Exploiting Common Subexpressions in Numerical CSPs

Ignacio Araya, Bertrand Neveu, and Gilles Trombettoni

INRIA, Université de Nice-Sophia, Certis
{Ignacio.Araya,Bertrand.Neveu,Gilles.Trombettoni}@sophia.inria.fr

Abstract. It is acknowledged that the symbolic form of the equations is crucial for interval-based solving techniques to efficiently handle systems of equations over the reals. However, only a few automatic transformations of the system have been proposed so far. Vu, Schichl, Sam-Haroud, Neumaier have exploited common subexpressions by transforming the equation system into a unique directed acyclic graph. They claim that the impact of common subexpressions elimination on the gain in CPU time would be only due to a reduction in the number of operations.

This paper brings two main contributions. First, we prove theoretically and experimentally that, due to interval arithmetics, exploiting certain common subexpressions might also bring additional filtering/contraction during propagation. Second, based on a better exploitation of n-ary plus and times operators, we propose a new algorithm I-CSE that identifies and exploits *all* the "useful" common subexpressions. We show on a sample of benchmarks that I-CSE detects more useful common subexpressions than traditional approaches and leads generally to significant gains in performance, of sometimes several orders of magnitude.

1 Introduction

Granvilliers et al. [9] show in a survey several ways to combine symbolic and interval methods to improve performance of solvers. They noticed that Gröbner basis computation [3] introduces redundancies that often improve the pruning effect of interval techniques. The use of several forms of the equations together in the same system (e.g., the natural and centered forms) has the same effect.

The presence of multiple occurrences of the same variable in a given equation is well-known to lower the power of interval arithmetics [17]. Thus, several practitioners apply by hand symbolic transformations of their systems, such as factorizations, to limit the number of occurrences of variables [9,15].

Common subexpression elimination (CSE) is an important feature of compiler optimization [16]. CSE searches in the code for common subexpressions with identical evaluation and replaces them by auxiliary variables. It generally fasten the program by decreasing the number of instructions. Symbolic tools like Mathematica [2] or Maple [11] represent equations by directed acyclic graphs (DAGs), where nodes with several parents correspond to *common subexpressions* (CSs). This decreases the number of evaluations and also stores all the expressions with

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 342–357, 2008.

less memory. Ceberio and Granvilliers in [4] use Gaussian elimination to reduce the number of non-linear terms in equations. As a side effect, their algorithm identifies some CSs.

Following the representation used by symbolic tools, Schichl and Neumaier have proposed a unique DAG to represent a system of equations handled by interval analysis techniques [18]. CSs are the nodes of the DAG with several parents and the main interval analysis operators are redefined on this data structure: evaluation of functions, computation of derivatives, etc. Vu, Schichl and Sam-Haroud have described in [20] how to carry out propagation in the DAG. In particular, an interval is attached to internal nodes and the propagation is performed in a sophisticated way: two queues are managed, one for the evaluation, the other for the narrowing/propagation (see below), and the top-down narrowing operations have priority over the bottom-up evaluation. All the researchers who have exploited CSs manually or automatically [9,20] think that the gain in performance due to common subexpressions would be only implied by a reduction of the number of operations.

The first good news is that CSE in interval analysis might bring *a stronger contraction/filtering power*. Section 3 clearly states which types of CSs are useful for bringing additional filtering. Section 4 presents a new algorithm `I-CSE` (Interval CSE) to detect CSs and generate a new system of equations. For a given form of the equations, `I-CSE` is able to find *all* the "useful" CSs, because it finds all the n-ary maximal CSs corresponding to sums and products, modulo the commutativity and the associativity of these operators, including overlapping CSs. In addition, `I-CSE` is not intrusive in that it produces a new system that can be handled by any interval solver using a classical propagation scheme. Finally, experiments shown in Section 6 highlight that the CSs are extracted very quickly. The new system of equations then leads solving algorithms using `HC4` to significant gains in performance (of sometimes several orders of magnitude).

2 Background

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 1. *A* **numerical CSP** *(NCSP)* $P = (V, C, B)$ *contains a set of constraints* C *and a set* V *of* n *variables. Every variable* $x_i \in V$ *can take a real value in the interval* X_i *and* $\mathbf{B}$ *is the cartesian product (called a* **box***)* $X_1 \times ... \times X_n$*. A solution to* P *is an assignment of the variables in* V *satisfying all constraints in* C*.*

Finding all the solutions to an NCSP follows a scheme analogous to branch and prune for CSPs. *Branch:* Bisections divide the domain of one variable into two sub-domains in a combinatorial way. *Prune:* Two types of algorithms are used. Algorithms from interval analysis, like interval Newton [17], contract/filter the current box in all the dimensions simultaneously and can often guarantee that a box contains a unique solution. Algorithms from constraint programming are

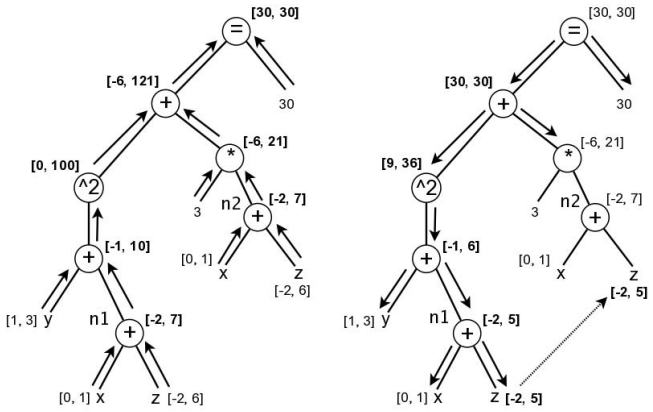

Fig. 1. Evaluation and narrowing in the `HC4-revise` algorithm. The tree represents the constraint: $(x + y + z)^2 + 3(x + z) = 30$.

also useful. `HC4` follows a propagation loop like that of AC3 and handles the constraints individually with a procedure `HC4-revise` that removes inconsistent values on the bounds of intervals [1,12]. Stronger consistencies like 3B [13], similar to SAC [7] for finite domain CSPs, often obtain a better performance. At the end, the solving algorithm finds an approximation of all the solutions of the NCSP. The algorithm `HC4-revise` uses a tree representation of one constraint, where leaves are constants or variables, and internal nodes correspond to *primitive operators* like `+`, $\times$, `sinus`. An interval is associated with every node. `HC4-revise` works in two phases. The *evaluation* phase is performed bottom-up from the leaves (variables and constants) to the root. Using the *natural extension* of primitive functions, this phase evaluates the intervals of the sub-expressions represented by the tree nodes (see Fig. 1-left). The *narrowing* phase traverses the tree top-down from root to leaves and applies in every node a narrowing operator (also called projection; see Fig. 1-right). The narrowing operator contracts the intervals of the nodes eliminating inconsistent values w.r.t. the corresponding unary or binary primitive operator. In Fig. 1, the intervals in bold have been narrowed. If an empty interval is obtained during the narrowing phase, this means that the constraint is inconsistent w.r.t. the initial domains. The intervals computed in the internal nodes are not stored from one call to `HC4-revise` to another, as opposed to the intervals of the leaves (i.e., the variables).

3 Properties of HC4 and CSE

We call *Common Subexpression* (in short CS) a numerical expression that occurs several times in one or several constraints.

If we observe carefully the `HC4-revise` algorithm, we can note that the contraction obtained by a narrowing operator on a given expression f is in general partially lost in the next evaluation of f. Consider for instance a sum $x + z$

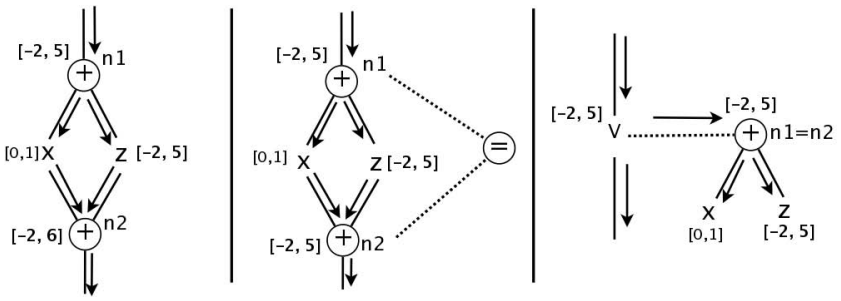

Fig. 2. Narrowing/Evaluation without and with CSE

that is shared by two expressions n_1 and n_2. Following Fig. 1, the narrowing phase of `HC4-revise` applied to n_1 contracts its interval to $[-2, 5]$. Then, when the evaluation phase of `HC4-revise` applies to n_2, its interval is set to $[-2, 6]$ (Fig. 2–left). Clearly, the interval of n_2 is larger than that of n_1. To avoid this loss of information, the idea is to replace n_1 and n_2 by a common variable v, and to add a new constraint $v = x + z$. The new system is equivalent to the original one (both have the same solutions) while it improves the contraction power of HC4. The introduction of v (Fig. 2–right) amounts to adding a redundant equation $n_1 = n_2$ (Fig. 2–center). If one applies evaluation and narrowing phases of `HC4-revise` until the fixed-point on the new system, one will obtain the interval $[0.19, 4.14]$ for z, instead of $[-0.36, 4.58]$.

3.1 Additional Propagation

Proposition 1 underlines that `HC4` might obtain a better filtering when new auxiliary variables and equations corresponding to CSs are added in the system.

Proposition 1. *Let S be a NCSP and S' be the NCSP obtained by replacing in S one CS f in common between two expressions (belonging to constraints in S) by an auxiliary variable v, and by adding the new equation $v = f$. Then,* `HC4` *(with a floating-point precision) applied to S' produces a contracted box B' that is smaller than or equal to the box B produced by* `HC4` *applied to S.*

Proof. One first produces a system S_1 by replacing in S the first occurrence of f by an auxiliary variable v_1 and the second one by v_2. We add the equations $v_1 = f$ and $v_2 = f$. Because `HC4-revise` works on acyclic graphs, `HC4` computes the 2B-consistency of the decomposed system (i.e., ternary system equivalent to S where all the operators are replaced by auxiliary variables). It is thus well-known that S and S_1 are equivalent: `HC4` applied to S_1 and `HC4` applied to S produce the same contracted box B [6]. Finally, creating S' amounts to adding the constraint $v_1 = v_2$ to S_1. Thus, the box B' is smaller than or equal to B. □

Of course, this result is useless if the box B' is equal to B, and we want to determine conditions for obtaining a box B' that might be *strictly* smaller than B. Among the set of *primitive operators* that are defined in a standard

implementation of HC4, the analysis presented below highlights that the following subset of non-monotonic or non-continuous operators might bring additional contraction when they occur several times (as CS) in the same system: $sin(x)$, $cos(x)$, $tan(x)$ with non-monotonic domains, x^{2c} (c positive integer and $0 \in X$), $cosh(x)$ with $0 \in X$, $1/x$ with $0 \in X$ and binary operators $(+,-,\times,/)$.

3.2 Unary Operators

Let us first introduce some definitions. An *evaluation* function associated with a function f computes a *conservative* interval, i.e., the application of f on any tuple of values picked inside the input intervals falls inside the computed interval.

Definition 2. *Let IR be the set of all the intervals over the reals. $F : IR \rightarrow IR$, $Y = [\underline{y}, \overline{y}] = F(X)$ is an* **evaluation operator** *associated with a unary primitive operator f if: $\forall x \in X, \exists y \in Y$ such that $f(x) = y$.*

A *narrowing* operator N_F^x associated with a function f allows us to filter/contract the domain of a variable x.

Definition 3. *Let X be the domain of a variable x, let F be an evaluation operator associated with f, and let Y be an interval. N_F^x is a* **narrowing operator** *of F on x, if $X' = [\underline{x}, \overline{x}] = N_F^x(Y)$ verifies:*

$$f(\underline{x}) \in Y \wedge f(\overline{x}) \in Y \wedge \forall y \in Y, \forall x \in (X - X') : f(x) \neq y$$

Definition 4. *Let f be a function defined on $I(f)$. f is a* **monotonic function** *on an interval X if: $\forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \leq f(x_2)$* **or** *$\forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \geq f(x_2)$*

As said above, a necessary condition to replace a CS is when the contraction obtained by a narrowing operator on a given expression f is partially lost in the next evaluation of f. More formally:

Condition 1. *$\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\supseteq Y$ where X is the domain of variable x, F is the evaluation operator associated with f, and N_F^x is the projection narrowing operator of F on x.*

The following proposition indicates a simple condition to identify a *useless CS* for which no filtering is expected.

Proposition 2. *Let F be the evaluation operator associated with a unary operator f. Let N_F^x be the narrowing operator of F on a variable x of domain X. If f is a monotonic and continuous function, then:*

$$\forall Y \subseteq F(X), X' = N_F^x(Y) : F(X') \subseteq Y$$

Proof. WLOG we suppose that f is monotonically increasing. $X' = N_F(Y)$, then using Def. 3: $f(\underline{x}), f(\overline{x}) \in Y$, where $X' = [\underline{x}, \overline{x}]$. Finally, with Defs. 2 and 4, $F(X') = [f(\underline{x}), f(\overline{x})] \subseteq Y$. □

Proposition 3. *With the same notations as above, if f is a non-monotonic function, then:* $\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\subseteq Y$

Proof. The non monotonicity of f means:

$$\exists x_1, x_2, x_3 \subseteq X^3,\ x_1 \leq x_2 \leq x_3 \ s.t.\ f(x_2) > f(x_1)\ \wedge\ f(x_2) > f(x_3)$$

Using values x_1, x_2 and x_3 that satisfy the existency condition, we can suppose that $Y = [f(x_1), f(x_3)]$. As $(f(x_2) > f(x_1)) \wedge (f(x_2) > f(x_3))$, $f(x_2) \notin Y$. $X' = N_F(Y)$, then, with Def. 1, $[x_1, x_3] \subseteq X'$. Since $x_1 \leq x_2 \leq x_3$, $x_2 \in X'$, with Def. 2, $f(x_2) \in F(X')$. Finally, $F(X') \not\subseteq Y$. □

Example. Let $X = [-1, 3]$ be the domain of a variable x, and x^2 be an expression shared by two or more constraints. Suppose that in the narrowing phase of `HC4-revise`, the node corresponding to one of the expressions x^2 is contracted to: $Y = [3, 4]$. Applying the narrowing operator on x produces $X' = [-1, 2]$. In the next evaluation of the expression, $F(X') = [0, 4] \not\subseteq Y$.

Considering the standard operators managed in `HC4` (except operators like `floor`), the *useful CSs* do not satisfy Proposition 2 and satisfy Proposition 3.

3.3 N-Ary Operators (Sums, Products)

For binary (n-ary) primitive functions, Condition 1 above can be extended to the following **Condition 2**:

$$\exists Z \subseteq F(X, Y), X' = N_F^x(Z, Y), Y' = N_F^y(Z, X) : F(X', Y') \not\subseteq Z$$

where X, Y are the domains of variables x and y respectively, F is the evaluation operator associated with f, N_F^x and N_F^y are the narrowing/projections operators on x and y resp. This condition 2 is generally satisfied by the n-ary operators $+$ and $\times$ (resp. $-$ and $/$). Many examples prove this result (see below). The result is due to intrinsic "bad" properties of interval arithmetics. First, the set of intervals IR is not a group for addition. That is, let I be an interval: $I - I \neq [0, 0]$ (in fact, $[0, 0] \subset I - I$). Second, $IR \backslash \{0\}$ is not a group for multiplication, i.e., $I/I \neq [1, 1]$.

The proposition 4 provides a *quantitative idea* of *how much* we can win when replacing additive CSs. It estimates the width Δ that is lost in binary sums (when an additive CS is not replaced by an auxiliary variable). Note that an upper bound of Δ is $2 \times \min(Diam(X), Diam(Y))$ and depends only on the initial domains of the variables.

Proposition 4. *Let $x + y$ be a sum related to a node n inside the tree representation of a constraint. The domains of x and y are the intervals X and Y resp. Suppose that* `HC4-revise` *is carried out on the constraint: in the evaluation phase, the interval of n is set to $V = X + Y$; in the narrowing phase, the interval V is contracted to $V_c = [\underline{V} + \alpha, \overline{V} - \beta]$ (with $\alpha, \beta \geq 0$ being the decrease in left and right bounds of V); X and Y are contracted to X_c and Y_c resp. The difference Δ between the diameter of V_c (current projection) and the diameter of the sum $X_c + Y_c$ (computed in the next evaluation) is:*

$$\Delta = \min(\alpha, Diam(X), Diam(Y), Diam(V) - \alpha)\ +\ \min(\beta, Diam(X), Diam(Y), Diam(V) - \beta)$$

Example. Consider $X = [0, 1]$ and $Y = [2, 4]$. Thus, $V = X + Y = [2, 5]$. Suppose that after applying HC4-revise we obtain $V_c = [2 + \alpha, 5 - \beta] = [4, 4]$ ($\alpha = 2$, $\beta = 1$). With Proposition 4, the narrowing operator yields $X_c = [0, 1]$ and $Y_c = [3, 4]$. Finally, $X_c + Y_c = [3, 5]$ is $\Delta = 2$ units larger than $V_c = [4, 4]$.

The properties related to multiplication are more difficult to establish. Concise results (not reported here) have been obtained only in the cases when 0 does not belong to the domains or when 0 is a bound of the domains.

4 The I-CSE Algorithm

The novelty of our algorithm I-CSE lies in the way additive and multiplicative CSs are taken into account.

First, I-CSE manages the commutativity and associativity of $+$ and $\times$ in a simple way thanks to *intersections* between expressions. An **intersection** between two sums (resp. multiplications) f_1 and f_2 produces the sum (resp. multiplication) of their common terms. For example: $+(x, \times(y, +(z, x^2)), \times(5, z)) \cap +(x^2, x, \times(5, z)) = +(x, \times(5, z))$. Consider two expressions $w_1 \times x \times y \times z_1$ and $w_2 \times y \times x \times z_2$ that share the CS $x \times y$. We are able to view these two expressions as $w_1 \times (x \times y) \times z_1$ and $w_2 \times (x \times y) \times z_2$ since $\times(w_1, x, y, z_1) \cap \times(w_2, y, x, z_2) = \times(x, y)$.

Second, contrarily to existing CSE algorithms, I-CSE handles *conflictive* subexpressions. Two CSs f_a and f_b included in f are **in conflict** (or **conflictive**) if $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. An example of conflictive CSs occurs in the expression $f : x \times y \times z$ that contains the conflictive CSs $f_a : x \times y$ and $f_b : y \times z$. Since $x \times y$ and $y \times z$ have a non empty intersection y, it is not possible to directly replace both f_a and f_b in f.

I-CSE works with the n-ary trees encoding the original equations[1] and produces a DAG. The roots of this DAG correspond to the initial equations; the leaves correspond to the variables and constants; every internal node f corresponds to an *operator* ($+, \times, sin, exp$, etc) applied to its *children* $t_1, t_2..., t_n$. f represents the expression $f(t_1, t_2, ..., t_n)$ and $t_1, ..., t_n$ are the *terms* of the expression. *The CSs extracted by* I-CSE *are the nodes with several parents.*

We illustrate I-CSE with the following system made of two equations.

$$x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 = 2$$
$$\frac{(x^2 + y^3)(x^2 + cos(y)) + 14}{x^2 + cos(y)} = 8$$

4.1 Step 1: DAG Generation

This step follows a standard algorithm that traverses simultaneously the n-ary trees corresponding to the equations in a bottom-up way (see e.g. [8]). By labelling nodes with identifiers, two nodes with common children and with the same operator are identified equivalent, i.e., they are CSs.

[1] The $+$ and $\times$ operators are viewed as **n-ary** operators. They include $-$ and $/$. For example, the 3-ary expression $x^2 y/(2 - x)$ is viewed as $*(x^2, y, 1/(2 - x))$.

4.2 Step 2: Pairwise Intersection between Sums and Products

Step 2 pairwise intersects, in any order, the nodes corresponding to n-ary sums on one hand, and to n-ary products on the other hand. This step creates *intersection nodes* corresponding to CSs. *Inclusion arcs* link their parents to intersection nodes. If the intersection expression is already present in the DAG, an inclusion arc is just added from each of the two intersected parents to this node.

For instance, on Fig. 3-a, the node 1.4 is obtained by intersecting the nodes 1 and 4, and we create inclusion arcs from the nodes 1 and 4 to the node 1.4. This means that 2 (i.e., y and x^2) among the 3 terms of the sum/node 4 are in common with 2 among the 4 terms of the sum/node 1. (Note that the two terms are in different orders in the intersected nodes.) The node 10 corresponds to the intersection between nodes 4 and 10 (in fact the node 10 is included in the node 4), but it has already been created at the first step.

Step 2 is a key step because it makes appear CSs modulo the commutativity and associativity of $+$ and $\times$ operators, and creates at most a quadratic number of CSs. By storing the maximal expressions obtained by intersection, intersection nodes and inclusion arcs enable `I-CSE` to compute all the CSs before adding them in the DAG in the next step[2].

4.3 Step 3: Integrating Intersection Nodes into the DAG

In this step, all the intersection nodes are integrated into the DAG, creating the definitive DAG. The routine is top-down and follows the inclusion arcs. Every node f is processed to incorporate into the DAG its "children" reached by an inclusion arc.

If f has no conflictive child, the inclusion arcs outgoing from f are transformed into plain arcs. Also, to preserve the equivalence between the DAG and the system of equations, one removes arcs from f to the children of its intersection terms/children. For instance, on Fig. 3-b, Step 3 modifies the inclusion arc $1 \rightarrow 1.4$ and removes the arcs $1 \rightarrow 12$ and $1 \rightarrow y$.

Otherwise, f has children/CSs in conflict, i.e., there exist at least two CSs f_a and f_b, included in f, such that $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. In this case, one or several nodes $(f_1, f_2, ..., f_r)$ equivalent to f are added such that: the set $f, f_1, f_2, ..., f_r$ cover all the CSs included in f. To maintain the DAG equivalent to the original system, a node *equal* is created with the children: $f, f_1, f_2, ..., f_r$. For example on Fig. 3, the node 4 associated with the expression $y + x^2 + y^3 - 1$ has two CSs in conflict. Step 3 creates a node $4b$ (attaching the conflictive CS $x^2 + y^3$) redundant to the node 4 (attaching the other conflictive CS $y + x^2$).

A greedy algorithm has been designed to generate a small number r of redundant nodes, with a small number of children. r is necessarily smaller than the number of CSs included in f. We illustrate our greedy algorithm handling conflictive CSs on a more complicated example, in which an expression (node) $u = s + t + x + y + z$ contains 3 CSs in conflict: $v_1 = s + t$, $v_2 = t + x$, $v_3 = y + z$

[2] If one did not want to manage conflictive expressions, one would incorporate directly, in Step 2, the new CSs into the DAG.

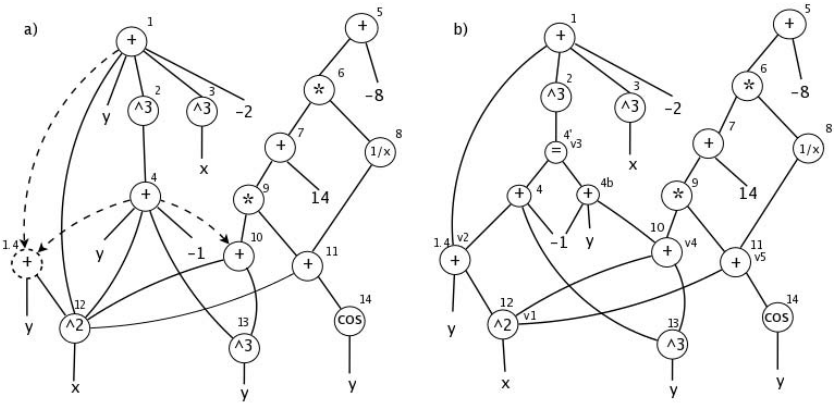

Fig. 3. (a) DAG obtained after the first two steps of `I-CSE`. **Step 1**: the system is transformed into a DAG including all the nodes, excepting 1.4, together with the arcs in plain lines. (For the sake of clarity, we have not merged the variables with multiple occurrences.) **Step 2**: the $\times$ and $+$ nodes are pairwise intersected, resulting in the creation of the node 1.4 and the three *inclusion arcs* in dotted lines. **(b)** DAG obtained after **Step 3**: all the inclusion arcs have been integrated into the DAG. For the conflictive subexpressions (nodes 1.4 and 10), a redundant node $4b$ and an equality node $4'$ have been created. The node 1.4 is attached to 4 whereas the node 10 is attached to $4b$. **Step 4** generates the auxiliary variables corresponding to the useful CSs ($v1$, $v2$, $v4$ and $v5$) and to the equality nodes ($v3$).

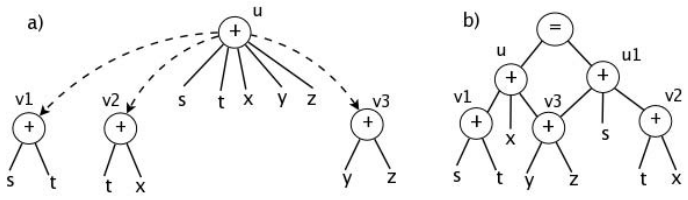

Fig. 4. Integrating intersection nodes into the DAG. a) The node u has three CSs in conflict. b) The DAG, with an equality node, obtained by the greedy algorithm.

(Fig. 4-a). The greedy algorithm works in two phases. In the first phase, several occurrences of u are generated until all the CSs are replaced. On the example, $u = v_1 + x + v_3$ and $u_1 = s + v_2 + y + z$ are created. The second phase handles all the redundant equations that have been created in the first phase. In a greedy way, it tries to introduce CSs into every equation to obtain a shorter equation that improves filtering. On the example, it transforms $u_1 = s + v_2 + y + z$ into $u_1 = s + v_2 + v_3$. Finally, an equality node (=) is associated with the node u and the redundant node u_1 (Fig. 4-b).

4.4 Step 4: Generation of the New System

A first way to exploit CSs for solving an NCSP is to use the DAG obtained after Step 3. As shown by Vu et al. in [20], the propagation phase cannot still be carried out by a pure `HC4`, and a more sophisticated propagation algorithm must consider the unique DAG corresponding to the whole system.

Alternatively, in order to still be able to use `HC4` for propagation, and thus to be compatible with existing interval-based solvers, Step 4 generates a new system of equations in which an auxiliary variable v and an equation $v = f$ are added for every *useful* CS. Avoiding the creation of new equations for useless CSs, which cannot provide additional contraction, decreases the size of the new system. In addition, redundant expressions $(f, f_1, f_2, ..., f_r)$ linked by an equality node, add a new auxiliary variable v' and the equations $v' = f$, $v' = f_1$,...,$v' = f_r$. To achieve these tasks, Step 4 traverses the DAG bottom-up and generates variables and equations in every node.

Finally, the new system will be composed by the modified equations (in which the CSs are replaced by their corresponding auxiliary variable), by the auxiliary variables and by the new constraints $v = f$ corresponding to CSs. The new system corresponding to the example in Fig. 3 is the following:

$$\begin{array}{lll} v_2 + (v_3)^3 + x^3 - 2 = 0 & v_1 = x^2 & v_3 = -1 + y + v_4 \\ \dfrac{v_4 \times v_5 + 14}{v_5} - 8 = 0 & v_2 = y + v_1 & v_4 = v_1 + y^3 \\ & v_3 = v_2 + y^3 - 1 & v_5 = v_1 + cos(y) \end{array}$$

For a given system of equations, our interval-based solver manages two systems: the new system generated by `I-CSE` is used only for `HC4` and the original system is used for the other operations (bisections, interval Newton). The intervals in both systems must be synchronized during the search of solutions. First, this allows us to clearly validate the interest of `I-CSE` for `HC4`. Second, carrying out Newton or bisection steps on auxiliary variables would need to be validated both in theory and in practice. Finally, this implementation is similar to the DAG-based solving algorithm proposed by Vu et al. which also considers only the initial variables for bisections and interval Newton computations, the internal nodes corresponding to CSs being only used for propagation [20].

4.5 Time Complexity

The time complexity of `I-CSE` mainly depends on the number n of variables, on the number k of a-ary operators and on the maximum arity a of an a-ary sum or multiplication expression in the system. $k + n$ is the size of the DAG created in Step 1, so that the time complexity of Step 1 is $O(k + n)$ on average if the identifiers are maintained using hashing. In Step 2, the number i of intersections performed is quadratic in the number of sums (or products) in the DAG, i.e., $i = O(k^2)$. Every intersection requires $O(a)$ on average using hashing (a worst-case complexity $O(a\,log(a))$ can be reached with sets encoded by trees/heaps). The worst-case for Step 3 depends on the maximum number of inclusion arcs which

Table 1. Time complexity of `I-CSE` on three representative scalable systems of equations (see Section 6). The CPU times have been obtained with a processor Intel 2.40 `GHz`. The CPU time increases linearly in the size $k+n$ of the DAG for `Trigexp1` and `Katsura` while the time complexity for `Brown` reaches the worst-case one.

Benchmark	`Trigexp1`			`Katsura`			`Brown`		
Number n of variables	10	20	40	5	10	20	10	20	40
Number k of operators	46	96	196	15	55	208	10	20	40
`I-CSE` time in second	0.19	0.28	0.63	0.08	0.19	0.91	0.05	0.20	1.26

is $O(k^2)$. Step 4 is linear in the size of the final DAG and is $O(k+n+i)$. Overall, `I-CSE` is thus $O(n + a\,log(a)\,k^2)$. Table 1 illustrates how the time complexity evolves in practice with the size of the system.

5 Implementation of `I-CSE`

`I-CSE` has been implemented using Mathematica version 6. Mathematica first automatically transforms the equations into a canonical form, where additions and multiplications are n-ary and where are performed reductions, i.e., factorizations by a constant. For instance, the expression $2x-y+x+z$ is transformed into $+(\times(3,x),-y,z)$. The n-ary representation of equations is useful for the pairwise intersections of `I-CSE` (Step 2).

The solving algorithms are developed in the open source interval-based library in `C++` called `Ibex` [5]. A given benchmark is solved by a branch and prune process: the variables are bisected in a round-robin manner and contracted by constraint propagation (`HC4` only, or `3BCID` using `HC4` – `3BCID` is a variant of `3B` [19]) and interval Newton. As mentioned above, `Ibex` offers facilities to create two systems of equations in memory for which domains of variables are synchronized during the search of solutions.

`I-CSE-B` and `I-CSE-NC`

We have proven theoretically that the interest of `I-CSE` resides in the additional pruning it permits and not only in a decrease of the number of operations. To confirm in practice this significant result, we have designed two variants of `I-CSE` that compute fewer CSs. `I-CSE-B` (Basic `I-CSE`) simply ignores the step 2 of `I-CSE`. The commutativity and associativity of $+$ and $\times$ are not taken into account. Additive and multiplicative n-ary expressions are considered in a fixed binary form in which only a few subexpressions can be detected. For instance, the CS $x+y$ is detected in two expressions $x+y+z_1$ and $x+y+z_2$, but not in expressions $x+z_1+y$ and $x+z_2+y$.

`I-CSE-NC` (`I-CSE` with No Conflicts) completely exploits the commutativity and associativity of $+$ and $\times$, but does not take into account conflictive CSs. `I-CSE-NC` lowers the worst case time complexity of `I-CSE`, but does not replace all the CSs. If a given system does not contain CSs in conflict, `I-CSE` and `I-CSE-NC`

return the same new system (with no redundant equations). In the example, `I-CSE-NC` does not create the redundant equation $v_3 = y + v_7$, so that the equation $v_7 = v_9 + y^3$ is not created either. The second (initial) equation finally becomes: $\frac{(x^2+y^3)\times v_8+14}{v_8} = 8$.

Existing CSE algorithms take place between `I-CSE-B` and `I-CSE-NC` in terms of number of detected useful CSs. We assume here that the algorithm by Vu et al. [20] is similar to `I-CSE-NC`.

6 Experiments

Benchmarks have been taken in the first two sections (polynomial and non-polynomial systems) of the COPRIN page[3]. The selected sample fulfills systematic criteria: every tested benchmark is an NCSP with a finite number of isolated solutions (no optimization); all the solutions can be found by the ALIAS system [14] in a time comprised between one second and one hour; selected systems are written with the following primitive operators: `+`, `-`, $\times$, `/`, `sin`, `cos`, `tan`, `exp`, `log`, `power`. With these criteria, we have selected 40 benchmarks. The `I-CSE` algorithm detects no CS in 16 of them. There are also two more benchmarks (`Fourbar` and `Dipole2`) for which no test has finished before the timeout (one hour), providing no indication. 9 of the remaining 22 benchmarks are scalable, that is, can be defined with any number of variables. Table 2 provides information about the selected benchmarks. When there is no conflictive CS, `I-CSE` and `I-CSE-NC` return the same new system and there is no redundant constraints (#rc=0). The interval-based solver results will be the same.

For all the benchmarks, the CPU time required by `I-CSE` *(and variants) is often negligible and always less than 1 second.*

Remark. In the benchmarks marked with a star (*), the equations have not been initially rewritten into the canonical form by `Mathematica` (see Section 5). This leads to fewer CSs, but these CSs correspond to larger subexpressions shared by more expressions, providing generally better results.

Tables 3 and 4 compare the CPU times required by `Ibex` to solve the initial system (Init) and the systems generated by `I-CSE-B`, `I-CSE-NC` and `I-CSE`. Table 3 reports results obtained by a standard branch and prune approach with bisection, Newton and **HC4**. Table 4 reports results obtained by a branch and prune approach with bisection, Newton and **3BCID** (using `HC4` as a refutation algorithm). Both tables report CPU times in seconds obtained on a 2.40 `GHz Intel Core 2` processor with 1 `Gb` of RAM, and the corresponding gain w.r.t. the solving of the original system. The time limit has been set to 3600 seconds. The tables also report the number of generated boxes (#Boxes) during the search. This corresponds to the number of nodes in the tree search and highlights the additional pruning due to `I-CSE`. The precision of solutions has been set to 10^{-8} for all the benchmarks. The parameter used by `HC4` has been set to 1% in Table 3. The parameters used by `HC4` and `3BCID` have been set to

[3] `www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html`

Table 2. Selected benchmarks. The columns yield the name of the benchmark, the number of solutions (#s), the number of variables (n), the number of useful CSs (#cs) found by `I-CSE-B`, `I-CSE-NC`, `I-CSE`, the number of redundant constraints created by `I-CSE` due to conflictive CSs (#rc).

Benchmark			I-CSE-B	ICSE-NC	I-CSE		Benchmark			I-CSE-B	ICSE-NC	I-CSE	
	#s	n	#cs	#cs	#cs	#rc		#s	n	#cs	#cs	#cs	#rc
6body	5	6	2	3	3	0	Katsura-20	7	21	90	90	90	0
Bellido	8	9	0	1	1	0	Kin1	16	6	13	13	19	3
Brown-7	3	7	3	7	21	24	Pramanik	2	8	0	15	15	0
Brown-7*	3	7	3	1	1	0	Prolog	0	21	0	7	7	0
Brown-30	2	30	26	53	435	783	Rose	16	3	5	5	5	0
BroyBand-20	1	20	22	37	97	73	Trigexp1-30	1	30	29	29	29	0
BroyBand-100	1	100	102	119	479	473	Trigexp1-50	1	50	49	49	49	0
Caprasse	18	4	6	7	11	2	Trigexp2-11	0	11	15	15	15	0
Design	1	9	3	3	3	0	Trigexp2-19	0	19	27	27	27	0
Dis-Integral-6	1	6	4	6	18	9	Trigonom-5	2	5	7	9	20	14
Dis-Integral-20	3	20	18	34	207	171	Trigonom-5*	2	5	7	6	6	0
Eco9	16	8	0	3	7	1	Trigonom-10	24	10	15	15	26	15
EqCombustion	4	5	7	8	11	1	Trigonom-10*	24	10	15	12	12	0
ExtendWood-4	3	4	2	2	2	0	Yamamura-8	7	8	5	10	36	48
Geneig	10	6	11	14	14	0	Yamamura-8*	7	8	5	1	1	0
Hayes	1	8	9	8	8	0	Yamamura-12	9	12	9	18	78	119
I5	30	10	3	4	10	5	Yamamura-12*	9	12	9	1	1	0
Katsura-19	5	20	81	81	81	0	Yamamura-16	9	16	13	26	136	224

Table 3. Results obtained with `HC4` and interval Newton

Benchmark	**Time in second**				**Time(Init) / Time**			**#Boxes**		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	Init	ICSE-NC	I-CSE
EqCombustion	>3600	26.1	0.35	**0.14**	>137	>10000	**>25000**	>1e+08	3967	**1095**
Rose	>3600	500	**101**	**101**	>7.2	**>35**	**>35**	>3e+07	**865099**	**865099**
Hayes	141	51.9	**15.7**	**15.7**	2.7	**9**	**9**	550489	**44563**	**44563**
6-body	0.22	**0.07**	**0.07**	**0.07**	**3.1**	**3.1**	**3.1**	4985	**495**	**495**
Design	176	65.2	**63.2**	**63.2**	2.7	**2.8**	**2.8**	425153	**122851**	**122851**
I5	>3600	>3600	**1534**	1565	?	**>2.3**	>2.3	>3e+07	**7e+06**	7e+06
Geneig	3323	2910	2722	2722	1.14	**1.22**	**1.22**	7e+08	4e+08	4e+08
Kin1	8.52	8.32	8.32	**8.01**	1.02	1.02	**1.06**	**905**	909	**905**
Pramanik	89.3	92.1	**84.9**	**84.9**	0.97	**1.05**	**1.05**	487255	**378879**	**378879**
Bellido	15.7	15.9	**15.6**	**15.6**	0.99	**1.01**	**1.01**	29759	**29319**	**29319**
Eco9	**23.9**	**23.9**	24	24.1	**1.00**	**1.00**	0.99	126047	117075	**110885**
Caprasse	**1.56**	1.81	1.68	2.16	0.86	0.93	0.72	8521	7793	**7491**
Brown-7*	500	350	**0.01**	**0.01**	1.42	**49500**	**49500**	6e+06	**95**	**95**
Dis-Integral-6	201	0.46	1.3	**0.03**	437	155	**6700**	653035	4157	**47**
ExtendWood-4	29.9	**0.03**	**0.03**	**0.03**	**997**	**997**	**997**	422705	**353**	**353**
Brown-7	500	350	30.7	**1.49**	1.42	16.1	**332**	6e+06	258601	**3681**
Trigexp2-11	1118	208	**56.2**	**56.2**	5.38	**19.9**	**19.9**	1e+06	**316049**	**316049**
Yamamura-8*	13	13.3	**0.75**	**0.75**	0.98	**17.3**	**17.3**	29615	**2161**	**2161**
Broy-Banded-20	778	759	261	**58.1**	1.03	2.98	**13.4**	172959	46761	**12623**
Trigonometric-5*	15.8	12.3	**1.49**	**1.49**	1.28	**10.6**	**10.6**	10531	**1503**	**1503**
Trigonometric-5	15.8	12.3	8.94	**6.97**	1.28	1.77	**2.27**	10531	7369	**5307**
Yamamura-8	13	13.3	44.6	**10.8**	0.98	0.3	**1.20**	29615	115211	**13211**
Katsura-19	**1430**	1583	1583	1583	0.90	0.90	0.90	**145839**	153193	153193
Trigexp1-30	**2465**	3244	3244	3244	0.76	0.76	0.76	1e+07	1e+07	1e+07

10% in Table 4. We have put at the end of both tables the results corresponding to scalable benchmarks. To return a fair comparison between algorithms, we have selected for the scalable systems the instance with the largest number of variables n such that the solver on the original system finds the solutions in less

Table 4. Results obtained with 3BCID using HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	**Init**	ICSE-NC	I-CSE
Rose	2882	5.17	**4.04**	**4.04**	557	**713**	**713**	4e+06	**5711**	**5711**
Prolog	38.5	60	**0.14**	**0.14**	0.64	**275**	**275**	4647	**11**	**11**
EqCombustion	0.42	0.37	**0.06**	**0.06**	1.35	7	7	427	**23**	**23**
Hayes	32.6	27.2	**5.67**	**5.67**	1.13	**5.7**	**5.7**	17455	**1675**	**1675**
Design	52	17.9	**13.3**	**13.3**	2.9	**3.9**	**3.9**	16359	**4401**	**4401**
I5	33.5	41.1	**17.9**	**17.8**	0.81	1.9	**1.9**	10619	4387	**4281**
6-body	0.14	**0.08**	0.1	0.1	**1.75**	1.4	1.4	173	**51**	**51**
Kin1	1.66	2.66	1.76	**1.23**	0.62	0.94	**1.35**	**85**	161	197
Bellido	10.3	10.4	**9.98**	**9.98**	1	**1.03**	**1.03**	4487	**4341**	**4341**
Eco9	**11.6**	**11.6**	12.4	13.2	**1**	0.94	0.88	6205	6045	**5749**
Pramanik	**73.8**	114	96.8	96.8	0.65	0.76	0.76	124663	**95305**	**95305**
Caprasse	1.96	2.51	2.5	2.92	0.74	0.78	0.67	1285	1311	**1219**
Geneig	**696**	1050	1050	1050	0.66	0.66	0.66	362225	**362045**	**362045**
Trigexp2-19	2308	2.23	**0.03**	**0.03**	1035	**77000**	**77000**	250178	**7**	**7**
Brown-7*	600	318	**0.01**	**0.01**	1.88	**60000**	**60000**	662415	**9**	**9**
ExtendWood-4	185	**0.03**	**0.03**	**0.03**	6167	6167	6167	669485	**35**	**35**
Dis-Integral-6	135	0.18	0.51	**0.03**	750	264	**4500**	86487	185	**7**
Brown-7	600	318	4.75	**0.22**	1.88	126	**2700**	662415	2035	**23**
Yamamura-12*	1751	1842	**1.01**	**1.01**	0.95	**1700**	**1700**	364105	**307**	**307**
Yamamura-12	1751	1842	31.1	**8.72**	0.95	56.3	**200**	364105	5647	**445**
Trigonometric-10*	1344	506	**19.4**	**19.4**	2.67	**69**	**69**	140512	**2033**	**2033**
Trigonometric-10	1344	506	156	**49.6**	2.67	8.62	**27**	140512	19883	**3339**
Broy-Banded-100	9.96	20.3	14.8	**8.21**	0.49	0.67	**1.21**	13	23	**11**
Trigexp1-50	**0.15**	0.19	0.17	0.17	0.79	0.88	0.88	**1**	**1**	**1**
Katsura20	**3457**	5919	5919	5919	0.58	0.58	0.58	**62451**	120929	120929
Brown-30	>3600	>3600	>3600	**22.9**	?	?	>150	>210021	>151527	**31**
Dis-Integral-20	>3600	>3600	>3600	**1.12**	?	?	**>3200**	>111512	>75640	**39**
Yamamura-16	>3600	>3600	681	**35.6**	?	>5	**>100**	>522300	96341	**919**

than one hour. This number n is greater with 3BCID (Table 4) than with only HC4 (Table 3) because 3BCID is generally more efficient than HC4.

Tables 3 and 4 clearly highlight that I-CSE is very interesting in practice. We observe a gain in performance greater than a factor 2 on 15 among the 24 lines (on both tables). The gain is of two orders of magnitude (or more) for 5 benchmarks with HC4 (corresponding to 4 different systems) and for 10 benchmarks with 3BCID (corresponding to 8 different systems).

I-CSE clearly outperforms the variants extracting fewer useful CSs, as shown on Table 3 (see Brown-7, Dis-Integral-6, Broyden-Banded-20) and Table 4 (see Brown-7, Dis-Integral-6, Yamamura-12, Trigonometric-10). In these cases, the gains in CPU time are significant. They are sometimes of several orders of magnitude. The few exceptions for which I-CSE is worse than its simpler variants give only a slight advantage to I-CSE-NC or I-CSE-B.

The number of boxes is generally decreasing from the left to the right of tables. This confirms our theoretical analysis that expects gains in filtering when a system has additional equations due to CSs. This experimentally proves that exploiting conflictive CSs is useful. This confirms an intuition shared by a lot of practitioners of partial consistency algorithms that redundant constraints are often useful because they allow a better pruning effect [10]. Benchmarks like Brown-30, Dis-Integral-20 and Yamamura-16, have been added at the end of Table 4 to highlight this trend: I-CSE produces a gain in performance of 3 orders of magnitude while it adds hundreds of redundant equations.

Most of the obtained results are good or very good, but four benchmarks observe a loss of performance lying between 20% and 42%: `Caprasse` with both strategies, and `Pramanik`, `Geneig`, `Katsura` with `3BCID`. The loss in performance observed for `Katsura-20` (10% or 42% according to the strategy) is due to the domains of the variables that are initialized to [0,1]. Without detailing, such domains imply that the pruning in the search tree is due to the evaluation (bottom-up) phase and not to the (top-down) narrowing phase of `HC4-revise`.

7 Conclusion

This paper has presented the algorithm `I-CSE` for exploiting common subexpressions in numerical CSPs. A theoretical analysis has shown that gains in filtering can only be expected when CSs do not correspond to monotonic and continuous operators like x^3 or log. Contrarily to a belief in the community, this means that CSs can bring significant gains in filtering/contraction, and not only a decrease in the number of operations. These are good news for the significance of this line of research.

Experiments have been performed on 40 benchmarks among which 24 contain CSs. Significant gains of one or several orders of magnitude have been observed on 10 of them. `I-CSE` differs from existing CSEs in that it also detects conflictive CSs. As compared to `I-CSE-NC` (similar to existing CSEs), the additional contraction involved by the corresponding redundant equations leads to improvements of one or several orders of magnitude on 4 benchmarks (`Brown`, `Dis-Integral`, `Yamamura` and, only for `HC4`, `BroyBanded`).

A future work is to compare our implementation based on the standard `HC4` algorithm (and the management of two systems), with the sophisticated propagation algorithm carried out on the elegant DAG-based structure proposed by Vu, Schichl and Sam-Haroud. However, our experimental results have underlined that the gain in contraction has a greater impact on efficiency than the time required to reach the fixed-point of propagation. Thus, we suspect that both implementations will show similar performances.

References

1. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revising Hull and Box Consistency. In: Proc. ICLP, pp. 230–244 (1999)
2. Brown, D.P.: Calculus and Mathematica. Addison Wesley, Reading (1991)
3. Buchberger, B.: Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. Multidimensional Systems Theory, 184–232 (1985)
4. Ceberio, M., Granvilliers, L.: Solving Nonlinear Equations by Abstraction, Gaussian Elimination, and Interval Methods. In: Armando, A. (ed.) FroCos 2002. LNCS (LNAI), vol. 2309. Springer, Heidelberg (2002)
5. Chabert, G. (2008), `http://ibex-lib.org`
6. Collavizza, H., Delobel, F., Rueher, M.: Comparing partial consistencies. Reliable Computing 5(3), 213–228 (1999)
7. Debruyne, R., Bessière, C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In: Proc. IJCAI, pp. 412–417 (1997)

8. Flajolet, P., Sipala, P., Steyaert, J.-M.: Analytic variations on the common subexpression problem. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 220–334. Springer, Heidelberg (1990)
9. Granvilliers, L., Monfroy, E., Benhamou, F.: Symbolic-Interval Cooperation in Constraint Programming. In: Proc. ISSAC, pp. 150–166. ACM, New York (2001)
10. Harvey, W., Stuckey, P.J.: Improving Linear Constraint Propagation by Changing Constraint Representation. Constraints 7, 173–207 (2003)
11. Heck, A.: Introduction to Maple. Springer, Heidelberg (2003)
12. Lebbah, Y.: Contribution à la Résolution de Contraintes par Consistance Forte. Phd thesis, Université de Nantes (1999)
13. Lhomme, O.: Consistency Tech. for Numeric CSPs. In: IJCAI, pp. 232–238 (1993)
14. Merlet, J.-P.: ALIAS: An Algorithms Library for Interval Analysis for Equation Systems. Technical report, INRIA Sophia (2000), `http://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html`
15. Merlet, J.-P.: Interval Analysis and Robotics. In: Symp. of Robotics Research (2007)
16. Muchnick, S.: Advanced Compiler Design and Implem. M. Kauffmann (1997)
17. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
18. Schichl, H., Neumaier, A.: Interval analysis on directed acyclic graphs for global optimization. Journal of Global Optimization 33(4), 541–562 (2005)
19. Trombettoni, G., Chabert, G.: Constructive Interval Disjunction. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 635–650. Springer, Heidelberg (2007)
20. Vu, X.-H., Schichl, H., Sam-Haroud, D.: Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In: Proc. ICTAI 2004, pp. 72–81. IEEE, Los Alamitos (2004)