more frequent than slow convergence. However, Op_{nat} is typically interleaved with a tree search (or is called from inside another higher-order filtering algorithm). During this interleaved process, slow convergence phenomena may occur and considerably increase the required computing time.

Op_{box} The comments above remain true for Op_{box}, although it may take more time to be computed and may perform some stronger pruning in some cases.

Op_{Tay} The interval Newton operator, on the one hand, may have a very efficient behavior. It may have an asymptotically quadratic convergence when it is used near the solution. In our experience, quadratic convergence is essential to compute precise roots of nonlinear systems of equations.

On the other hand, far from the solution, the Jacobi matrix has a great chance of being singular, which typically leads to the "early quiescence" problem. Hence Op_{Tay} does not have really slow convergence problems, but it needs expensive computation since the preconditioning of the Jacobi matrix needs to compute an inversion of its midpoint matrix. On some problems like Moreaux problem [46] with huge dimension $n \geqslant 320$, Op_{Tay} is very expensive, whereas by Op_{nat} the solution is found quickly.

$Op_{kB(w)}$ $kB(w)$-consistency filtering algorithms may perform a very strong pruning, making the tree-search almost useless for many problems.

For example, we have tried $4B(w)$-filtering over the transistor problem [41,43]. It finds the unique solution, without search, in the same cpu time as the $3B(w)$-filtering + search method used in [41]. We have also tried $4B(w)$-filtering over the benchmarks listed in [45]. They are all solved without search (only p choice points are made when the system has $p+1$ isolated solutions). Unfortunately, most of the time slow convergence phenomena occur during a $kB(w)$-filtering.

The different filtering algorithms are thus complementary and the more robust way to solve a problem is probably to use several of them together. In the fixed point schema of Fig. 1, the operator *Op* would be the result of the composition of some operators above. In the remainder of this section, we focus on the problem of slow convergence that occurs in Op_{nat} and $Op_{kB(w)}$.

The observation of many slow convergences of those algorithms led us to notice that some kinds of "regularity" often exist in a slow convergence phenomenon. Our intuition was that such regularities in the behavior of algorithms could be exploited to optimize their convergence. As seen in Section 2, the filtering algorithms are abstracted through a sequence of interval vectors. Accelerating a filtering algorithm thus consists in transforming that sequence into another sequence, hoping it converges faster. In numerical analysis, engineers use such transformation methods. Unfortunately, they cannot be sure of the reliability of their results. But this does not change the essence of usual floating-point computation: unreliability is everywhere! For filtering techniques, the completeness of the

results must be guaranteed, or in other words, no solution of the CSP can be lost. Thus the question of reliability becomes crucial. This leads us to define a *reliable transformation*.

Definition 3 (*Reliable transformation*). Let $\{S_n\}$ be a sequence that is complete for a set of solutions *Sol*: $\forall k, Sol \subseteq S_k$. Let $\mathcal{A}$ be a transformation and let $\{T_n\} = \mathcal{A}(\{S_n\})$. $\mathcal{A}$ is a reliable transformation for $\{S_n\}$ w.r.t. *Sol* if and only if

$$\forall k, \quad Sol \subseteq T_k.$$

The practical interest of a reliable transformation is directly related to its ability to accelerate the greatest number of sequences. Acceleration of a sequence is traditionally defined in terms of improvement of the convergence order of the sequence. Convergence order characterizes the asymptotic behavior of the sequences. (See Section 3.2 for a formal definition of the convergence order.) In addition to convergence order, some practical criteria may be of importance, like, for example, the time needed to compute a term of a sequence.

To build a reliable transformation that accelerates the original sequence, we will exploit some regularities in the sequence. When we detect a regularity in the filtering sequence, the general idea is to assume that this regularity will continue to appear in the following part of the sequence. The regularities that we are looking for are those which allow computations to be saved. A first kind of regularity that we may want to exploit is cyclicity. Section 3.1 summarizes a previous work based on that idea. Another kind of regularity, that can be caught by extrapolation methods, is then developed in Section 3.2.

3.1. A previous work: dynamic cycle simplification

This subsection summarizes a previous work [34,35], built on the idea that there is a strong connection between the existence of cyclic phenomena and slow convergence. More precisely, slow convergence phenomena move very often into cyclic phenomena after a transient period (a kind of stabilization step). The main goal is to dynamically identify cyclic phenomena while executing a filtering algorithm and then to simplify them in order to improve performance.

This subsection is more especially dedicated to the acceleration of Op_{nat} and Op_{box} algorithms, although:

- a direct use of those accelerated algorithms also leads to significant gain in speed for $kB(w)$-filtering algorithms since they typically require numerous computations of them;
- this approach could be generalized to identify cyclic phenomena in $kB(w)$-filtering algorithms.

Considering the application of Op_{2B} over $\mathcal{D}_i$, there may exist several projection functions that perform a reduction of the domain of a given variable. As Op_{2B} performs an intersection, and since domains are intervals, there may be 0, 1 or 2 projection functions of interest for each variable. (One that gives the greatest lower bound, one that gives the

lowest upper bound.) Call these projection functions *relevant* for $\mathcal{D}_i$, and denote by $\mathcal{R}_i$ the set of those relevant projection functions for $\mathcal{D}_i$.

Thus we have $\bigcap_{f\in\mathcal{R}_i} f(\mathcal{D}_i) = Op_{2B}(\mathcal{D}_i)$; that is, if we know in advance all the $\mathcal{R}_i$, we can compute $\Phi_{2B}(\mathcal{D})$ more efficiently by applying only relevant projection functions. This is precisely the case in a cyclic phenomenon.

We will say we have a cyclic phenomenon of period p when:

$$\forall i < N, \quad \mathcal{R}_{i+p} = \mathcal{R}_i,$$

where N is a "big" number.

Now, consider $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$. If a projection function is in $\mathcal{R}_{i+1}$, this is due to the reduction of domains performed by some projection functions in $\mathcal{R}_i$. We will say that $f \in \mathcal{R}_j$ depends on $g \in \mathcal{R}_i$, where $j > i$, denoted by $g \to f$ if and only if $g \neq f$ and g computes the projection over a variable that belongs to $var(f)$.

The dependency graph is the graph whose vertices are pairs $\langle f, i\rangle$, where $f \in \mathcal{R}_i$, and arcs are dependency links. (See Fig. 4(a).) If we assume that we are in a cyclic phenomenon, then the graph is cyclic. (See Fig. 4(b) where $\hat{0}$ denotes all the steps i such that $i \bmod 3 = 0$.) According to this assumption, two types of simplification can be performed:

- Avoid the application of non-*relevant* projection functions.
- Postpone some projection functions: a vertex $\langle f, i\rangle$ which does not have any successor in the dynamic dependency graph corresponds to a projection function that can be postponed. Such a vertex can be removed from the dynamic dependency graph. Applying this principle recursively will remove all non-cyclic paths from the graph. For instance, in graph (b) of Fig. 4, all white arrows will be pruned.
  When a vertex is removed, the corresponding projection function is pushed onto a stack. (The removing order must be preserved.) Then, it suffices to iterate on the simplified cycle until a fixed point is reached, and, when the fixed point has been reached, to evaluate the stacked projection functions.

The transformation that corresponds to the above two simplifications together is clearly a reliable transformation. It does not change the convergence order, but is in general an accelerating transformation. In [34] first experimental results are reported, gains in

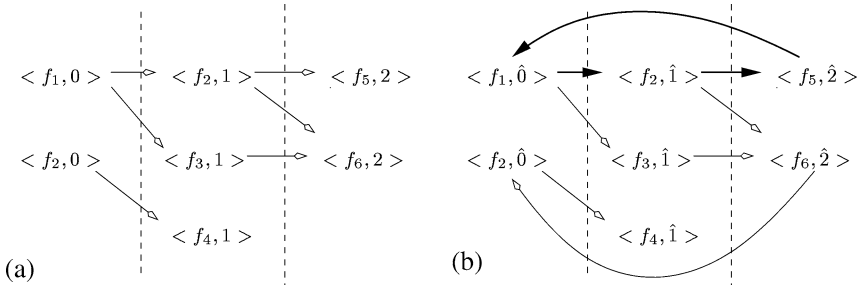

Fig. 4. Dynamic dependency graphs.

efficiency range from 6 to 20 times faster for $2B$-filtering and $3B(w)$-filtering. More complete experiments have been performed in [23], but, for the sake of simplicity, only the first simplification (applying only relevant projection functions) has been tried. Different combinations of several improvements of $2B$-filtering are tested. For all problems, the fastest combination uses cycle simplification. Ratio in CPU time varies from 1 to 20 compared with the same combination without cycle simplification.

3.2. Extrapolation

The previous section aims at exploiting cyclicity in the way projection functions are applied. The gain is in the computation of each term of $\{\mathcal{D}_n\}$, but the speed of convergence of $\mathcal{D}_n$ is unchanged. Now we address how to accelerate the convergence of $\{\mathcal{D}_n\}$.

$\{\mathcal{D}_n\}$ is a sequence of intervals. Numerical analysis provides different mathematical tools for accelerating the convergence of sequences of real numbers. Extrapolation methods are especially interesting for our purposes, but $\{\mathcal{D}_n\}$ is a sequence of interval vectors and there does not exist any extrapolation method to accelerate interval sequences. Nevertheless an interval can be seen as two reals and $\mathcal{D}$ can be seen as a 2-column matrix of reals. The first column is the lower bounds, and the second the upper bounds. Thus we can apply the existing extrapolation methods. The field of extrapolation methods, for real number sequences, is first summarized; for a deeper overview see [10]. Then we will show how to use extrapolation methods for accelerating filtering algorithms.

3.2.1. Extrapolation methods

Let $\{S_n\} = (S_1, S_2, \ldots)$ be a sequence of real numbers. A sequence $\{S_n\}$ converges if and only if it has a limit S: $\lim_{n\to\infty} S_n = S$. We say that the numeric sequence $\{S_n\}$ has the order $r \geqslant 1$ if there exist two finite constants A and B such that[7]

$$A \leqslant \lim_{n\to\infty} \frac{|S_{n+1} - S|}{|S_n - S|^r} \leqslant B.$$

A quadratic sequence is a sequence which has the order 2. We say that a sequence is linear if

$$\lim_{n\to\infty} \frac{(x_{n+1} - x)}{(x_n - x)} = \lambda, \quad \text{with } 0 < |\lambda| < 1.$$

The convergence order enables us to know exactly the convergence speed of the sequence. For example [8], for linear sequences with $\lambda = 0.999$, we obtain a significant number every 2500 iterations. Whereas, for sequences of order $r = 1.01$, the number of significant numbers doubles every 70 iterations. These examples show the interest of using sequences of order $r > 1$.

Accelerating the convergence of a sequence $\{S_n\}$ amounts of applying a transformation $\mathcal{A}$ which produces a new sequence $\{T_n\}$: $\{T_n\} = \mathcal{A}(\{S_n\})$.

[7] For more details, see [10].

As given in [10], in order to present some practical interest, the new sequence $\{T_n\}$ must exhibit, at least for some particular classes of convergent sequences $\{S_n\}$, the following properties:

(1) $\{T_n\}$ converges to the same limit as $\{S_n\}$: $\lim_{n\to\infty} T_n = \lim_{n\to\infty} S_n$.
(2) $\{T_n\}$ converges faster than $\{S_n\}$: $\lim_{n\to\infty} (T_n - S)/(S_n - S) = 0$.

These properties do not hold for all converging sequences. Particularly, a universal transformation $\mathcal{A}$ accelerating all converging sequences cannot exist [18]. Thus any transformation can accelerate a limited class of sequences. This leads us to a so-called *kernel*[8] of the transformation which is the set of convergent sequences (S_n) for which $\exists N, \forall n \succeq N, T_n = S$ where $\{T_n\} = \mathcal{A}(\{S_n\})$.

A well-known transformation is the iterated Δ^2 process from Aitken [1]

$$\{T_n\} = \Delta^2(\{S_n\}),$$

which gives a sequence $\{T_n\}$ of nth term

$$T_n = \frac{S_n S_{n+2} - S_{n+1}^2}{S_{n+2} - 2S_{n+1} + S_n}.$$

The kernel of Δ^2 process is the set of the converging sequences which have the form $S_n = S + \alpha\lambda^n$, where $\alpha \neq 0$ and $\lambda \neq 1$. Aitken's transformation has a nice property [10]: it transforms sequences with linear convergence into sequences with quadratic convergence.

We can apply the transformation several times, leading to a new transformation. For example, we can apply Δ^2 twice, giving $\Delta^2(\Delta^2(\{S_n\}))$. Many acceleration transformations (G-algorithm, ε-algorithm, θ-algorithm, *overholt*-process, ...) are multiple application of transformations. See [11] and [9] for attempts to build a unifying framework of transformation. Scalar transformations have been generalized to the vectoral and matrix cases.

Two kinds of optimization for filtering algorithms are now given. The first one makes a direct use of extrapolation methods and leads to a transformation which is not reliable. The second one is a reliable transformation.

3.2.2. Applying extrapolation directly

Let $\{\mathcal{D}_n\}$ be a sequence generated by a filtering algorithm. We can naively apply extrapolation method directly on the main sequence $\{\mathcal{D}_n\}$. The experimental results given in the rest of the paper are for scalar extrapolations, which consider each element of the matrix—each bound of a domain— independently of the others. For example, the scalar Δ^2 process uses for each bound of domain the last three different values to extrapolate a value.

[8] The definition of the kernel given here considers only converging sequences.

Accelerating directly the convergence of $\{\mathcal{D}_n\}$ can dramatically boost the convergence, as illustrated in the following problem:

$$\begin{cases} x * y + t - 2 * z = 4, \quad x * \sin(z) + y * \cos(t) = 0, \\ x - y + \cos(z)^2 = 0, \quad x * y * z - 2 * t = 0, \\ x \in [0, 1000],\ y \in [0, 1000],\ z \in [0, \pi],\ t \in [0, \pi]. \end{cases}$$

The following table shows the domain of the variable t in the 278th, 279th, 280th and 281st iterations of $3B$-filtering (after a few seconds on a Sun Sparc 5). The precision obtained is about 10^{-4}.

it	*t*
278	[**3.141**33342842583..., **3.14159265358979**...]
279	[**3.141**34152921220..., **3.14159265358979**...]
280	[**3.141**34937684900..., **3.14159265358979**...]
281	[**3.141**35697924715..., **3.14159265358979**...]

By applying Aitken's process on the domains of the iterations 278, 279 and 280, we obtain the domain below. The precision of this extrapolated domain is 10^{-14}. Such precision has not been obtained after 5 hours of the $3B$-filtering algorithm without extrapolation.

t
[**3.14159265358977**..., **3.14159265358979**...]

Let's take another example:

$$\begin{cases} y - x = 0; \\ y - 1.001 * x = 0; \\ \exp(y) - z_1 = 0; \\ \exp(z_1) - z_2 = 0; \\ x \in [-10^8, +10^8];\ y \in [-10^8, +10^8];\ z_1 \in [-10^8, +10^8];\ z_2 \in [-10^8, +10^8]; \end{cases}$$

Table 1 shows the domain of the variables x and y in the first, second and third iterations of $5B(w)$-filtering. The precision obtained is about 10^{-6}.

Table 1
$5B(w)$-filtering on the problem above

Iteration	domains for x and y
1	$x = [-1.1873006558144143\text{e}{-}06, 1.0460589320202715\text{e}{-}06]$
	$y = [-1.1884879564702285\text{e}{-}06, 1.0471049909522916\text{e}{-}06]$
2	$x = [-1.1778448513063027\text{e}{-}06, 1.0377279935029648\text{e}{-}06]$
	$y = [-1.1790226961576087\text{e}{-}06, 1.0387657214964676\text{e}{-}06]$
3	$x = [-1.1684643539568771\text{e}{-}06, 1.0294634035769800\text{e}{-}06]$
	$y = [-1.1696328183108338\text{e}{-}06, 1.0304928669805568\text{e}{-}06]$

By applying the Δ^2 process on the domain of the iterations 1, 2 and 3, we obtain the domains below. The precision of this extrapolated domain is 10^{-19}. Such a precision has not been obtained after many hours of the $5B(w)$-filtering algorithm without extrapolation.

$x = [-9.83\text{e}{-19}, 3.19\text{e}{-19}]$
$y = [-8.93\text{e}{-19}, -8.85\text{e}{-19}]$

This result is not surprising since we have the following proposition:

Theorem 1 (Convergence property of Aitken's process [7]). *If we apply* Δ^2 *on some sequence* (S_n) *which converges to* S *and if we have*:

$$\lim_{n\to\infty} \frac{\Delta S_{n+1}}{\Delta S_n} \neq 1$$

then the sequence $\Delta^2((S_n))$ *converges to* S*, and more quickly than* (S_n).

Note that, in the solution provided by Aitken's process, we have a valid result for x, but not for y. This example shows that extrapolation methods can lose solutions. The extrapolated sequence may or may not converge to the same limit as the initial sequence. This anomaly can be explained by the kernel of the transformation: when the initial sequence belongs to the kernel, then we are sure that the extrapolated sequence converges to the same limit. Furthermore, intuition suggests that, if the initial sequence is "close" to the kernel then there are good hopes to get the same limit. However, it may be the case that the limits are quite different. This is cumbersome for the filtering algorithms which must ensure that no solution is lost.

We propose below a reliable transformation that makes use of extrapolation.

3.2.3. Reliable transformation by extrapolation

The reliable transformation presented in this section is related to the domain sequences generated by $kB(w)$-filtering algorithms. For the sake of simplicity, we will only deal with $3B(w)$-filtering, but generalisation is straightforward.

This transformation is reliable thanks to the proof-by-contradiction mechanism used in $3B(w)$-algorithm: it tries to prove—with a $2B$-filtering—that no solution exists in a subpart of a domain. If such a proof is found, then the subpart is removed from the domain, else the subpart is not removed. The point is that we may waste a lot of time trying to find a proof that does not exist. If we could predict with good probability that such a proof does not exist, we could save time in not trying to find it. Extrapolation methods can do the job. The idea is simply that if an extrapolated sequence converges to a $2B$-satisfiable CSP (which can be quickly known), then it is probably a good idea not to try to prove the $2B$-unsatisfiability. This can be done by defining a new consistency, called $P2B(w)$-consistency, that is built upon the existence of a predicate $2B\text{-}predict(\mathcal{D})$ that predicts $2B$-satisfiability. ($P2B$ stands for $2B$ based on **P**rediction.)

Definition 4 ($P2B$*-consistency*). A CSP $\langle \mathcal{X}, D, C \rangle$ is $P2B$-consistent if and only if it is $2B$-consistent or $2B\text{-}predict(\mathcal{D})$ is true.

$$Op_{P2B}(\mathcal{D}_t) = \begin{cases} D_{t-1} & \text{if } 2B\text{-}predict(\mathcal{D}_1, \ldots, \mathcal{D}_{t-1}) \\ Op_{2B}(\mathcal{D}_{t-1}) & \text{otherwise} \end{cases}$$

Fig. 5. $P2B$-consistency filtering schema.

$2B$-*predict*$(\mathcal{D})$ may use extrapolation methods, for example the Δ^2 process. Thus, the prediction $2B$-*predict*$(\mathcal{D})$ may be wrong, but from the Proposition 1 we know that a filtering algorithm by $P2B$-consistency cannot lose any solutions.

Proposition 1. $\Phi_{P2B}(\mathcal{D}) \supseteq \Phi_{2B}(\mathcal{D})$.

The proof is straightforward from the definition.

A filtering algorithm that achieves $P2B$-consistency can be a fixed point algorithm where *Op* is as defined in Fig. 5. The main difference with Op_{2B} is, before testing for $2B$-satisfiability, to try in the function $2B$-*predict*, to predict $2B$-satisfiability by extrapolation methods. Following that idea, the algorithm schema for *Fast*-$3B(w)$-consistency can be modified, as given in Operator 4. (We may obtain in the same way the algorithm schema for *Fast*-$kB(w)$-consistency. It needs a P-kB operator that applies an extrapolator over the domains generated by the kB operator.)

Operator 4. Let $\mathcal{P} = \langle \mathcal{X}, D, C \rangle$, the filtering operator $Op_{\textit{Fast-}3B(w)}$ is defined as follows:

$Op_{\textit{Fast-}3B(w)}(\mathcal{P}) = \langle \mathcal{X}, D', C \rangle, \quad k \geqslant 3.$

$\mathcal{D}'$ being computed as follows:

$\mathcal{D}' := \mathcal{D}$;

for $i = 1, \ldots, n$ **do**

while $D'_i \neq \emptyset \wedge \Phi_{P2B}\big(\underline{\psi}(\mathcal{D}', i, w)\big) = \emptyset \wedge \Phi_{2B}\big(\underline{\psi}(\mathcal{D}', i, w)\big) = \emptyset$

do $\underline{D'_i} := \underline{D'_i} + w$;

while $D'_i \neq \emptyset \wedge \Phi_{P2B}\big(\overline{\psi}(\mathcal{D}', i, w)\big) = \emptyset \wedge \Phi_{2B}\big(\overline{\psi}(\mathcal{D}', i, w)\big) = \emptyset$

do $\overline{D'_i} := \overline{D'_i} - w$;

endfor

The following proposition means that this algorithm schema allows acceleration methods to be applied while keeping the completeness property of filtering algorithms. We thus have a reliable transformation.

Proposition 2 (Completeness). *Fast-*$kB(w)$*-algorithm does not lose any solutions.*

The proof is built on the fact that a domain is reduced only when we have a proof—by $|k-1|B(w)$-satisfiability and without extrapolation—that no solution exists for the removed part.

Table 2
Fast-3$B(w)$-filtering results over some benchmarks

Problem	$\frac{nbr-\pi(Fast\text{-}kB)}{nbr-\pi(kB)}$	$\frac{time(Fast\text{-}kB)}{time(kB)}$
brown	0.10	0.11
broyden(20)	0.28	0.47
caprasse	0.46	0.60
chemistry	0.61	0.69
cosnard(320)	1.00	1.09
neuro-100	0.53	0.66

The counterpart of this result is that improvements in efficiency of *Fast*-$kB(w)$-filtering compared with $kB(w)$-filtering may be less satisfactory than improvement provided by direct use of extrapolation. Another counterpart is that the greatest fixed point of $Op_{Fast\text{-}kB(w)}$ is generally greater than the greatest fixed point of $Op_{kB(w)}$.

In practice the overhead in time has always been negligible and the improvement in efficiency may vary from 1 to 10. Table 2 compares *Fast*-3B-filtering with 3B-filtering over some problems taken from [23,46]. It gives the ratios in time ($\frac{time(Fast\text{-}kB)}{time(kB)}$) and in number of projection function calls ($\frac{nbr-\pi(Fast\text{-}kB)}{nbr-\pi(kB)}$) for the two algorithms.

4. Related works

Two methods commonly used for solving numeric CSPs can be seen as reliable transformations: preconditioning and adding redundant constraints.

4.1. Preconditioning in the interval Newton operator

Numeric CSPs allow general numeric problems to be expressed, without any limitation on the form of the constraints. In numerical analysis, many specific cases of numeric CSPs have been studied. The preconditioning of squared linear systems of equations is among the most interesting results for its practical importance.

We say that a linear system of equation $Ax = b$ is well conditioned when

$$cond(A) = \|A\| \left\|A^{-1}\right\|$$

is near 1.

In practice, a well conditioned system is better solved than an ill conditioned one. Preconditioning methods transform the system $Ax = b$ to a new system $A'x = b'$ which has the same solution but is better conditioned than the first system. Solving $A'x = b'$ gives better precision and more reliable computations than solving the original system. A classic preconditioning method consists of multiplying the two sides of the system $Ax = b$ by an approximate inverse M of A. Thus we have $A' = MA$ and $b' = Mb$.

In interval analysis, the interest of preconditioning is not reliability, which already exists in interval methods, but precision and convergence. As already presented in Section 2.3, preconditioning is a key component of the interval Newton method. Experimental results

(for example see [28,45]) show the effectiveness of preconditioning for solving squared nonlinear systems of equations. Many theoretical results can be found in [2,28,38,39].

4.2. Redundant constraints

A classic reliable transformation is the adding of some redundant constraints to the original constraint system. This approach is very often used for discrete CSPs to accelerate the algorithms. It is not the case for interval analysis methods over numeric CSPs, since they exploit the fact that the system is square. For artificial intelligence methods over numeric CSPs, Benhamou and Granvilliers [4] propose to add some redundant polynomial constraints that are automatically generated by a depth-bounded Groebner bases algorithm.

5. Conclusion and perspectives

Our aim in this paper was to accelerate existing filtering algorithms. That led us to the concept of reliable transformation over the filtering algorithms, which preserves completeness of the filtering algorithms. Two kinds of reliable transformation have been proposed. They exploit some regularities in the behavior of the filtering algorithms. The first one is based on cyclic phenomena in the propagation queue. The second one is an extrapolation method: it tries to find a numeric equation satisfied by the propagation queue and then solves it.

A first perspective is to detect other kinds of regularities and to exploit them. A reliable transformation always has some intrinsic limitations; for example, logarithmic sequences cannot be accelerated by extrapolation methods. However, in that case, the cyclic phenomena simplification may improve the running time. Thus, combining different reliable transformations to try to accumulate the advantages of each transformation may be of high interest. Finally, a direction of research that could be fruitful comes from the remark that algorithms are designed with efficiency and simplicity in mind only. Regularity is never considered as an issue. Perhaps it is time to consider it as an issue, and to try to make more regular the existing algorithms in order to exploit their new regularities.

Acknowledgements

We would like to thank Christian Bliek, Michel Rueher and Patrick Taillibert for their constructive comments on an early draft of the paper, and Kathleen Callaway for a lot of English corrections. This work has been partly supported by the Ecole des Mines de Nantes.

References

[1] A. Aitken, On Bernoulli's numerical solution of algebraic equations, Proc. Roy. Soc. Edinburgh 46 (1926) 289–305.

[2] G. Alefeld, J. Hezberger (Eds.), Introduction to Interval Computations, Academic Press, New York, 1983.

[3] A. Babichev, O. Kadyrova, T. Kashevarova, A.S. Leshchenko, A.L. Semenov, UniCalc, A novel approach to solving systems of algebraic equations, Reliable Comput. 2 (1993) 29–47.
[4] F. Benhamou, L. Granvilliers, Automatic generation of numerical redundancies for non-linear constraint solving, Reliable Computing 3 (3) (1997) 335–344.
[5] F. Benhamou, W. Older, Applying interval arithmetic to real, integer and boolean constraints, J. Logic Programming 32 (1) (1997) 1–24.
[6] F. Benhamou, D. McAllester, P. Van Hentenryck, CLP(intervals) revisited, in: Proc. 1994 International Logic Programming Symposium, Ithaca, NY, 1994, pp. 124–138.
[7] C. Brezinski (Ed.), Algorithmes d'Accélération de la Convergence: Étude Numériques, Technip, 1978.
[8] C. Brezinski (Ed.), Introduction à la Pratique du Calcul Numérique, Dunod Université, Dunod, Paris, 1988.
[9] C. Brezinski, A.C. Matos, Derivation of extrapolation algorithms based on error estimates, J. Comput. Appl. Math. 66 (1996) 5–26.
[10] C. Brezinski, R. Zaglia (Eds.), Extrapolation Methods, Studies in Computational Mathematics, North-Holland, Amsterdam, 1990.
[11] C. Brezinski, R. Zaglia, A general extrapolation procedure revisited, Adv. Comput. Math. 2 (1994) 461–477.
[12] L. Bordeaux, E. Monfroy, F. Benhamou, Improved bounds on the complexity of kB-consistency, in: Proc. IJCAI-01, Seattle, WA, Morgan Kaufmann, San Mateo, CA, 2001, pp. 303–308.
[13] B. Botella, P. Taillibert, Interlog: Constraint logic programming on numeric intervals, in: Proc. 3rd International Workshop on Software Engineering, Artificial Intelligence and Expert Systems, Oberammergau, 1993.
[14] H. Collavizza, F. Delobel, M. Rueher, A note on partial consistencies over continuous domains solving techniques, in: Proc. Fourth International Conference on Principles and Practice of Constraint Programming (CP-98), Springer, Berlin, 1998, pp. 147–161.
[15] A. Colmerauer, Spécifications de Prolog IV, Tech. Rept., GIA, Faculté des Sciences de Luminy, Marseille, France, 1994.
[16] Dassault Electronique, Interlog 1.0: Guide d'utilisation, Tech. Rept., Dassault Electronique, Saint Cloud, France, 1991.
[17] E. Davis, Constraint propagation with interval labels, Artificial Intelligence 32 (1987) 281–331.
[18] J. Delahaye, B. Germain-Bonne, Résultats négatifs en accélération de la convergence, Numer. Math. 35 (1980) 443–457.
[19] B. Faltings, Arc consistency for continuous variables, Artificial Intelligence 60 (2) (1994) 363–376.
[20] B. Faltings, E. Gelle, Local consistency for ternary numeric constraints, in: Proc. IJCAI-97, Nagoya, Japan, Vol. 1, 1997, pp. 392–397.
[21] E. Freuder, Synthesizing constraint expressions, Comm. ACM 21 (1978) 958–966.
[22] E. Freuder, A sufficient condition for backtrack-bounded search, J. ACM 32 (4) (1985) 755–761.
[23] L. Granvilliers, Consistances locales et transformations symboliques de contraintes d'intervalles, Ph.D. Thesis, Université d'Orléans, France, 1998.
[24] E. Hansen (Ed.), Global Optimization Using Interval Analysis, Marcel Dekker, New York, 1992.
[25] D. Haroud, B. Faltings, Consistency techniques for continuous constraints, Constraints 1 (1–2) (1996) 85–118.
[26] E. Hyvönen, Constraint reasoning based on interval arithmetic: The tolerance propagation approach, Artificial Intelligence 58 (1992) 71–112.
[27] ILOG Solver 4.0, Reference Manual, 1997.
[28] R.B. Kearfott, Rigorous Global Search: Continuous Problems, Kluwer Academic, Dordrecht, 1996.
[29] V. Kreinovich, A. Lakeyev, J. Rohn, P. Kahl, Computational Complexity and Feasibility of Data Processing and Interval Computations, Kluwer, Dordrecht, 1998.
[30] Y. Lebbah, Contribution à la résolution de contraintes par consistence forte, Ph.D. Thesis, Université de Nantes 2, Nantes, France, 1994.
[31] Y. Lebbah, O. Lhomme, Acceleration methods for numeric CSPs, in: Proc. AAAI-98, Madison, WI, 1998, pp. 19–24.
[32] O. Lhomme, Consistency techniques for numeric CSPs, in: Proc. IJCAI-93, Chambéry, France, 1993, pp. 232–238.
[33] O. Lhomme, Contribution à la résolution de contraintes sur les réels par propagation d'intervalles, Ph.D. Thesis, Université de Nice—Sophia Antipolis, 1994.

[34] O. Lhomme, A. Gotlieb, M. Rueher, Dynamic optimization of interval narrowing algorithms, J. Logic Programming 37 (1–3) (1998) 165–183.
[35] O. Lhomme, A. Gotlieb, M. Rueher, P. Taillibert, Boosting the interval narrowing algorithm, in: Proc. 1996 Joint International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1996, pp. 378–392.
[36] A. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[37] R. Moore (Ed.), Interval Analysis, Prentice Hall, Englewood Cliffs, NJ, 1966.
[38] A. Neumaier, Interval Methods for Systems of Equations, Encyclopedia of Mathematics and its Applications, Vol. 37, Cambridge University Press, Cambridge, UK, 1990.
[39] A. Neumaier, A simple derivation of the Hansen-Bliek-Rohn-Ning-Kearfott enclosure for linear interval equations, Reliable Computing 5 (1999) 131–136.
[40] W. Older, A. Velino, Extending prolog with constraint arithmetic on real intervals, in: Proc. IEEE Canadian Conference on Electrical and Computer Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 14.1.1–14.1.4.
[41] J. Puget, P. Van Hentenryck, A constraints satisfaction approach to a circuit design problem, J. Global Optim. 13 (1) (1998) 75–93.
[42] H. Ratschek, J. Rokne, Computer Methods for the Range of Functions, Ellis Horwood Ser.: Math. Appl., Ellis Horwood, Chichester, 1984.
[43] H. Ratschek, J. Rokne, Experiments using interval analysis for solving a circuit design problem, J. Global Optim. 3 (1993) 501–518.
[44] G. Sidebottom, W. Havens, Hierarchical arc consistency applied to numeric constraint processing in logic programming, Technical Report CSS-IS TR 91-06, Center for Systems Science, Simon Fraser University, Burnaby, BC, 1991.
[45] P. Van Hentenryck, D. McAllester, D. Kapur, Solving polynomial systems using branch and prune approach, SIAM, J. Numer. Anal. 34 (2) (1997) 797–827.
[46] P. Van Hentenryck, L. Michel, Y. Deville, Numerica: A Modeling Language for Global Optimization, MIT Press, Cambridge MA, 1997.