Boosting the Interval Narrowing Algorithm

Olivier Lhomme
École des Mines de Nantes
4, rue Alfred Kastler, La Chantrerie, 44070 Nantes Cedex 03, France

Arnaud Gotlieb
Dassault Electronique *and* Université de Nice – Sophia Antipolis

Michel Rueher
Université de Nice – Sophia Antipolis I3S-CNRS
Route des colles, BP 145, 06903 Sophia Antipolis, France

Patrick Taillibert
Dassault Electronique
55, Quai Marcel Dassault 92214 Saint-Cloud, France

Abstract

Interval narrowing techniques are a key issue for handling constraints over real numbers in the logic programming framework. However, the standard fixed-point algorithm used for interval narrowing 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. First experimental results show that this approach improves 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 [13] 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 [20], Newton [1], CLP(BNR) [3], Interlog [12, 5, 14], Prolog IV [4]). All these systems use an arc consistency like algorithm [17] adapted for numeric constraints [8, 7].

The standard interval narrowing algorithm has two main limitations :

- the so-called problem of early quiescence [8], i.e., the algorithm stops before reaching a good approximation of the set of possible values. This problem is due to the fact that interval narrowing algorithm guarantees only a partial consistency;
- the problem of the existence of slow convergences, leading to impossibly long response times for certain constraint systems.

Unlike the first problem, for which many algorithms have been proposed [11, 14, 10, 9, 1, 6], the second problem has never been studied in the literature. This paper addresses this second problem. It shows that there is a strong connection between the existence of cyclic phenomena and slow convergence. The aim is to identify cyclic slow convergence phenomena while executing the interval narrowing algorithm and then to simplify them in order to improve performance.

1.1 Motivating example

The interval narrowing algorithm works iteratively: constraints are used for reducing domains until a fixed point is reached. Experimental running times of this algorithm are generally well below the upper bound of the running time as given by a theoretical analysis (i.e., $O(ma)$ where m is the number of constraints and a the size of the largest domain). However, slow — or asymptotic — convergence phenomena sometimes occur, and then the experimental running time approaches the theoretical bound, as in the example described in figure 1.

$$\begin{array}{llllll}
Y = X & (a) & Y = 1.001 * X & (b) & Y = 2 * X & (c) \\
Z_1 = e^Y & (d) & Z_2 = e^{Z_1} & (e) & & \\
\multicolumn{6}{l}{D_X = [0,10] \;\; D_Y = (-\infty,+\infty) \;\; D_{Z_1} = (-\infty,+\infty) \;\; D_{Z_2} = (-\infty,+\infty)}
\end{array}$$

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

etc.

Figure 1: A slow convergence phenomenon

Intuitively these phenomena are cyclic. In the above case 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). The

cycle would thus be simplified to (a, c) and constraints (d, e) would only be applied once the fixed point has been reached. The number of computations carried out by the interval narrowing algorithm 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 fixed point u such that $u = f(u)$. The equation $x = f(x)$ could 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. The equation $x = f(x)$ could also be solved by numeric methods. In particular, methods from interval analysis [19] have the same nice property as interval narrowing: a safe approximation of the set of solutions can be computed. However it is unclear how such methods can be generalized to non-square systems.
Thus the aim of this research is to simplify the equation $x = f(x)$ in order to accelerate convergence towards the fixed point 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 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 interval narrowing algorithm is far more frequent than slow convergence. However, when the interval narrowing algorithm ends prematurely, a kind of enumeration interleaved with this algorithm is generally performed (e.g. domain splitting [7] or stronger consistencies [14]). During this interleaved process, slow convergence phenomena have a great chance to occur and to increase the required computing time considerably.
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 the interval narrowing solver and a linear solver is especially worthwhile in this latter case [6, 22, 18]. For arbitrary non-linear systems, slow convergence very often leads to a cyclic phenomenon. As arbitrary non-linear systems cannot be tackled with a symbolic solver, automatic cycle simplification is the only way to accelerate convergence in the majority of real applications.

1.3 Organization of the paper

Section 2 introduces some basic definitions. In section 3, the concept of propagation cycle is introduced. This section shows that the standard interval narrowing algorithm will not allow cyclic phenomena to be satisfactorily simplified. Thus, a revised interval narrowing algorithm is proposed in which cyclic phenomena can be significantly simplified. Such a simplification of a cycle is proposed in section 4. In section 5, first experimental results are provided. Finally, in section 6, the limits of such an approach are discussed.

2 Interval narrowing

2.1 Basic notations and definitions

- $\overline{\mathbb{F}}$ denotes the set of floating-point numbers augmented with the two symbols $\{-\infty, +\infty\}$ which represents respectively all numbers smaller (resp. greater) than the smallest (resp. the biggest) floating-point number;
- $\mathcal{I}(\overline{\mathbb{F}})$ denotes the set of intervals $[a, b]$ where $a, b \in \overline{\mathbb{F}}$;
- A CSP [17] 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 i^{th} component of $\vec{\mathcal{D}}$ being the domain of x_i, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ denotes a set of constraints.
  This paper concentrates only on CSPs whose domains are intervals over the floating-point numbers, i.e., $\vec{\mathcal{D}} \in \mathcal{I}(\overline{\mathbb{F}})^n$ [8, 11, 14, 10, 1];
- A k-ary constraint C is a relation over the reals (i.e. a subset of $\mathbb{R}^k$). *appx*(C) denotes the smallest (w.r.t. inclusion) subset of $\mathcal{I}(\overline{\mathbb{F}})^k$ which contains C (we consider as in [13, 1] that results of floating-points operations are outward-rounded to preserve correctness of the computation).

2.2 Narrowing functions

The interval narrowing algorithm uses an approximation of the unary projection of the constraints to reduce the domains of the variables.
Let C be a k-ary constraint over $(x_{i_1}, \ldots, x_{i_k})$, and $(I_1, \ldots, I_k) \in \mathcal{I}(\overline{\mathbb{F}})^k$: for each j in $1..k$, $\pi_{i_j}(C, I_1 \times \ldots \times I_k)$ denotes the projection of *appx*(C) on x_{i_j} in the part of the space delimited by $I_1 \times \ldots \times I_k$, i.e.,

$$\pi_{i_j}(C, I_1 \times \ldots \times I_k) = \{a_j \mid \exists (a_1, \ldots, a_k) \in appx(C) \cap I_1 \times \ldots \times I_k\}$$

$AP_i(C, I_1 \times \ldots \times I_k)$ denotes an approximation of the projection of a constraint equal to the smallest interval encompassing the projection:

$$[\inf\ \pi_i(C, I_1 \times \ldots \times I_k),\ \sup\ \pi_i(C, I_1 \times \ldots \times I_k)]$$

Such an approximation[1] 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 $\mathcal{I}(\overline{\mathbb{F}})^n$ to $\mathcal{I}(\overline{\mathbb{F}})^n$. For a k-ary constraint C over $(x_{i_1}, \ldots, x_{i_k})$ there are k narrowing functions, one for each x_i where $i \in \{i_1, \ldots, i_k\}$. The narrowing function of C over the variable x_i is the function $f : \mathcal{I}(\overline{\mathbb{F}})^n \longrightarrow \mathcal{I}(\overline{\mathbb{F}})^n$ defined as $f(\vec{\mathcal{D}}) = \vec{\mathcal{D}'}$ such that :

- $D'_i = AP_i(C, D_{i_1} \times \ldots \times D_{i_k})$;
- $j \in \{1, ..., n\}, i \neq j \Longrightarrow D'_j = D_j$ (except the i^{th} domain, all domains of $\vec{\mathcal{D}'}$ and $\vec{\mathcal{D}}$ are identical)

A narrowing function f may reduce the domain of only one variable (x_i in the above definition), called left-variable of f and denoted $f.y$. 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 $f.x_s$.

Properties 2.1. *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 shows such a view of a CSP ($\Pi_j(C)$ denotes the narrowing function of C over the variable x_j; thus $f = \Pi_1(a)$ reduces D_1 based on D_2).

Let $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{C} >$ be a CSP where $\mathcal{C} = \{a, b\}$:

(a) $x_1 - x_2 + 3 = 0$

(b) $x_3 = x_1$

This CSP can be formulated in the form $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{F} >$ where $\mathcal{F} = \{f, g, h, i\}$

$f = \Pi_1(a)$ $\quad h = \Pi_2(a)$ $\quad i = \Pi_1(b)$ $\quad g = \Pi_3(b)$

Figure 2: A CSP in the form $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{F} >$

Using the above notations, the standard interval narrowing algorithm [8, 7] can be written as in figure 3.

In the rest of this paper, a set of narrowing functions $\mathcal{T}$ will be associated to a filtering operator $\tilde{\mathcal{T}}$ that computes the intersection of the domains narrowed by the functions in $\mathcal{T}$: Let $\mathcal{T} = \{f_1 \ldots, f_p\} \subset \mathcal{F}$, $\tilde{\mathcal{T}}(\vec{\mathcal{D}})$ is defined by $f_1(\vec{\mathcal{D}}) \cap \ldots \cap f_p(\vec{\mathcal{D}})$. If $\mathcal{T} = \emptyset$ then by convention $\tilde{\mathcal{T}}(\vec{\mathcal{D}}) = \vec{\mathcal{D}}$.

Just note that if $\vec{\mathcal{D}'}$ is the fixed point reached by the interval narrowing algorithm then $\vec{\mathcal{D}'} = \tilde{\mathcal{F}}(\vec{\mathcal{D}'})$.

```
IN-1(in $\mathcal{F}$, inout $\vec{\mathcal{D}}$)
    Queue $\leftarrow \mathcal{F}$ ;
    while Queue $\neq \emptyset$
        $f \leftarrow$ POP Queue;   $\vec{\mathcal{D}}' \leftarrow f(\vec{\mathcal{D}})$;
        if $\vec{\mathcal{D}}' \neq \vec{\mathcal{D}}$ then   $\vec{\mathcal{D}} \leftarrow \vec{\mathcal{D}}'$;
                                  Queue $\leftarrow$Queue $\cup \{g \in \mathcal{F} \mid g.c \neq f.c \textit{ and } f.y \in g.x_s\}$
        endif
    endwhile
```

Figure 3: Interval narrowing algorithm

3 Towards a characterization of the cyclic phenomenon

When the interval narrowing algorithm runs into a slow convergence phenomenon a cyclic phenomenon may occur after a transient period. In this section, we give a precise definition of a cyclic phenomenon. Further definitions are now required to formalize such cyclic phenomena. 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) such information about dynamic dependencies cannot be identified in the framework of the `IN-1` algorithm. This is due to the fact that the order in which the narrowing functions are en-queued plays a major role in `IN-1`;

(3) in order to get information about some dynamic dependencies we introduce a revised version of the `IN-1` algorithm.

3.1 Static dependency

A static dependency between two narrowing functions f and g — denoted by $f \overset{s}{\rightarrow} g$ — means that after an evaluation of f which modifies the domain of $f.y$, g may reduce the domain of $g.y$ (the narrowing functions en-queued in interval narrowing algorithm are the ones which statically depend on f).

Definition 3.1. *(static dependency) A static dependency $f \overset{s}{\rightarrow} g$ holds 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 $\mathcal{T}$: $succ_s(\mathcal{T}) = \{g \in \mathcal{F} \mid \exists f \in \mathcal{T} \land f \overset{s}{\rightarrow} g\}$.
Static dependency information may not be sufficient for cycle simplification. For instance, consider the example in figure 2: $f \overset{s}{\rightarrow} g$, i.e., $g(f(\vec{\mathcal{D}}))$ 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, $f \overset{s}{\rightarrow} g$ 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 $f \overset{d}{\rightarrow} g$ that ensures that a modification induced by f actually implies a modification induced by g. The first idea is to follow interval narrowing algorithm and try to identify such dynamic dependencies.

3.2 Dynamic dependency

Algorithm `IN-1` computes the terms of a sequence of i^{th} term $f_i(f_{i-1}(\ldots f_0(\overrightarrow{\mathcal{D}})))$ characterizing the order in which the narrowing functions f_j are en-queued: $f_i(f_{i-1}(\ldots f_0(\overrightarrow{\mathcal{D}})))$ corresponds to the en-queueing order $(f_0, f_1, \ldots, f_i)$. Let us assume that a dynamic dependency holds between f and g if $f \overset{s}{\rightarrow} g$ and $g(f(\overrightarrow{\mathcal{D}})) \neq f(\overrightarrow{\mathcal{D}})$. Such a definition would lead to several problems:

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

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

(3) The narrowing functions from which g dynamically depends may be dynamically dependent between themselves, meaning that the dependencies are interleaved.

Example

Let $< \mathcal{X}, \overrightarrow{\mathcal{D}}, \mathcal{F} >$ be a CSP where :
- $\mathcal{F} = \{f, g, h\}$ • $D_4 = [0, 2\pi]$.
- $f = \Pi_1(x_1 = x_9)$ • $g = \Pi_2(x_2 = x_1)$ • $h = \Pi_3(x_3 = x_2 + cos(x_4 + x_1))$

Suppose that $h(g(f(\overrightarrow{\mathcal{D}})))$ is computed (according to en-queueing order of the narrowing functions). Suppose also that $\overrightarrow{\mathcal{D}}$ verifies:
- $g(\overrightarrow{\mathcal{D}}) = \overrightarrow{\mathcal{D}}$ and $h(\overrightarrow{\mathcal{D}}) = \overrightarrow{\mathcal{D}}$, • $f(\overrightarrow{\mathcal{D}}) \neq \overrightarrow{\mathcal{D}}$,
- $g(f(\overrightarrow{\mathcal{D}})) \neq f(\overrightarrow{\mathcal{D}})$, • $h(g(f(\overrightarrow{\mathcal{D}}))) \neq g(f(\overrightarrow{\mathcal{D}}))$.

That is f, g and h perform a reduction. The static dependencies are: $f \overset{s}{\rightarrow} g, f \overset{s}{\rightarrow} h, g \overset{s}{\rightarrow} h$. According to the above naive definition, $f \overset{d}{\rightarrow} h$ and $g \overset{d}{\rightarrow} h$ hold. However h actually depends only on g (the reduction of D_3 is only due to the modification of D_2 computed by g).

Identifying dynamic dependencies which allow optimal cycle simplifications would require considering a great number of permutations of narrowing functions in the queue, and thus, would be far too expensive to be computed. What is proposed here is a definition of the dynamic dependencies such that:
- most of the cycles can be reduced significantly,

- the set of dynamic dependencies can be computed in an efficient way.

A dynamic dependency is parameterized by the domains of the variables $\vec{\mathcal{D}}$ and by a set of narrowing functions $\mathcal{T}$ (whose meaning will be made clear in the next subsection).

Definition 3.2. *(dynamic dependency)*

A dynamic dependency $f \overset{d(\mathcal{T},\vec{\mathcal{D}})}{\longrightarrow} g$ *holds iff*

- $f \in \mathcal{T}$, $f \overset{s}{\rightarrow} g$,
- $g(\tilde{\mathcal{T}}(\vec{\mathcal{D}})) \neq \tilde{\mathcal{T}}(\vec{\mathcal{D}})$
  (intuitively g reduces a domain due to a narrowing function in $\mathcal{T}$).

3.3 Revised algorithm for interval narrowing

We reformulate the interval narrowing algorithm such that the dynamic dependencies can be computed in an efficient way. The revised version (Figure 4) applies on the same vector $\vec{\mathcal{D}}$ all the narrowing functions which may reduce a domain. This will make it possible to find the dynamic dependencies.
The fixed point towards which the revised algorithm[2] converges is identical to that of the standard algorithm (i.e., a domain vector $\vec{\mathcal{D}'}$ such that $\tilde{\mathcal{F}}(\vec{\mathcal{D}'}) = \vec{\mathcal{D}'}$).

```
Revised-IN-1(in $\mathcal{F}$, inout $\vec{\mathcal{D}}$)
    $\mathcal{T} \leftarrow \mathcal{F}$ ;
    while $\mathcal{T} \neq \emptyset$
        $\vec{\mathcal{D}'} \leftarrow \vec{\mathcal{D}}$;   $\vec{\mathcal{D}} \leftarrow \tilde{\mathcal{T}}(\vec{\mathcal{D}})$;
        $\delta \leftarrow \{f \in \mathcal{T} \mid D'_i \neq D_i$ and $f.y = x_i\}$;
        $\mathcal{T} \leftarrow succ_s(\delta)$;
```

Figure 4: Revised interval narrowing algorithm

The revised interval narrowing algorithm computes the terms of a sequence of n^{th} term $\tilde{\mathcal{T}}_n(\tilde{\mathcal{T}}_{n-1}(\ldots(\tilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0))))$ where

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

Let also $\vec{\mathcal{D}}_i$ be the domain vector at the i^{th} step: $\vec{\mathcal{D}}_i = \tilde{\mathcal{T}}_{i-1}(\ldots\tilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0))$.
Of course, $\tilde{\mathcal{T}}_i(\vec{\mathcal{D}}_i) = \tilde{\mathcal{F}}(\vec{\mathcal{D}}_i)$ and so $\tilde{\mathcal{T}}_i(\ldots\tilde{\mathcal{T}}_0(\vec{\mathcal{D}}_0)) = \tilde{\mathcal{F}}^{i+1}(\vec{\mathcal{D}}_0)$.

3.4 *Relevant* narrowing functions

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

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

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

Computing $\mathcal{R}_i$ only consists — when applying $\widetilde{\mathcal{T}}_i$ in `Revised-IN-1` — in keeping, for each bound of a domain, the narrowing function that leads to the strongest reduction.
In a cyclic phenomenon, the *relevant* narrowing functions will be *a priori* known and then it will be sufficient to compute $\widetilde{\mathcal{R}}_i\,(\vec{\mathcal{D}_i})$ in place of $\widetilde{\mathcal{T}}_i\,(\vec{\mathcal{D}_i})$.

3.5 Computing the dynamic dependencies

As the *non-relevant* narrowing functions will be removed from the cycle, the dynamic dependencies have to be computed only for the *relevant* narrowing functions.
Let G be the dynamic dependency graph. The dynamic dependencies are functions of $\mathcal{R}_i$ and $\vec{\mathcal{D}_i}$. The vertices are some 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 which is only concerned with the i^{th} step of the algorithm. G_i is a bipartite graph from $< \mathcal{R}_i, i >$ to $< \mathcal{R}_{i+1}, i+1 >$.
A function g in $\mathcal{R}_{i+1}$ being *relevant* it performs a reduction of a domain. Thus there is an arc from $< f, i >$ to $< g, i+1 >$ iff $f \overset{s}{\rightarrow} g$. Then, the set of dynamic dependencies represented by G_i is the subset of the static dependencies whose starting functions are in $\mathcal{R}_i$ and the ending ones are in $\mathcal{R}_{i+1}$.
The dynamic dependency graph G is just the union of its subgraphs G_i at the different steps. An example of a dynamic dependency graph is given in figure 5 (a).

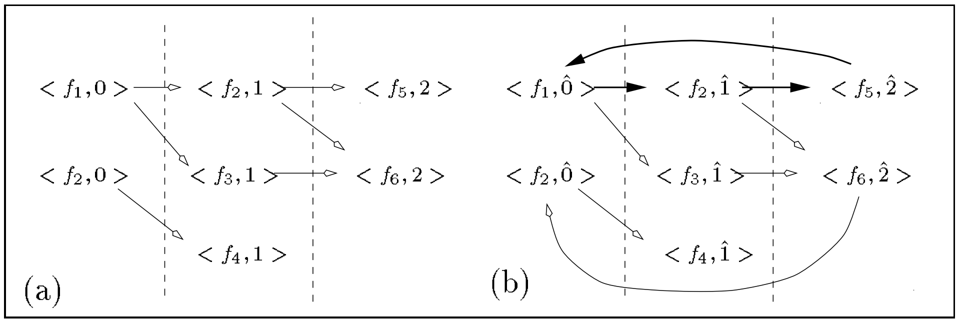

Figure 5: Dynamic dependency graphs

3.6 Definition of a cyclic phenomenon

A *propagation cycle* formalizes a cyclic phenomenon:

Definition 3.4. *A propagation cycle is a quintuple* $< \mathcal{X}, \overrightarrow{\mathcal{D}}, \mathcal{F}, p, ArrayR >$ *where:*

- $< \mathcal{X}, \overrightarrow{\mathcal{D}}, \mathcal{F} >$ *is a CSP;* $|\mathcal{F}| = m$*;*
- $\exists N >> m, \mathcal{F}^N(\overrightarrow{\mathcal{D}}) \neq \mathcal{F}^{N-1}(\overrightarrow{\mathcal{D}})$ *i.e, a slow convergence*[4] *occurs;*
- p *is the period of the cycle;*
- *let* $\mathcal{R}_i$ *be the* relevant *narrowing functions at the step* i*,*
  then $\forall i < N, \mathcal{R}_{i+p} = \mathcal{R}_i$ *,i.e., the sets of* relevant *narrowing functions re-occurs periodically;*
- $ArrayR[i \; mod \; p] = \mathcal{R}_i$ *(the* relevant *narrowing functions are kept in* $ArrayR$*).*

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 5 (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.

The first one is now interleaved with the cycle definition (where the $\mathcal{R}_i$ sets have to be known).

The second one 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 recursively will remove all the non-cyclic paths from the graph. For instance, in the graph (b) of the figure 5, the 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). Let $\mathcal{R}'_i \subseteq \mathcal{R}_i$ be the sets of narrowing functions whose corresponding vertices have not been removed from the graph, let $s_1, s_2, ..., s_l$ be the stacked narrowing functions (s_1 being the first stacked one); then it can be shown that

$$s_1(s_2(...s_l(\widetilde{\mathcal{R}'}_i \; (\widetilde{\mathcal{R}'}_{i-1} \; (... \; \widetilde{\mathcal{R}'}_0 \; (\overrightarrow{\mathcal{D}}_0)))))) = \widetilde{\mathcal{R}}_i \; (\widetilde{\mathcal{R}}_{i-1} \; (... \; \widetilde{\mathcal{R}}_0 \; (\overrightarrow{\mathcal{D}}_0)))) = \widetilde{\mathcal{F}}^i \; (\overrightarrow{\mathcal{D}}_0)$$

4.2 `IN-2` algorithm

The algorithm proposed for cycle simplification is called `IN-2`. `IN-2` 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 a removed vertex in the dynamic dependency graph;

(3) iterate on the simplified cycle until a fixed point is reached;

(4) when the fixed point has been reached, evaluate the stacked narrowing functions.

Step 1 boils down to running the `IN-1` 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-1` 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 for 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 $O(v)$ where v is the number of vertices in the dynamic dependency graph. Note that in examples built for this special purpose v could be a very large number. However, v is generally of the same order of magnitude as m, the number of narrowing functions. The first step of the algorithm can take into account an upper bound for the number of vertices in the sub-cycle.

Step 3 consists in computing $\widetilde{\mathcal{R}'}_i\ (\widetilde{\mathcal{R}'}_{i-1}\ (\ldots\ \widetilde{\mathcal{R}'}_0\ (\overrightarrow{\mathcal{D}_0})))$, using the fact that $\mathcal{R}'_i = ArrayR[i \ mod \ p]$. An upper bound of the running time of this iteration procedure is $O(a * m')$ 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 $O(a * m')$ can be compared with the complexity of the standard interval narrowing algorithm: $O(a * m)$ where m is the total number of narrowing functions[16, 23, 14].

Step 4 evaluates the *relevant* narrowing functions corresponding to the removed vertices when the fixed point of the interval narrowing algorithm 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 the removed vertices.

Table 1: List of Examples

Problem	System of Constraints
1	$x_1 = sin(x_2)$ $x_2 = x_1$
2	$x_1 = sin(x_2)$ $x_3 = x_1 * x_2$ $x_2 = x_1$
3	$x_1 = sin(x_2)$ $x_3 = exp(x_1)$ $x_4 = exp(x_3)$ $x_2 = x_1$
4	$x_1 = sin(x_2)$ $x_3 = exp(x_1)$ $x_4 = exp(x_3)$ $x_5 = 3 * x_4$ $x_6 = exp(x_4)$ $x_7 = x_6$ $x_8 = 5 * x_7$ $x_9 = x_7$ $x_{10} = 4 * x_9$ $x_2 = x_1$

Table 2: Computation Results

Problem	Postponed narrowing functions	Improvement rate
1	0	3.4
2	1	6.2
3	2	12
4	8	32.3

5 Implementation & experimental results

The `IN-2` algorithm has been implemented and integrated in Interlog [12, 15], a CLP(Intervals) system. The standard interval narrowing algorithm is always restarted after the four steps of `IN-2` in order to make sure that the same fixed point is computed.
The examples in Table 1 only differ by an increasing number of narrowing functions that can be postponed. Table 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 the standard interval narrowing `IN-1` and t_2 is the running time of the revised interval narrowing with cycle simplification `IN-2`. Note that even for a problem without any cycle simplification (first problem) the improvement factor is more than 3 times. This is only due to the fact that the en-queueing/de-queueing operations are no longer performed.

6 Discussion

First of all, let us not forget that the algorithm suggested here does not detect a propagation cycle but a sub-cycle. Although, in the vast majority of cases, this sub-cycle corresponds to the cycle, this is not always the case. One way of tackling this problem is simply to interrupt the iteration in the `IN-2` algorithm after a certain number of iterations (but before reaching the fixed point), then to run the algorithm again from step 1. This approach offers two further advantages:

- In an over-constrained problem (which has no solution) the removed vertices may possibly detect a contradiction. It may therefore be useful to periodically re-apply the narrowing functions corresponding to the removed vertices before reaching the fixed point.

- Secondly, so far the working hypothesis has been that there is a cyclic phenomenon. In fact, when a phenomenon of slow convergence happens in the interval narrowing algorithm it is usually, but not always a single cyclic. 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 where meta-evaluation is authorized, before iteration it is possible to transform the table $ArrayR$ into explicit code and thus the cycle would really be compiled. The revised model of the interval narrowing algorithm applies the narrowing functions on the same domain vector whereas in the standard interval narrowing algorithm they are applied sequentially. Once a propagation cycle has been detected and simplified, it is possible to use a sequential iteration procedure (closer to the standard algorithm). Let $ArrayR[k]$ be the set $\{f_1, ..., f_q\}$, the iteration procedure (step 3) can apply $f_1(...f_q(\overrightarrow{\mathcal{D}}))$ instead of $T(\overrightarrow{\mathcal{D}})$ 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 fixed point algorithm which is used in almost all interval narrowing systems. The approach could then be combined with some recent advances in the field like [1] and [10], which propose other narrowing functions.

A related work is [24]. 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 the interval narrowing algorithm. 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 consist in removing from the cycle those relevant narrowing functions that may be deferred. In this way it is possible to reduce the theoretical upper bound of the running time from $O(am)$ to $O(am')$, where m is the total number of narrowing functions, m' is the number of relevant narrowing functions occurring in the cycle and a is the maximum size of the domains of the variables.

In order to enable the simplification of the propagation cycles, a revised interval narrowing algorithm has been introduced.
First experimental results indicate that an automatic cycle simplification can produce significant improvements in efficiency over standard interval narrowing.

Acknowledgements

Thanks to Christian Bliek, Patrice Boizumault, Bernard Botella, Helene Collavizza, Narendra Jussien, Emmanuel Kounalis, Philippe Marti, Peter Sander, Serge Varennes, Dan Vlasie and the reviewers for their helpful comments.

Notes

[1]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 [19] allows it to be computed on a subset of the constraints set, called basic constraints. Each constraint can be approximated by decomposition in basic constraints [14]. 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. Any other approximation of the projection (e.g. [1]) could have been taken in place of this one.

[2] The theoretical complexity of the revised version is higher than that of the standard algorithm. However this algorithm will not be used for computing the fixed point but only for catching the dynamic dependencies. For information, an upper bound of the running time is in $O(n * m * a)$ instead of $O(m * a)$ where m is the number of constraints, n is the number of variables and a the size of the largest domain.

[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*.

[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 fixed point is much greater than m (the number of narrowing functions of $\mathcal{F}$), i.e, when $\exists N >> m, \forall k < N, \mathcal{F}^{k+1}(\vec{\mathcal{D}}) \neq \mathcal{F}^k(\vec{\mathcal{D}})$.

References

[1] F. Benhamou, D. Mc Allester, and P. Van Hentenryck, 'CLP(Intervals) Revisited', in *Proc. Logic Programming: Proceedings of the 1994 International Symposium*,MIT Press, (1994).

[2] F. Benhamou, 'Interval Constraint Logic Programming', in in [21]

[3] F. Benhamou and W. Older, 'Applying interval arithmetic to real, integer and boolean constraints', *Journal of Logic Programming,* (1994).

[4] A. Colmerauer, 'Spécifications de Prolog IV', *Draft*, 1994.

[5] 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.

[6] C. K. Chiu and J. H. M. Lee, 'Towards Practical Interval Constraint Solving in Logic Programming', in *ILPS'94: Proceedings 11th International Logic Programming Symposium*, (Ithaca), (1994).

[7] J. Cleary, 'Logical arithmetic,' *Future Computing Systems*, vol. 2, no. 2, pp. 125–149, 1987.

[8] E. Davis, 'Constraint propagation with interval labels', *Artificial Intelligence*, **32**, 281–331, (1987).

[9] 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.

[10] B. Faltings, 'Arc consistency for continuous variables,'*Artificial Intelligence*, 65(2), 1994.

[11] E. Hyvőnen, 'Constraint reasoning based on interval arithmetic: the tolerance propagation appoach', *Artificial Intelligence*, vol. 58, pp. 71–112, 1992.

[12] Dassault Electronique, 'INTERLOG 1.0 : Guide d'utilisation',DE, 55, quai Marcel Dassault, 92214 Saint Cloud, 1991.

[13] 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.

[14] O. Lhomme, 'Consistency techniques for numeric CSPs', in *Proc. IJCAI93, Chambery, (France)*, pp. 232–238, (August 1993).

[15] O. Lhomme, 'Contribution à la résolution de contraintes sur les réels par propagation d'intervalles', PhD dissertation, 1994. Université de Nice — Sophia Antipolis BP 145 06903 Sophia Antipolis

[16] 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.

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

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

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

[20] W. Older and A. Vellino, 'Constraint arithmetic on real intervals', in *Constraint Logic Programming: Selected Research*, eds., Frédéric Benhamou and Alain Colmerauer. MIT Press, (1993).

[21] Andreas Podelski,*Constraint Programming: Basics and Trends*, LNCS 910, 231–250, Springer Verlag (1995). (Châtillon-sur-Seine Spring School, France, May 1994).

[22] M. Rueher, '*An Architecture for Cooperating Constraint Solvers on Reals*', in [21]

[23] 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.

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