HAL
open science

Dynamic optimization of interval narrowing algorithms

Olivier Lhomme, Arnaud Gotlieb, Michel Rueher

To cite this version:

Olivier Lhomme, Arnaud Gotlieb, Michel Rueher. Dynamic optimization of interval narrowing algorithms. Journal of Logic Programming, 1998, 37 (1-3), pp.165-183. 10.1016/S0743-1066(98)10007-9 . inria-00540302

HAL Id: inria-00540302

https://inria.hal.science/inria-00540302v1

Submitted on 26 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DYNAMIC OPTIMIZATION OF INTERVAL NARROWING ALGORITHMS

Olivier LHOMME [1], Arnaud GOTLIEB [2,3], Michel RUEHER [3]

▷

Interval narrowing techniques are a key issue for handling constraints over real numbers in the logic programming framework. However, the standard fixpoint algorithm used for computing an approximation of arc consistency may give rise to cyclic phenomena and hence to problems of slow convergence. Analysis of these cyclic phenomena shows: 1) that a large number of operations carried out during a cycle are unnecessary; 2) that many others could be removed from cycles and performed only once when these cycles have been processed. What is proposed here is a revised interval narrowing algorithm for identifying and simplifying such cyclic phenomena dynamically. These techniques are of particular interest for computing stronger consistencies which are often required for a substantial pruning. Experimental results show that such dynamic optimizations improve performance significantly. ◁

1. Introduction

Interval narrowing techniques allow a safe approximation of the set of values that satisfy an arbitrary constraint system to be computed. Lee and van Emden [19] have shown that the logic programming framework can be extended with relational interval arithmetic in such a way that its logic semantics is preserved, i.e., answers are logical consequences of declarative logic programs, even when floating-point computations have been used. These reasons have motivated the development of numerous CLP systems based on interval arithmetic (e.g., BNR-Prolog [30],

Address correspondence to
(1) Ecole des Mines de Nantes 4, rue Alfred Kastler, BP 20722, 44307 Nantes Cedex 03, France
(2) Dassault Electronique 55, Quai Marcel Dassault 92214 Saint-Cloud, France
(3) Université de Nice – Sophia Antipolis I3S-CNRS Route des colles, BP 145, 06903 Sophia Antipolis, France

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994
655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

CLP(BNR) [2], Interlog [17, 6, 20], Prolog IV [5]). All these systems use an arc consistency like algorithm [25] adapted for numeric constraints [9, 8]. This "standard" interval narrowing algorithm (named algorithm `IN` in the following) has two main drawbacks:

- the existence of "slow convergences", leading to unacceptable response times for certain constraint systems;
- the "early quiescence" [9], 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 problem. It shows that there is a strong connection between the existence of cyclic phenomena and slow convergence. The main goal is to dynamically identify cyclic phenomena while executing algorithm `IN` and then to simplify them in order to improve performance. The second problem is due to the fact that interval narrowing algorithms only guarantee a partial consistency. Many alternative approaches [16, 20, 13, 11, 4, 7, 31] have been proposed for tackling this problem. The framework introduced in this paper also leads to significant gain in speed for some of these approaches that are based on higher consistencies than arc-consistency. This is due to the fact that achieving higher consistency filtering (e.g., 3B–consistency filtering [20]) requires numerous computations of an approximation of arc-consistency.

1.1. A motivating example

Algorithm `IN` works iteratively: constraints are used for reducing domains until a fixpoint is reached.

The worst case running time of algorithm `IN` is bounded below by $\Omega(r \times m)$ and above by $O(r \times m \times a)$ where r is the arity of constraints, m is the number of constraints and a is the number of floating points numbers in the domains ([21]). Experimental running times of this algorithm are generally well below the upper bound of the running time. However, slow — or asymptotic — convergence phenomena sometimes occur, and then the experimental running time approaches the theoretical upper bound (see the example described in figure 1.1).

Intuitively these phenomena are cyclic. In the example of figure 1.1, the cycle is made up of the five constraints (a, b, c, d, e). However, the reduction of D_X induced by constraint (c) is stronger than the reduction of D_X induced by constraint (b), so there is no point in applying constraint (b). Only (a), (c), (d) and (e) are relevant and the cycle could be simplified to (a, c, d, e).

Constraints (d) and (e) only intervene in the cycle to reduce the domains of Z_1 and Z_2. It would be better to defer applying constraints (d) and (e). Thus, the cycle would be simplified to (a, c) and constraints (d, e) would only be applied once, when the fixpoint has been reached. The number of computations carried out by algorithm `IN` at each step would hence be minimized.

The presence of a cycle implies the existence of a series $u_k = f(u_{k-1})$ which converges towards a fixpoint u such that $u = f(u)$. The equation $u = f(u)$ could be infered and be solved by a computer algebra system. In the above example, constraints (a) and (b) are linear and can be solved symbolically. However, a symbolic solution cannot be computed for arbitrary systems of constraints.

$Y = X \quad (a)$
$Y = 1.001 * X \quad (b)$
$Y = 2 * X \quad (c)$
$Z_1 = e^Y \quad (d)$
$Z_2 = e^{Z_1} \quad (e)$
$D_X = [0, 10] \;\; D_Y = (-\infty, +\infty) \;\; D_{Z_1} = (-\infty, +\infty) \;\; D_{Z_2} = (-\infty, +\infty)$

(a) & $D_X = [0, 10] \longrightarrow D_Y = [0, 10]$
(b) & $D_Y = [0, 10] \longrightarrow D_X = [0, 9.99]$
(c) & $D_Y = [0, 10] \longrightarrow D_X = [0, 5]$
(d) & $D_Y = [0, 10] \longrightarrow D_{Z_1} = [1, e^{10}]$
(e) & $D_{Z_1} = [1, e^{10}] \longrightarrow D_{Z_2} = [e, e^{e^{10}}]$
(a) & $D_X = [0, 5] \longrightarrow D_Y = [0, 5]$
(b) & $D_Y = [0, 5] \longrightarrow D_X = [0, 4.99]$
(c) & $D_Y = [0, 5] \longrightarrow D_X = [0, 2.5]$
(d) & $D_Y = [0, 5] \longrightarrow D_{Z_1} = [1, e^5]$
(e) & $D_{Z_1} = [1, e^5] \longrightarrow D_{Z_2} = [e, e^{e^5}]$
etc.

FIGURE 1.1. A slow convergence phenomenon

The aim of this paper is to dynamically simplify the evaluation of the terms of the series $u_k = f(u_{k-1})$ in order to accelerate convergence towards the fixpoint u. Two types of cycle simplifications are proposed: removing the non-*relevant* narrowing functions and postponing some other ones. More precisely, given a cyclic phenomenon (a, b, c, d, e) such that:

- b performs a weaker reduction than c,
- d and e could be processed only once at the end of the cycle,

the goal is to replace n iterations of (a, b, c, d, e) by n iterations of (a, c) followed by one iteration of (d, e).

1.2. Relevance of automatic cycle simplification

At first sight, one could think that slow convergence phenomena do not occur very often. It is true that early quiescence of algorithm `IN` is far more frequent than slow convergence. However, when algorithm `IN` ends prematurely, a kind of enumeration interleaved with this algorithm is generally performed (e.g. domain splitting [8] or stronger consistencies [20, 11, 4, 35, 12]). During this interleaved process, slow convergence phenomena may occur and considerably increase the required computing time.

Slow convergence phenomena move very often into cyclic phenomena after a transient period (a kind of stabilization step). For linear systems of constraints, slow convergence always entails a cyclic phenomenon. Of course, in this case the slow convergence phenomenon can be removed by simplifying the linear system with a linear solver. Cooperation between an interval narrowing solver and a linear solver is especially worthwhile in this latter case [1, 31, 7, 26, 32]. For arbitrary

non-linear systems, slow convergence very often leads to a cyclic phenomenon too. As arbitrary non-linear systems cannot be tackled with a symbolic solver, automatic cycle simplification is the only way to accelerate convergence in many real applications.

1.3. Layout of the paper

Section 2 reviews some basics concepts required for the rest of the paper. In section 3, the concept of propagation cycle is introduced. It is shown that algorithm `IN` does not allow cyclic phenomena to be satisfactorily simplified. Thus, a revised interval narrowing algorithm is proposed in which cyclic phenomena can be significantly simplified. Simplification of a cycle is described in section 4. In section 5, experimental results are provided. Finally, in section 6, the limitations and possible extensions of our approach are discussed.

2. Interval narrowing

In this section, we recall some basic concepts concerning interval narrowing techniques. More complete information on that subject can be found in [16, 20, 13, 4, 35].

2.1. Basic notations

Let be $\mathcal{R}^\infty = \mathcal{R} \cup \{-\infty, +\infty\}$ the set of real numbers augmented with the two infinity symbols. $\overline{\mathbb{F}}$ denotes a finite subset of $\mathcal{R}^\infty$ containing $\{-\infty, +\infty\}$. Practically, $\overline{\mathbb{F}}$ corresponds to the set of floating-point numbers used in the implementation. $\{-\infty, +\infty\}$ represents respectively all numbers smaller (resp. greater) than the smallest (resp. the biggest) floating-point number. Let $a \in \overline{\mathbb{F}}$, a^+ (resp. a^-) corresponds to the smallest (resp. largest) number of $\overline{\mathbb{F}}$ strictly greater (resp. smaller) than a.

Definition 2.1. [Interval] An interval $[a, b]$ with $a, b \in \overline{\mathbb{F}}$ is the set of real numbers $\{r \in \mathcal{R} \mid a \leq r \leq b\}$

Let r be a real number. $\tilde{r}$ denotes the smallest (w.r.t. inclusion) interval of $\overline{\mathbb{F}}$ containing r. $\mathcal{I}$ denotes the set of intervals and is ordered by set inclusion (this kind of intervals are sometimes called floating point intervals in the literature). $\mathcal{U}(\mathcal{I})$ denotes the set of unions of intervals. $\subset$ denotes the usual inclusion on vectors of $\mathcal{I}^k$ or $\mathcal{U}(\mathcal{I})^k$.

A CSP [25] is a triple $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ denotes a set of variables, $\vec{\mathcal{D}} = (D_1, \ldots, D_n)$ denotes a vector of domains, D_i the ith component of $\vec{\mathcal{D}}$ being the interval containing all acceptable values for x_i, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ denotes a set of constraints.

A k-ary constraint C is a relation over the reals. $\rho(C)$ denotes the subset of $\mathcal{R}^k$ satisfying constraint C. $\tilde{\rho}(C)$ denotes the smallest[1](w.r.t. inclusion) subset of $\mathcal{I}^k$

[1] The term smallest (w.r.t. inclusion) subset must be understood here according to the precision

which contains $\rho(C)$, i.e.,

1. $< r_1, \ldots, r_k > \in \rho(C) \Rightarrow < \tilde{r_1}, \ldots, \tilde{r_k} > \in \tilde{\rho}(C)$
2. $< I_1, \ldots, I_k > \in \tilde{\rho}(C) \Rightarrow \exists < r_1, \ldots, r_k > \in < I_1, \ldots, I_k > \mid$
   $\forall j \in 1..k,\ \tilde{r_j} = I_j$ and $< r_1, \ldots, r_k > \in \rho(C)$.
3. $\vec{\mathcal{I}} \in \tilde{\rho}(C) \wedge \vec{\mathcal{I}'} \subset \vec{\mathcal{I}} \wedge \vec{\mathcal{I}'} \neq \vec{\mathcal{I}} \Rightarrow \vec{\mathcal{I}'} \notin \tilde{\rho}(C)$

$P_\emptyset$ denotes an empty CSP, i.e, a CSP with at least one empty domain. $\vec{\mathcal{D}'} \subset \vec{\mathcal{D}}$ means $D'_i \subset D_i$ for all $i \in 1..n$.

2.2. 2B–consistency

Most of the CLP systems over intervals (e.g., [29, 17, 2, 5]) compute an approximation of arc-consistency [25] which will be named *2B–consistency* [20] in this paper. 2B–consistency states a local property on a constraint and on the bounds of the domains of its variables (B in 2B–consistency stands for *bound*). Roughly speaking, a constraint C is 2B–consistent if for any variable x in C the bounds a and b of the domain $D_x = [a, b]$ have a support in the domains of all other variables of C. However, the bounds may only be floating point numbers whereas constraint C may hold for values of x which are not floating point numbers; the formal definition makes use of semi-open intervals to take this point into account.

Definition 2.2. [2B–consistency]
Let $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ be a CSP and $C \in \mathcal{C}$ a k-ary constraint over the variables $(x_1, \ldots, x_k)$. C is 2B–consistent iff:
$\forall x_i \in (x_1, \ldots, x_k)$ let $D_{x_i} = [a, b]$
$\exists v_1 \in D_{x_1}, \ldots, \exists v_i \in [a, a^+), \ldots, \exists v_k \in D_{x_k}$ such that $(v_1, \ldots, v_k) \in \rho(C)$
and $\exists v'_1 \in D_{x_1}, \ldots, \exists v'_i \in (b^-, b], \ldots, \exists v'_k \in D_{x_k}$ such that $(v'_1, \ldots, v'_k) \in \rho(C)$

A CSP is 2B–consistent iff all its constraints are 2B–consistent.

Example 2.1. Let $P_1 = (\{x, y\}, \{D_x = [1, 4],\ Dy = [-2, 2]\}, \{x = y^2\})$. P_1 is 2B–consistent because $\{< 1, 1 >, < 4, -2 >, < 4, 2 >\} \subset \rho(x = y^2)$.

2B–consistency is a weaker consistency than arc consistency. For instance, P_1 is 2B–consistent but not arc–consistent since there is no value in D_x which satisfies the constraint when $y = 0$.

Closure by 2B–consistency of a CSP $P = (\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ is a CSP $P' = (\mathcal{X}, \vec{\mathcal{D}'}, \mathcal{C})$ such that :

- P and P' have the same solutions;
- P' is 2B–consistent;

of floating-points operations. In the rest of the paper, we consider —as in [19, 4]— that results of floating-points operations are outward-rounded to preserve correctness of the computation.

- $\vec{\mathcal{D}'} \subset \vec{\mathcal{D}}$ and domains in $\vec{\mathcal{D}'}$ are the largest for which P' is 2B–consistent.

We note $\Phi_{2B}(P)$ the closure by 2B–consistency of P.

2.3. 3B–consistency

2B–consistency is only a partial consistency, and then it is often too weak for computing an accurate approximation of the set of solutions of a CSP. In the same way that arc-consistency has been generalized to higher consistencies (e.g., path consistency [14]), 2B–consistency can be generalized to 3B–consistency [20].

Definition 2.3. [3B–consistency] Let $P = (\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ be a CSP and x a variable of $\mathcal{X}$ with domain $[a, b]$. Let also be:

- P_1 the CSP derived from P by substituting D_x in $\vec{\mathcal{D}}$ by $D_x^1 = [a, a^+]$;
- P_2 the CSP derived from P by substituting D_x in $\vec{\mathcal{D}}$ by $D_x^2 = [b^-, b]$.

D_x is 3B–consistent iff:

1. $\Phi_{2B}(P_1) \neq P_\emptyset$
2. $\Phi_{2B}(P_2) \neq P_\emptyset$

A CSP is 3B–consistent iff all its domains are 3B–consistent.

It results from this definition that any CSP which is 3B–consistent is also 2B–consistent. The generalization of the 3B–consistency to kB–consistency is straightforward and is given in [21, 22].

Closure by kB–consistency of P is defined in a similar way as closure by 2B–consistency of P, and is denoted by $\Phi_{kB}(P)$.

Filtering algorithms for computing 2B–consistency and 3B–consistency closures use an approximation of the unary projection of the constraints to reduce the domains of the variables. Next section introduces the narrowing functions used for computing such projections. Algorithms will be introduced afterwards.

2.4. Narrowing functions

Let C be a k-ary constraint over $(x_{i_1}, \ldots, x_{i_k})$, and $< I_1, \ldots, I_k > \in \mathcal{I}^k$: for each j in $1..k$, $\pi_{i_j}(C, I_1 \times \ldots \times I_k)$ denotes the projection of $\tilde{\rho}(C)$ on x_{i_j} in the part of the space delimited by $I_1 \times \ldots \times I_k$.

Definition 2.4. [projection of a constraint] $\pi_{i_j}(C, I_1 \times \ldots \times I_k) : (\mathcal{C}, \mathcal{I}^k) \rightarrow \mathcal{U}(\mathcal{I})$ is the projection of $\tilde{\rho}(C)$ on x_{i_j}iff:

$$\pi_{i_j}(C, I_1 \times \ldots \times I_k) = \{\tilde{a}_j \mid \exists < \tilde{a}_1, \ldots, \tilde{a}_k > \in \tilde{\rho}(C) \ \cap \ I_1 \times \ldots \times I_k\}$$

$AP_i(C, I_1 \times \ldots \times I_k) : (\mathcal{C}, \mathcal{I}^k) \rightarrow \mathcal{I}$ denotes an approximation of the projection of a constraint equal to the smallest interval encompassing the projection, i.e., the interval $[a, b]$ such that a (resp. b) is the smallest (resp. largest) value of $\pi_i(C, I_1 \times$

$\ldots \times I_k$). For instance, let C be the constraint $x_1 - x_2 + 3 = 0$, $AP_1(C, I_1 \times I_2)$ can be expressed by $I_1 \cap (I_2 - 3)$ using interval arithmetic [28].

Such an approximation [2] is computed by the evaluation of what will be called a *narrowing function*. For convenience, a narrowing function will be considered as a filtering operator over all the domains, i.e., from $\mathcal{I}^n$ to $\mathcal{I}^n$. For a k-ary constraint C over $(x_{i_1}, \ldots, x_{i_k})$ there are k narrowing functions, one for each x_{i_j}.

Definition 2.5. [narrowing function] The narrowing function of C over the variable x_i is the function $f : \mathcal{I}^n \longrightarrow \mathcal{I}^n$ such that $f(\vec{\mathcal{D}}) = \vec{\mathcal{D}'}$ where :

$$\forall j \in \{1, \ldots, n\} \quad D'_j = \begin{cases} D_j & \text{if } j \neq i \\ AP_i(C, D_{i_1} \times \ldots \times D_{i_k}) & \text{if } j = i \end{cases}$$

A narrowing function f reduces the domain of at most one variable (x_i in the previous definition), called left-variable of f and denoted $f.y$ (on the analogy of the notation $y = f(x)$). The constraint from which the function f is issued is denoted $f.c$ and the set of variables whose domains are required for the evaluation of the domain of $f.y$ is called right-variables set and is denoted $f.x_s$. The three following properties trivially hold:

- $f(\vec{\mathcal{D}}) \subseteq \vec{\mathcal{D}}$
- $f(f(\vec{\mathcal{D}})) = f(\vec{\mathcal{D}})$
- if f and g are narrowing functions of the same constraint (i.e, $g.c = f.c$) then $f(g(f(\vec{\mathcal{D}}))) = g(f(\vec{\mathcal{D}}))$

In this paper, a numeric CSP $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ will also be denoted by a triple $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ where $\mathcal{F}$ is the set of narrowing functions corresponding to the constraints in $\mathcal{C}$. Figure 2.1 shows such a view of a CSP ($\Pi_j(C)$ denotes the narrowing function of C over the variable x_j; e.g., $f = \Pi_1(x_1 - x_2 + 3 = 0)$ reduces D_1 by using D_2).

Let $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ be a CSP where $\mathcal{C} = \{x_1 - x_2 + 3 = 0, x_3 = x_1\}$:
This CSP can be formulated in the form $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ where $\mathcal{F} = \{f, g, h, i\}$

$f = \Pi_1(x_1 - x_2 + 3 = 0)$	$h = \Pi_2(x_1 - x_2 + 3 = 0)$
$i = \Pi_1(x_3 = x_1)$	$g = \Pi_3(x_3 = x_1)$

FIGURE 2.1. A CSP in the form $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$

[2] For most non-linear constraint systems, $AP_i(C_p, I_1 \times \ldots \times I_k)$ cannot be computed in a straightforward way. However, interval arithmetic [28] allows $AP_i(C_p, I_1 \times \ldots \times I_k)$ to be computed on a subset of the constraints set, called basic constraints. Each constraint can be approximated by decomposition in basic constraints. Other approximation of the projection (e.g. [4, 35]) can also be used.

2.5. Filtering operator $\widetilde{\mathcal{T}}$

A set of narrowing functions $\mathcal{T}$ will be associated to a filtering operator $\widetilde{\mathcal{T}}$ that computes the intersection of the domains narrowed by the functions in $\mathcal{T}$ (all functions in $\mathcal{T}$ are applied to the same vector of domains $\vec{\mathcal{D}}$).

Definition 2.6. [filtering operator $\widetilde{\mathcal{T}}$] Let $\mathcal{T} = \{f_1 \ldots, f_p\} \subset \mathcal{F}$. $\widetilde{\mathcal{T}}\,(\vec{\mathcal{D}})$ is defined by:

- If $\mathcal{T} \neq \emptyset$ then $\widetilde{\mathcal{T}}\,(\vec{\mathcal{D}}) = f_1(\vec{\mathcal{D}}) \cap \ldots \cap f_p(\vec{\mathcal{D}})$
- If $\mathcal{T} = \emptyset$ then by convention $\widetilde{\mathcal{T}}\,(\vec{\mathcal{D}}) = \vec{\mathcal{D}}$.

2.6. Interval Narrowing Algorithms

Using the above notations, algorithm `IN` [9, 8] can be written down as in figure 2.2. `IN` implements the computation of the closure by 2B-consistency of a CSP P. The following proposition also characterizes the fixpoint computed by `IN`.

Proposition 2.1. Let $P = (\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ be a CSP and $P' = (\mathcal{X}, \vec{\mathcal{D}'}, \mathcal{F})$ be the closure by 2B-consistency of P. Then:

- $\vec{\mathcal{D}'} = \widetilde{\mathcal{F}}\,(\vec{\mathcal{D}'})$
- *$\vec{\mathcal{D}'}$ is the largest fixpoint for $\widetilde{\mathcal{F}}$ included in $\vec{\mathcal{D}}$*

```
IN(in F, inout D)
    Queue ← F ;
    while Queue ≠ ∅
        f ← POP Queue;
        D'← f(D);
        if D' ≠ D then    D ← D';
                          Queue ←Queue ∪ {g ∈ F | g.c ≠ f.c and f.y ∈ g.x_s}
        endif
    endwhile
```

FIGURE 2.2. Algorithm `IN`

Algorithms for computing higher consistencies can be found in [20, 21, 22]. We just give here the main idea of a naive 3B–filtering algorithm.
Let $P = (\mathcal{X}, \vec{\mathcal{D}}, \mathcal{C})$ be a CSP, x be a variable of $\mathcal{X}$ with domain $[a, b]$ and $D'_x = [c, d]$ be the domain of x in the closure by 3B-consistency of P.
In order to find c, we try to refute the part $[a, c)$ of the domain D_x. First we try

to prove that CSP P_l derived from P by substituting D_x in $\vec{\mathcal{D}}$ by $[a,(a+b)/2]$ is inconsistent (i.e., $\Phi_{2B}(P_l) = P_\emptyset$). If successful, this process is restarted with the midpoint of the remaining interval, otherwise we try to refute a smaller part on the left of D_x (e.g., $[a,(a+b)/4]$). The process stops when the part of D_x which could be removed is smaller than a given value ϵ. The same process could be applied to find the upper bound d. In fact, the algorithm works in a round-robin way over all the variables.

The key point is that kB–filtering algorithms make an intensive use of algorithm `IN`. Thus, any cycle optimization in `IN` will dramatically improve kB–filtering algorithms.

3. Towards a characterization of the cyclic phenomenon

When algorithm `IN` runs into a slow convergence phenomenon a cyclic phenomenon may occur after a transient period. In this section, we give a precise characterization of a cyclic phenomenon. Let us outline our approach in very general terms:

1. we show that information about some dynamic dependencies (in place of static ones) between narrowing functions is required;
2. we show that such information about dynamic dependencies cannot be identified in the framework of algorithm `IN`. This is due to the fact that the order in which the narrowing functions are enqueued plays a major role in `IN`;
3. we introduce a revised version of algorithm `IN` in order to get information about some dynamic dependencies.

Further definitions are now required to formalize such cyclic phenomena.

3.1. Static dependencies

A static dependency between two narrowing functions f and g means that after an evaluation of f which does modify the domain of $f.y$, g may reduce the domain of $g.y$ (the narrowing functions enqueued in algorithm `IN` are the ones which statically depend on f).

Definition 3.1. [Static dependency] There is a static dependency between two narrowing functions f and g, denoted $f \xrightarrow{s} g$, iff:

- $g.c \neq f.c$ (f and g are functions not issued from the same constraint)
- $f.y \in g.x_s$ (the left-variable of f occurs in the right-variables set of g)

We note $succ_s(\mathcal{T})$ the successors in the static dependency graph of a set of narrowing functions $\mathcal{T}$: $succ_s(\mathcal{T}) = \{g \in \mathcal{F} \mid \exists f \in \mathcal{T} \text{ and } f \xrightarrow{s} g\}$.

Static dependency information may not be sufficient for cycle simplification. For instance, consider the example in figure 2.1: $f \xrightarrow{s} g$, hence, $g(f(\vec{\mathcal{D}}))$ may be different from $f(\vec{\mathcal{D}})$. However, let $\vec{\mathcal{D}'} = f(\vec{\mathcal{D}})$ and suppose that D'_3 is included in D'_1, then

$g(f(\vec{\mathcal{D}})) = f(\vec{\mathcal{D}})$. Such an equality would allow g to be removed from a possible cycle; unfortunately, $f \xrightarrow{s} g$ does not allow to infer this equality and thus no cycle simplification can be performed in this case.

What is needed is a dynamic dependency $f \xrightarrow{d} g$ that ensures that a modification induced by f actually implies a modification induced by g. The first idea is to follow algorithm `IN` and try to identify such dynamic dependencies.

3.2. Dynamic dependencies

To have a dynamic dependency between function f and function g, the following conditions are required:

- f was applied before g
- f reduced some domain and g reduced some domain
- g statically depends on f : $f \xrightarrow{s} g$

Algorithm `IN` computes the terms of a sequence of ith term $f_i(f_{i-1}(\ldots f_0(\vec{\mathcal{D}})))$ characterizing the order in which the narrowing functions f_j are enqueued: $f_i(f_{i-1}(\ldots f_0(\vec{\mathcal{D}})))$ corresponds to the enqueueing order $(f_0, f_1, \ldots, f_i)$.

Let us assume that a dynamic dependency holds between f and g if $f \xrightarrow{s} g$ and $g(f(\vec{\mathcal{D}})) \neq f(\vec{\mathcal{D}})$. Such a definition would lead to several problems:

1. $g(f(\vec{\mathcal{D}}))$ is not always computed by algorithm `IN` since some narrowing functions may have been enqueued between f and g, e.g., `IN` may compute $g(h_1(\ldots(h_k(f(\vec{\mathcal{D}})))))$.

2. The fact that $f \xrightarrow{s} g$ and $g(f(\vec{\mathcal{D}})) \neq f(\vec{\mathcal{D}})$ does not always imply an effective dynamic dependency between f and g since $g(\vec{\mathcal{D}})$ could be different from $\vec{\mathcal{D}}$. For instance, if $g(f(h_1(\ldots(h_k(\vec{\mathcal{D}}_0)))))$ is computed, then the effective dynamic dependency may hold between h_j and g.

3. The narrowing functions which g dynamically depends on may be dynamically dependent between themselves; this means that the dependencies are interleaved.

The above three problems are simultaneously illustrated in example 3.1.

Example 3.1. Let $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ be a CSP where :

- $\{f, g, h\} \subset \mathcal{F}$
- $\{x_1, x_2, x_3, x_4, x_5\} \subset \mathcal{X}$
- $D_4 = [0, 2\pi]$
- $f = \Pi_1(x_1 = x_5)$
- $g = \Pi_2(x_2 = x_1)$
- $h = \Pi_3(x_3 = x_1 + cos(x_4 + x_2))$

Suppose that $h(g(f(\vec{\mathcal{D}})))$ is computed (according to some enqueueing order of the narrowing functions). Suppose also that $\vec{\mathcal{D}}$ verifies:

- $g(\vec{\mathcal{D}}) = \vec{\mathcal{D}}$ and $h(\vec{\mathcal{D}}) = \vec{\mathcal{D}}$,
- $f(\vec{\mathcal{D}}) \neq \vec{\mathcal{D}}$,
- $g(f(\vec{\mathcal{D}})) \neq f(\vec{\mathcal{D}})$,
- $h(g(f(\vec{\mathcal{D}}))) \neq g(f(\vec{\mathcal{D}}))$.

That is f, g and h perform reductions. The static dependencies are: $f \xrightarrow{s} g, f \xrightarrow{s} h, g \xrightarrow{s} h$. According to the above naive definition $g \xrightarrow{d} h$ holds but we cannot infer that h depends on f since $h(f(\vec{\mathcal{D}}))$ is not computed. However h actually only depends on f (the reduction of D_3 is only due to the modification of D_1 computed by f since one period of *cosine* is in the domain D_4).

It follows that a stronger definition of the dynamic dependency is required. The trick is to find a definition that not only avoids the above-mentioned problems but also allows an efficient computation of these relations. Before giving the definition used in the paper, we introduce a revised algorithm for interval narrowing that provides support for identifying relevant narrowing functions.

3.3. Revised algorithm for interval narrowing

Since closure by 2B-consistency is a fixpoint for $\widetilde{\mathcal{F}}$, it may be computed by repeatedly applying $\widetilde{\mathcal{F}}$ over $\vec{\mathcal{D}}$. When computing the terms of such a sequence, some narrowing functions cannot reduce any domain. It follows that it suffices to evaluate the terms of the sequence:

$$\widetilde{\mathcal{T}}_n(\widetilde{\mathcal{T}}_{n-1}(\ldots(\widetilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0)))) \text{ where}$$

- $\vec{\mathcal{D}}_0 = \vec{\mathcal{D}}$, $\mathcal{T}_0 = \mathcal{F}$,
- $\mathcal{T}_i = succ_s(\{f \in \mathcal{T}_{i-1}$ such that $f.y = x_j$ and D_j has been reduced by $\widetilde{\mathcal{T}}_{i-1}\})$.

Definition 3.2. $\vec{\mathcal{D}}_i$ denotes the domain vector at the ith step:

$$\vec{\mathcal{D}}_i = \widetilde{\mathcal{T}}_{i-1}(\ldots\widetilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0))$$

Proposition 3.1. $\widetilde{\mathcal{T}}_i(\vec{\mathcal{D}}_i) = \widetilde{\mathcal{F}}(\vec{\mathcal{D}}_i)$

Corollary 3.1. $\widetilde{\mathcal{T}}_i(\ldots\widetilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0)) = (\widetilde{\mathcal{F}})^{i+1}(\vec{\mathcal{D}}_0)$.

Algorithm `Revised-IN` (Figure 3.1) computes this sequence: it applies on the same vector $\vec{\mathcal{D}}$ all the narrowing functions which may reduce a domain.

Corollary 3.2. The fixpoint computed by algorithm `Revised-IN` *is identical to the one computed by algorithm* `IN`.

Using this algorithm to compute the fixpoint would push the upper bound of the running time to $O(\frac{r^2 \times m^2 \times a}{n})$ instead of $O(r \times m \times a)$ for `IN` where m is the number of (basic) constraints, n is the number of variables, r is the arity of constraints and a the size of the largest domain. Thus, it will only be used for computing the dynamic dependencies.

Revised-IN(**in** $\mathcal{F}$, **inout** $\vec{\mathcal{D}}$)
  $\mathcal{T} \leftarrow \mathcal{F}$;
  while $\mathcal{T} \neq \emptyset$
    $\vec{\mathcal{D}}' \leftarrow \vec{\mathcal{D}}$;
    $\vec{\mathcal{D}} \leftarrow \tilde{\mathcal{T}}\ (\vec{\mathcal{D}})$;
    $\delta \leftarrow \{f \in \mathcal{T} \mid D'_i \neq D_i \text{ and } f.y = x_i\}$;
    $\mathcal{T} \leftarrow\ succ_s(\delta)$;

FIGURE 3.1. Algorithm `Revised-IN`

3.4. Relevant narrowing functions

As outlined in the introduction, when two narrowing functions perform a reduction of the domain of a given variable, it is possible to remove the narrowing function which performs the weakest reduction of that domain.

The *relevant* narrowing functions are those narrowing functions which perform the strongest reductions of the domains of the variables during the application of the operator $\tilde{\mathcal{T}}_i$ on $\vec{\mathcal{D}}_i$. Since the domains are intervals, there may be 0, 1 or 2 (one for the lower bound, one for the upper bound) *relevant* narrowing functions for each variable. $\mathcal{R}_i$ denotes the set of those relevant narrowing functions.

Definition 3.3. [relevant narrowing functions] $\mathcal{R}_i \subseteq \mathcal{T}_i$ is the minimal[3] subset of $\mathcal{T}_i$ such that $\tilde{\mathcal{R}}_i\ (\vec{\mathcal{D}}_i) = \tilde{\mathcal{T}}_i\ (\vec{\mathcal{D}}_i)$.

It follows that:

Proposition 3.2. $\tilde{\mathcal{R}}_i\ (\vec{\mathcal{D}}_i) = \tilde{\mathcal{F}}(\vec{\mathcal{D}}_i)$.

Corollary 3.3. $\tilde{\mathcal{R}}_i(...\tilde{\mathcal{R}}_0(\vec{\mathcal{D}}_0)) = (\tilde{\mathcal{F}})^{i+1}(\vec{\mathcal{D}}_0)$.

Computing $\mathcal{R}_i$ only consists — when applying $\tilde{\mathcal{T}}_i$ in `Revised-IN` — in keeping, for each bound of a domain, the narrowing function that leads to the strongest reduction.

Now assume the *relevant* narrowing functions are known (as it will be the case in a cyclic phenomenon), then it is sufficient to compute $\tilde{\mathcal{R}}_i\ (\vec{\mathcal{D}}_i)$ instead of $\tilde{\mathcal{T}}_i\ (\vec{\mathcal{D}}_i)$.

3.5. Computing the relevant dynamic dependencies

As the *non-relevant* narrowing functions will be removed from the cycle, the dynamic dependencies have only to be computed for the *relevant* narrowing functions. Thus, we are now in position to propose a definition of the dynamic dependencies such that:

- most of the cycles can be reduced significantly, and
- the set of those dynamic dependencies can be computed in an efficient way.

[3] If two narrowing functions perform the same reduction on the same bounds, only the first one according to a lexical order is considered as *relevant*.

This dynamic dependency relation is parameterized by the domains of the variables $\vec{\mathcal{D}}$ and by a set of *relevant* narrowing functions $\mathcal{R}$ (intuitively, $\mathcal{R}$ and $\vec{\mathcal{D}}$ define the context in which a narrowing function is applied). The key point is that all narrowing functions in $\mathcal{R}$ are applied on the same domains $\vec{\mathcal{D}}$.

Definition 3.4. [dynamic dependency] Let $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ be a CSP such that $\mathcal{R} \subseteq \mathcal{F}$ be the set of *relevant* narrowing functions, $f \in \mathcal{R}$, $g \in \mathcal{F}$. A dynamic dependency $f \overset{d(\mathcal{R},\vec{\mathcal{D}})}{\longrightarrow} g$ holds iff:

a) $f \in \mathcal{R}$, $f \overset{s}{\rightarrow} g$,
b) $g(\tilde{\mathcal{R}}\ (\vec{\mathcal{D}})) \neq \tilde{\mathcal{R}}\ (\vec{\mathcal{D}})$

$\mathcal{R}$ being the set of *relevant* narrowing functions, $\tilde{\mathcal{R}}\ (\vec{\mathcal{D}}) \subset g(\vec{\mathcal{D}})$. The second problem mentioned in section 3.2 does therefore no longer occur. Point (b) means that the reduction performed by g is due to the domain reduction achieved by some narrowing function in $\mathcal{R}$. Point (a) means that f could be such a narrowing function.

Cycle simplifications based on that definition may not be optimal but in practice this definition is strong enough to significantly reduce numerous cycles.

Computing the dynamic dependencies between the relevant narrowing functions can be done easily thanks to the following result.

Proposition 3.3. Let f and g be two relevant *narrowing functions such that $f \in \mathcal{R}_i$ and $g \in \mathcal{R}_{i+1}$. Then the following proposition holds:*

$$f \overset{d(\mathcal{R}_i,\vec{\mathcal{D}}_i)}{\longrightarrow} g \text{ iff } f \overset{s}{\rightarrow} g$$

Proof: Since every function in $\mathcal{R}_{i+1}$ is *relevant*, g will perform a reduction of a domain, and thus $g(\tilde{\mathcal{R}}_i\ (\vec{\mathcal{D}}_i)) \neq \tilde{\mathcal{R}}_i\ (\vec{\mathcal{D}}_i)$. □

Let G be the dynamic dependency graph. The dynamic dependencies are parameterized by $\mathcal{R}_i$ and $\vec{\mathcal{D}}_i$. The vertices of G are pairs $< f, i >$, where f is a narrowing function and i is the index of the inference step. An arc from $< f, i >$ to $< g, i+1 >$ will represent a dynamic dependency $f \overset{d(\mathcal{R}_i,\vec{\mathcal{D}}_i)}{\longrightarrow} g$.

Let G_i be the subgraph of G restricted to the ith step of the algorithm. G_i is a bipartite graph from $< \mathcal{R}_i, i >$ to $< \mathcal{R}_{i+1}, i+1 >$, where $< \mathcal{R}_i, i >$ is the set $\{< f, i > | f \in \mathcal{R}_i\}$.

The above proposition states that the set of dynamic dependencies represented by G_i is the subset of the static dependencies whose starting functions belong to $\mathcal{R}_i$ and the ending ones belong to $\mathcal{R}_{i+1}$.

The dynamic dependency graph G is just the union of subgraphs G_i at the different steps. An example of a dynamic dependency graph is given in figure 3.2 (a).

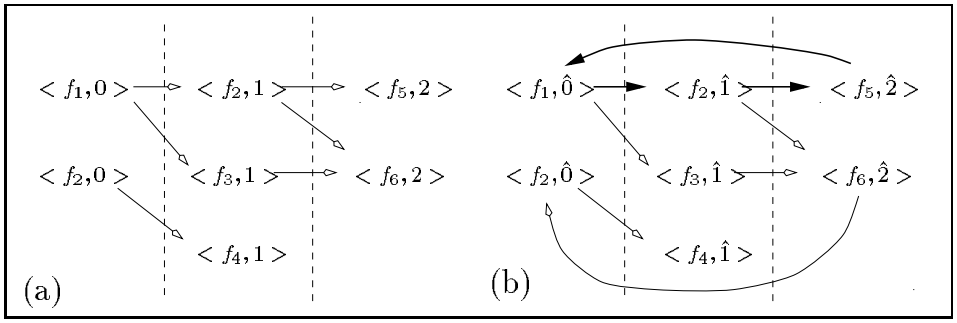

FIGURE 3.2. Dynamic dependency graphs

3.6. Definition of a cyclic phenomenon

A *propagation cycle* formalizes a cyclic phenomenon:

Definition 3.5. [propagation cycle]
A propagation cycle is a quintuplet $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{F}, p, ArrayR >$ where:

- $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ is a CSP; $|\mathcal{F}| = m$;
- $\exists N \gg m, \mathcal{F}^N(\vec{\mathcal{D}}) \neq \mathcal{F}^{N-1}(\vec{\mathcal{D}})$ i.e, a slow convergence[4] occurs;
- p is the period of the cycle;
- For all $i < N, \mathcal{R}_{i+p} = \mathcal{R}_i$ (the sets of *relevant* narrowing functions occur periodically), that is $\mathcal{R}_i = ArrayR[i \ mod \ p]$ if the *relevant* narrowing functions are kept in $ArrayR$.

For instance, a propagation cycle of period 3 means that the subgraph G_i is equal to the subgraph $G_{i \ mod \ 3}$; thus the dynamic dependency graph is cyclic (see figure 3.2 (b) where $\hat{0}$ denotes all the steps i such that $i \ mod \ 3 = 0$).

4. Simplifying a cycle

4.1. Pruning the dynamic dependency graph

Two types of simplifications were mentioned in the introduction:

1. Removing the non-*relevant* narrowing functions;
2. Postponing some narrowing functions.

[4] The speed of convergence is a relative notion. The revised algorithm is said to **converge slowly** for $(\mathcal{X}, \vec{\mathcal{D}}, \mathcal{F})$ when the number of iterations required to reach the fixpoint is much greater than m, the number of narrowing functions of $\mathcal{F}$.

The first point is now interleaved with the cycle definition: removing the non-*relevant* narrowing functions consists in only applying the *relevant* ones that have been identified during the cycle detection step.

The second point can now be formulated easily: a vertex $< f, i >$ which does not have any successor in the dynamic dependency graph corresponds to a narrowing 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 figure 3.2, all white arrows will be pruned.

When a vertex is removed, the corresponding narrowing function is pushed onto a stack (the removing order must be preserved).

The correctness of cycle simplification can trivially be established:

Proposition 4.1. Let $\mathcal{R}'_i \subseteq \mathcal{R}_i$ be the sets of relevant *narrowing functions whose corresponding vertices have not been removed from the graph, and let $s_1, s_2, ..., s_l$ be the stacked narrowing functions (s_1 being the first one stacked).*
If $(\widetilde{\mathcal{F}})^{k+1}(\vec{\mathcal{D}}_0) = (\widetilde{\mathcal{F}})^k(\vec{\mathcal{D}}_0)$, then

$$(\widetilde{\mathcal{F}})^k(\vec{\mathcal{D}}_0) \subseteq s_1(...s_l(\widetilde{\mathcal{R}'}_k\ (\widetilde{\mathcal{R}'}_{k-1}\ (...\ \widetilde{\mathcal{R}'}_0\ (\vec{\mathcal{D}}_0)))))$$

4.2. Algorithm `INC`

The algorithm proposed for cycle simplification is called `INC`. `INC` operates in 4 steps:

1. observe the dynamic behavior and try to detect a cycle;
2. simplify the detected cycle and stack the narrowing functions corresponding to vertices removed from the dynamic dependency graph;
3. iterate on the simplified cycle until a fixpoint is reached;
4. when the fixpoint has been reached, evaluate the stacked narrowing functions.

Step 1 boils down to running algorithm `IN` and observing that it continues to iterate after k iterations where k depends on the number of variables and the number of constraints of the problem. Henceforth, the existence of a propagation cycle is assumed. Then, `Revised-IN` is started for finding the period of the propagation cycle and building *ArrayR*.

To the authors' knowledge, there exists no efficient algorithm for finding the period of the propagation cycle in the general case. However, it is always possible to find the period of a sub-cycle. A history of the *relevant* narrowing functions just needs to be kept: when $ArrayR[k]$ is built, $ArrayR[k]$ and $ArrayR[0]$ need to be compared (implementation is a little more complex since a stabilization step has to be performed). If they are equal, we have a candidate that could be a sub-cycle of period $p = k$. It is then possible to verify that it is repeated during the following k steps. It is difficult to be sure that this sub-cycle is the propagation cycle as it could just be a cycle within the actual propagation cycle. Be this as it may, in most cases it is acceptable to take the first sub-cycle to be encountered.

Step 2 has been described in section 4.1. An upper bound of the running time for simplifying the cycle is $O(q)$ where q is the number of arcs in the dynamic

dependency graph. q is generally of the same order of magnitude as m, the number of narrowing functions[5].

Step 3 consists in computing $\widetilde{\mathcal{R}'}_i\ (\widetilde{\mathcal{R}'}_{i-1}\ (\ldots\ \widetilde{\mathcal{R}'}_0\ (\vec{\mathcal{D}}_0)))$, using the fact that $\mathcal{R}'_i = ArrayR[i\ mod\ p]$. An upper bound of the running time of this iteration procedure is $O(a \times m')$ where m' is the number of different narrowing functions occurring in $ArrayR$ and a is the maximum size of the domain of the variables (note that a is here a very large number). Since the existence of a propagation cycle leads to a phenomenon of slow convergence it is reasonable to suppose that the other parts of the general algorithm represent but a tiny part of the computation time, and $O(a \times m')$ can be compared with the complexity of algorithm `IN` : $O(a \times m)$ where m is the total number of narrowing functions [24, 34, 20].

Step 4 evaluates the *relevant* narrowing functions corresponding to the removed vertices when the fixpoint has been reached. This must be done in reverse order to their removal. This procedure is in $O(l)$ where l is the number of removed vertices.

Since it may happen that `step 1` has only identified a sub-cycle, algorithm `INC` may stop before reaching the fixpoint computed by algorithm `IN`. To make sure that the same fixpoint is computed we can either restart `INC` until no more change occurs or restart `IN` after the fourth step of `INC`.

5. Implementation and experimental results

Algorithm `INC` has been implemented and integrated in Interlog [6, 17, 21], a CLP(Intervals) system. To evaluate the proposed framework, we have performed various experimentations. First subsection reports some experimental results on small examples which very well illustrate the benefits one can expect. Second subsection concerns a real application for which `INC` has led to significant gain in speed. Third subsection shows the advantage of `INC` for computing higher consistencies.

5.1. Examples

The examples in Table 5.1 only differ by an increasing number of narrowing functions that can be postponed. Table 5.2 reports the improvement factor gained with dynamic cycle simplification. *Improvement rate* represents the ratio t_1/t_2 where t_1 is the running time of algorithm `IN` and t_2 is the running time of algorithm `INC` with cycle simplification. Note that even for a problem without any cycle simplification (first problem, for which all enqueued narrowing functions in `IN` are relevant and cannot be postponed) the improvement factor is more than 3 times. This is only due to the fact that using $ArrayR$ is more efficient than the enqueueing/dequeueing operations.

5.2. A chemical problem

The constraint system described in figure 5.1 comes from a chemical problem. On this problem, `IN` enters in a slow convergence phenomenon. `INC` runs more than 7

[5]Note that in examples built for this special purpose q could be a very large number. In this cases, the first step of the algorithm can take into account an upper bound for the number of vertices in the sub-cycle.

TABLE 5.1. List of Examples

Problem	System of Constraints
1	$x = sin(y) \quad y = sin(x)$
2	$x = sin(y) \quad y = sin(x) \quad z_1 = x * y$ $z_2 = x + y$
3	$x = sin(y) \quad y = sin(x) \quad z_1 = x * y$ $z_2 = x + y \quad z_3 = 3 * z_1 \quad z_4 = exp(x)$ $z_5 = z_1 * z_2 \quad z_6 = z_5 \quad z_7 = x + z_1$ $z_8 = y \quad z_9 = z_1 * z_3 \quad z_{10} = y + z_1$ $z_{11} = x + z_6 \quad z_{12} = z_4 + z_1$

TABLE 5.2. Computation Results

Problem	Postponed narrowing functions	Improvement rate
1	0	3.6
2	6	11.7
3	30	27.2

times faster than `IN`. This result is obtained by postponing 28 narrowing functions.

5.3. Higher order consistencies

The constraint system given in figure 5.2 cannot be solved by a 2-B-consistency algorithm (it is already 2-B-consistent). However, this constraint system can be tackled with a 3-B-consistency algorithm. A 3-B-consistency algorithm having to run a 2-B-consistency algorithm, it may be interesting to compare two 3-B-consistency algorithms: the first one uses `IN` while the second one uses `INC`. The version with `INC` runs more than 6 times faster than the 3-B-consistency algorithm with `IN`.

Other experiments on 3B-consistency have been done. Such an improvement factor in using `INC` in place of `IN` in 3B-consistency is not obtained for all the constraint systems, but no overhead was observed on any tested example.

6. Further work

The detection of the cycles is based on an approximation of the dynamic dependencies. The approximation used in this paper has the advantage that it can be computed efficiently. Indeed, both stronger and weaker definitions may allow an effective pruning of the propagation cycles for some specific problems. A topic for future research could be to evaluate experimentally different approximations of the dynamic dependencies on significant benchmarks.

The algorithm suggested here does not detect *the* propagation cycle but a sub-cycle. Although, in the vast majority of cases, this sub-cycle corresponds to the

$$
\begin{array}{l}
Dch = Dav1 + Dt2 + Dt3 + Du1, \\
Dav2 + Dav1 + Dt2 = Du2 + Dp1, \\
Dt3 + Dfg = Dp2 + Du3, \\
Du1 + Du2 + Du3 = Df1 + Dfg, \\
Dfl + Dal = Deg + Dch, \\
Dch * (2.0 * T1 - 4.18 * Tliq + 2400.0) = Pch, \\
2.0 * Dt2 * (T1 - Tt2) = Pt2, \\
2.0 * Dt3 * (T1 - Tt3) = Pt3, \\
Pt2 = 2.0 * Eta2 * Dt2 * (T1 + 273.0) * (1 - exp(0.25 * log(Prt2/P1))), \\
Pt3 = 2.0 * Eta3 * Dt3 * (T1 + 273.0) * (1 - exp(0.25 * log(Prt3/P1))), \\
(Dav1 + Dav2) * (2.0 * Tav + 2400.0) = Dav1 * (2.0 * T1 + 2400.0) + Pav, \\
Pav = (2800.0 + 2.0 * Deltats) * Dav2, \\
Pu1 = Du1 * (2400.0 + 2.0 * T1 - 422.0 * exp(0.25 * log(P1))), \\
Pu2 = Du2 * (2400.0 + 2.0 * T2 - 422.0 * exp(0.25 * log(P2))), \\
Pu3 = Du3 * (2400.0 + 2.0 * T3 - 422.0 * exp(0.25 * log(P3))), \\
Du1 * (2.0 * T1 + 2400.0) + Du2 * (2.0 * T2 + 2400.0) + Du3 * (2.0 * T3 + 2400.0) - \\
Pu1 - Pu2 - Pu3 = Dfl * (4.18 * Tf + 0.3) + Dfg * (2.0 * Tf + 2400.0), \\
Dfl * (Tf - 20.0) = (Dal + Dfl) * (Tliq - 20.0), \\
Tf = 100.0 * exp(0.25 * log(Pf/0.965)), \\
(Dav1 + Dav2) * Tav + Dt2 * Tt2 = (Dav1 + Dav2 + Dt2) * T2, \\
Dt3 * Tt3 + Dfg * Tf = (Dt3 + Dfg) * T3.
\end{array}
$$

FIGURE 5.1. Constraint system of a chemical problem

$$
\begin{array}{l}
x * y + t - 2 * z = 4, \\
x * \sin x + y * \cos t = 0, \\
x - y + \cos^2 z = \sin^2 t, \\
x * y * z = 2 * t. \\
\text{where } D_x = [0, 1000], D_y = [0, 1000], \\
\qquad D_z = [0, 3.1416], D_t = [0, 3.1416].
\end{array}
$$

FIGURE 5.2. Constraint system requiring 3B-consistency

cycle, this is not always the case. One way of tackling this problem consists simply in interrupting the iteration in algorithm `INC` after a certain number of steps (but before reaching the fixpoint), and then to run the algorithm again from step 1. This would also offer two further advantages:

- In an over-constrained problem (which has no solution) the removed vertices may detect a contradiction. It would therefore be useful to periodically apply the narrowing functions corresponding to the removed vertices before reaching the fixpoint.
- Secondly, so far the working hypothesis has been that there is a cyclic phenomenon. In fact, when a phenomenon of slow convergence happens in

algorithm `IN` it is usually, but not always a lonely cyclic phenomenon. As a general rule a phenomenon of slow convergence can be decomposed into a series of cyclic steps separated by a transient, acyclic one. By periodically reinitializing the cycle detection process it should be possible to detect a new cycle and to simplify it.

Using a language with meta-evaluation facilities, table $ArrayR$ could be transformed, before iteration, into explicit code and thus the cycle would really be compiled. Algorithm `Revised-IN` applies the narrowing functions on the same domain vector whereas in algorithm `IN` they are applied sequentially. Once a propagation cycle has been detected and simplified, it is possible to use a sequential iteration procedure (closer to algorithm `IN`). Let $ArrayR[k]$ be the set $\{f_1, ..., f_q\}$, the iteration procedure (step 3) can apply $f_1(...f_q(\vec{\mathcal{D}}))$ instead of $T(\vec{\mathcal{D}})$ where $T = ArrayR[k]$. This leads to another cyclic phenomenon, which could be itself optimized. The order in which the narrowing functions are evaluated can influence this cyclic phenomenon. However, it seems difficult to find an order that is "better" than all the others.

Dynamic cycle simplification is not based upon a specific kind of narrowing functions but on the fixpoint algorithm which is used in almost all interval narrowing systems. The framework introduced in this paper could be combined with some recent advances in the field like [4, 35] and [13], which propose other narrowing functions.

A related work is [36]. Although the problems of cycle detection are quite similar, the aim is not to optimize an algorithm but to generate an abstraction of repeating cycles of processes to perform more powerful reasoning in causal simulation.

7. Conclusion

This paper proposes a method for greatly accelerating the convergence of the cyclic phenomena in algorithm `IN` which is widely used in CLP systems over intervals. The first step requires simplifying this cyclic phenomenon by keeping just the relevant narrowing functions (i.e., the narrowing functions that actually perform the task). The second step consists in removing from the cycle those relevant narrowing functions that may be deferred.

Experimental results indicate that a dynamic cycle simplification can not only produce significant improvements in efficiency over standard interval narrowing, but that it can also boost stronger consistencies algorithms which are often required to achieve an effective pruning of the domains of the variables.

Acknowledgments

Patrick Taillibert gave us invaluable help on preliminary ideas for improving the interval narrowing algorithm. Thanks also to Christian Bliek, Patrice Boizumault, Bernard Botella, Hélène Collavizza, Philippe David, Narendra Jussien, Emmanuel Kounalis, Philippe Marti, Serge Varennes, Dan Vlasie for their helpful comments on earlier drafts of this paper.

REFERENCES

1. H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In C. Beierle and L. Plumer, editors, *Logic Programming: Formal Methods and Practical Applications*. North Holland, 1995. (Studies in Computer Science and Artificial Intelligence, Volume 11).
2. F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, (to appear).
3. C. Bliek. *Computer Methods for Design Automation*. PhD thesis, Massachusetts Institute of Technology, 1992.
4. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(intervals) revisited. In *Logic Programming: Proceedings of the 1994 International Symposium* (MIT Press). pp 109–123, 1994.
5. A. Colmerauer. Spécifications de Prolog IV *Draft*. GIA, Faculte des Sciences de Luminy,163, Avenue de Luminy 13288 MARSEILLE cedex 9 (France), 1994.
6. B. Botella and P. Taillibert. INTERLOG : constraint logic programming on numeric intervals. *3rd International Workshop on Software Engineering* Artificial Intelligence and Expert Systems, Oberammergau, 1993.
7. C.K. Chiu and J.H.M. Lee. Interval Linear Constraint Solving Using the Preconditioned Interval Gauss-Seidel Method. *Twelfth International Conference on Logic Programming*, pages 17–32, MIT Press, Kanagawa, Japan, June, 1995.
8. J.C. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.
9. E. Davis Constraint propagation with interval labels. *Artificial Intelligence*, **32**, 281–331, (1987).
10. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to Industrial and Engineering Problems. In *First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, USA, June 1988.
11. D. Haroud and B. Faltings. Global consistency for continuous constraints. In *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming* (A. Borning, ed.), (Seattle), May 1994.
12. D. Haroud and B. Faltings. Consistency Techniques for Continuous Constraints. *Constraints*, 1(1-2), pp. 85–118, 1996.
13. B. Faltings. Arc consistency for continuous variables. *Artificial Intelligence*, 65(2), pp 363-376, 1994.
14. E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, November 1978.
15. Hoon Hong and Volker Stahl. Safe starting regions by fixed points and tightening. *Computing*, 53:323–335, 1994.
16. E. Hyvönen. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, vol. 58, pp. 71–112, 1992.
17. Dassault Electronique. INTERLOG 1.0 : Guide d'utilisation, DE, 55, quai Marcel Dassault, 92214 Saint Cloud, 1991.
18. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
19. J. H. M. Lee and M. H. van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16:3–4, pp.255–276, 1993.

20. O. Lhomme. Consistency techniques for numeric CSPs. In *Proc. IJCAI93, Chambery, (France)*, pp. 232–238, (August 1993).

21. O. Lhomme. Contribution à la résolution de contraintes sur les réels par propagation d'intervalles', PhD dissertation (in French). Université de Nice — Sophia Antipolis BP 145 06903 Sophia Antipolis, 1994.

22. O. Lhomme and M. Rueher. Application des techniques CSP au raisonnement sur les intervalles. *RIA (Dunod)*, 1996. (to appear).

23. O. Lhomme, A. Gotlieb, M. Rueher and P. Taillibert. Boosting the Interval Narrowing Algorithm in *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming.* MIT Press. pp. 378–392, 1996.

24. A. Mackworth and E. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, vol. 25, pp. 65–73, 1985.

25. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

26. P. Marti and M. Rueher. A distributed cooperating constraints solving system. *IJAIT (International Journal on Artificial Intelligence Tools)* , 4(1-2), 93–113, June 1995.

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

28. R. Moore, *Interval Analysis.* Prentice Hall, 1966.

29. W.J. Older and A. Velino. Extending prolog with constraint arithmetic on real intervals. In *Proc. of IEEE Canadian conference on Electrical and Computer Engineering.* IEEE Computer Society Press, pp. 14.1.1–14.1.4, 1990.

30. W. Older and A. Vellino. Constraint arithmetic on real intervals. In *Constraint Logic Programming: Selected Research*, eds., F. Benhamou and A. Colmerauer. pp 175-195. MIT Press, 1993.

31. M. Rueher. An Architecture for Cooperating Constraint Solvers on Reals. In *Constraint Programming: Basics and Trends.* eds., A. Podelski. pp. 231–250, LNCS 910, Springer Verlag (1995).

32. M. Rueher, C. Solnon. Concurrent Cooperating Solvers within the Reals. *Reliable Computing* ,Kluwer Academic Publishers, Vol.3:3, pp. 325–333, 1997.

33. E. Tsang. Foundations of Constraint Satisfaction. Academic Press, 1993.

34. P. Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, October 1992.

35. P. Van Hentenryck, D. Mc Allester, and D. Kapur. Solving Polynomial Systems Using Branch and Prune Approach. *SIAM Journal (to appear).*

36. D. S. Weld. The use of aggregation in causal simulation. *Artificial Intelligence*, vol. 30, pp. 1–34, 1986.