Expert Systems with Applications 42 (2015) 8086–8094

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Adaptive filtering strategy for numerical constraint satisfaction problems

Ignacio Araya [a,*], Ricardo Soto [a,b,c], Broderick Crawford [a,d,e]

[a] Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile
[b] Universidad Autónoma de Chile, Santiago, Chile
[c] Universidad Científica del Sur, Lima, Peru
[d] Universidad San Sebastián, Chile
[e] Universidad Central de Chile, Chile

ARTICLE INFO

Article history:
Available online 26 June 2015

Keywords:
Interval-based solvers
Branch and bound
Filtering algorithms
Consistency techniques

ABSTRACT

The reliability and increasing performance of search-tree-based interval solvers for solving numerical systems of constraints make them applicable to various expert system domains. Filtering methods are applied in each node of the search tree to reduce the variable domains without the loss of solutions. Current interval-based solvers generally leave it up to the solver designer to decide which set of filtering methods to apply to solve a particular problem. In this work, we propose an adaptive strategy to dynamically determine the set of filtering methods that will be applied in each node of the search tree. Our goal is twofold: first, we want to simplify the task of the solver designer, and second, we believe that an adaptive strategy may improve the average performance of the current state-of-the-art strategies.

The proposed adaptive mechanism attempts to avoid calling costly filtering methods when their probability of filtering domains is low. We assume that fruitful filtering occurs in nearby revisions or *clusters*. Thus, the decision about whether or not to apply a filtering method is based on a cluster detection mechanism. When a cluster is detected, the associated methods are consecutively applied in order to exploit the cluster. Alternately, in zones without clusters, only a cheap method is applied, thus reducing the filtering effort in large portions of the search. We compare our approach with state-of-the-art strategies, demonstrating its effectiveness.

1. Introduction

Interval-based solvers can solve systems of numerical constraints (i.e., nonlinear equations or inequalities over the reals). Their reliability and increasing performance make them applicable to various expert system domains, such as robotics design and kinematics (Merlet, 2011), dynamic systems in robust control and autonomous robot localization (Kieffer, Jaulin, Walter, & Meizel, 2000), and proofs of conjectures (Sandretto, Trombettoni, & Daney, 2013; Tucker, 2002). Three main features distinguish this approach from others:

- their reliability, i.e., their robustness w.r.t. roundoff errors due to floating-point calculation (Neumaier, 1990),
- their ability to take into account bounded errors in the coefficients of the constraints, e.g., errors of measurement and
- their ability to prove properties shared by an infinite (while continuous and bounded) set of points (Goldsztejn, 2005; Jaulin, Kieffer, Didrit, & Walter, 2001; Rohn, 1986; Shary, 1995).

There exist several interval-based solvers designed for solving systems of numerical constraints or Numerical Constraint Satisfaction Problems (NCSPs) (e.g., RealPaver (Granvilliers & Benhamou, 2006), Newton (Van Hentenryck, McAllester, & Kapur, 1997), Ibex (Chabert & Jaulin, 2009)). These solvers are variations of a branch and prune algorithm based on the interval Newton operator (Neumaier, 1990), and they generally perform a two-phase process. In the branching phase, a variable is chosen and its domain is split into two sub-domains, thus generating two subproblems. In the pruning phase, a series of filtering methods or *contractors* attempt to reduce the variable domains of each subproblem without loss of solutions. The phases are interleaved, generating a search tree. The process starts with the initial domain and stops when the domain sizes in the leaves of the search tree

* Corresponding author.
E-mail addresses: ignacio.araya@ucv.cl (I. Araya), ricardo.soto@ucv.cl (R. Soto), broderick.crawford@ucv.cl (B. Crawford).
URLs: http://www.inf.ucv.cl/~iaraya (I. Araya), http://www.inf.ucv.cl/~rsoto (R. Soto).

http://dx.doi.org/10.1016/j.eswa.2015.06.030
0957-4174/

are smaller than the precision given as input. These leaves contain all the solutions of the NCSP.

Current interval-based solvers (e.g., Ibex and RealPaver) include several contractors focusing on different types of subproblems. These methods can be chosen and parameterized by the user, and they are generally applied in each node of the search tree. Among the most well-known methods, we can mention the following:

- The interval Newton operator and its numerous variants (Hansen, 1992b; Neumaier, 1990), which offer very good results when the system is well-constrained (i.e., square systems of independent equations with a finite number of solutions) and the domain sizes are small enough (Hansen, 1992a).
- Constraint propagation-based algorithms proposed by the constraint programming community (e.g., `HC4` (Lhomme, 1993; Benhamou, Goualard, Granvilliers, & Puget, 1999), `Box` (Benhamou et al., 1999), `Mohc` (Araya, Trombettoni, & Neveu, 2010), `3BCID` (Trombettoni & Chabert, 2007), `ACID` (Neveu, Trombettoni, & Araya, 2015)). They filter domains by enforcing some level of consistency (e.g., hull consistency (Lhomme, 1993)[1]) and may achieve strong domain reductions of variables related to nonlinear constraints. However, due to the reduced scope of local consistencies, they are not so effective when the graph representation of the constraint system has many cycles. Stronger levels of consistency (e.g., 3B-consistency (Lhomme, 1993)) can improve the solver's performance in some difficult instances, but they are generally too expensive to be applied in every node of the tree search.
- Filtering algorithms based on linear relaxations (Araya, Neveu, & Trombettoni, 2012; Lebbah, Michel, Rueher, Daney, & Merlet, 2005, 2007; Ninin, Messine, & Hansen, 2014) achieve good results when the constraint system is almost linear (i.e., when the domain sizes of nonlinear variables are small enough). These methods usually work on a linear relaxation of the entire system or a part of it. The Simplex algorithm is then used to narrow the domain of each variable.

In this work, we propose an adaptive strategy to dynamically determine the set of contractors that will be used in each node of the search tree. Our goal is twofold. First, we want to simplify the task of the solver user or designer related to the election of the adequate set of contractors for solving a particular problem. Ideally, we would like to select all the contractors provided by the solver and let the solver choose which methods to use and when. Second, we believe that an adaptive strategy may improve the average performance of the current state-of-the-art strategies. We hope to reduce the effort it takes to apply all the contractors in each node of the search tree but still maintain the effectiveness of the contraction phase.

In our proposal, we assume that fruitful domain reductions occur in nearby nodes or clusters (the so-called *clustering effect*, experimentally identified for finite-domain constraint satisfaction problems in Stergiou, 2008). Thus, the basic idea consists of monitoring the search process in order to find clusters. The monitoring is performed by the contractors, which are applied periodically (for instance, one call every 4 nodes of the search tree). Each method is associated with a frequency depending on its time cost, i.e., cheaper methods are applied more frequently than expensive ones. When a contractor `ctc` performs filtering successfully, we say that a cluster has been detected. In order to exploit this cluster, `ctc` is applied successively in the nodes of the tree until it fails. `ctc` is applied only to the *active constraints*, i.e., constraints that actively participated in the reduction of domains related to the monitored node.

1.1. Related work

In interval-based solvers, expensive contractors are generally parameterized in order to be applied only when the subproblem verifies some specific conditions. For instance, the Newton operator in Ibex is applied when the largest domain is smaller than a given precision (Chabert & Jaulin, 2009). Optimization-Based Bound Tightening (OBBT) is a filtering method commonly used in nonconvex mixed-integer nonlinear programming (e.g., in optimizers such as Couenne (Belotti, Lee, Liberti, Margot, & Wächter, 2009), BARON (Sahinidis, 1996), and ANTIGONE (Misener & Floudas, 2013)). Because applying a full round of OBBT amounts to solving 2n linear programs, it is typically applied at the root node and within the search tree only with limited frequency or based on its success rate (Gleixner & Weltge, 2013). In Araya et al. (2010), `Mohc`, a contractor that exploits the monotonicity of functions, is fully applied only if a test that estimates its impact is passed.

Additionally, various ad hoc mechanisms have been proposed to reduce the effort put into specific interval-based filtering algorithms. For instance, in Goldsztejn and Goualard (2010), the authors propose an adaptive mechanism for an algorithm enforcing the box consistency (Benhamou et al., 1999). It takes into account past difficulties encountered in filtering a domain. In Neveu et al. (2015), the authors propose `ACID`, an adaptive variant of `3BCID`. `3BCID` is a state-of-the-art contractor that enforces 3B-consistency. It basically splits the domain of each variable (one-at-a-time) into several sub-domains. Then, a method enforcing a weaker consistency (e.g., `HC4`) is applied in order to reduce the domain of the split variable by proving that the sub-domains do not contain solutions. `ACID` reduces the cost of `3BCID` by reducing the number of handled variables. The number of variables handled is auto-adapted during the search. In some cases, `ACID` only calls once the method enforcing the weak consistency. The Achterberg' heuristic proposed in Baharev, Achterberg, and Rév (2009) reduces the number of calls of the Simplex algorithm in linear-relaxation-based contractors. These contractors generally reduce each bound of the variable domains by applying the Simplex algorithm to minimize or maximize the variable subject to a linear relaxation of the nonlinear system. The Achterberg' heuristic next selects the variable that may potentially offer the maximum reduction according to the found optimal solutions in the relaxed problem. The mechanism stops when the potential reduction of the next variable is too small to justify the effort. This heuristic has been used by solvers in several recent works (Araya, Trombettoni, & Neveu, 2012; Baharev & Rév, 2009; Hladík & Horáček, 2014).

In constraint satisfaction problems with discrete domains, some recent works adaptively select which filtering method to apply to each node of the search tree and constraint of the system. In Stamatatos and Stergiou (2009), a preprocessing phase learns which level of consistency to apply to which parts of the instance. Once the level is learned, it is statically applied during the entire search. In Balafrej, Bessiere, Coletta, and Bouyakhf (2013), for each variable/constraint, the solver learns during the search a parameter that characterizes a parameterized level of consistency to apply to the variable/constraint. That parameterized level lies between AC and a stronger level. In Balafrej, Bessiere, Bouyakhf, and Trombettoni (2014), the authors propose adaptive variants of partition-one-arc-consistency that do not necessarily run until having proved the fixpoint. The pruning can be weaker than the full version, but the computational effort can be significantly

[1] hull consistency is the counterpart of bound consistency in discrete constraint programming.

reduced. In Stergiou (2008) and Paparrizou and Stergiou (2012), heuristics allow the solver to dynamically select AC or a stronger level of consistency (maxRPC) during the search depending on the variable/constraint. The approaches are based on the clustering effect. Basically, when a filtering algorithm F_{weak} is successful (according to some propagation indicator), a stronger filtering algorithm F_{strong} is applied successively in order to exploit the cluster until it fails. The approach is restricted to two filtering algorithms, F_{weak} and F_{strong}, such that F_{strong} is strictly stronger (in filtering) than F_{weak}.

Compared to the related work, the main contributions of our approach are the following.

- Unlike mechanisms in current interval-based solvers (which generally control the application of only one contractor), we propose a simple and unified mechanism for controlling the application of several contractors.
- Unlike ad hoc adaptive mechanisms, our approach treats contractors as black boxes. Thus, it can be extended to use more and diverse kind of contractors. The only information required by our method is to know the set of constraints actively used during the application of each contractor (in any case, if this information is not available, we can assume that all constraints were actively used).
- Unlike most of works on discrete domains, our approach may use more than two contractors, and they are not required to have a strict relation of consistency. This is very important feature because, in general, the consistency (or filtering power) among interval-based contractors is not comparable.

Section 2 shows some background related to intervals and interval-based contractors for numerical CSPs. In Section 3, we explain how to extract the active constraints from different types of contractors. In Section 4, we present our approach and an adaptation of Stergiou's approach. Experiments are shown in Section 5. Finally, conclusions are given in Section 6.

2. Background

2.1. Intervals

An *interval* $[x_i] = [\underline{x_i}, \overline{x_i}]$ defines the set of reals x_i s.t. $\underline{x_i} \leqslant x_i \leqslant \overline{x_i}$, where $\underline{x_i}$ and $\overline{x_i}$ are floating-point numbers. $\mathbb{IR}$ denotes the set of all intervals. The size or *width* of $[x_i]$ is $w([x_i]) = \overline{x_i} - \underline{x_i}$. $m([x_i])$ denotes the middle of $[x_i]$. A *box* $[x] = ([x_1], \ldots, [x_n])$ determines the Cartesian product of intervals $[x_1] \times \ldots \times [x_n]$. The union of several boxes is generally not a box, and a *Hull* operator has been defined instead to define the smallest box enclosing all of them.

A function $[f] : \mathbb{IR}^n \to \mathbb{IR}$ is an *interval extension* of a real function $f : \mathbb{R}^n \to \mathbb{R}$ if it computes an enclosure of the image of f over any box $[x]$, i.e., $\forall [x] \in \mathbb{IR}^n, [f]([x]) \supseteq \{f(x), x \in [x]\}$.

2.2. The interval-based NCSP solver

In this work, we address solving non-linear systems of constraints or NCSPs.

Definition 1 (*Numerical constraint satisfaction problem (NCSP)*). A *numerical CSP*, or constraint system $S = (c, x, [x])$, consists of a vector $x = (x_1, \ldots, x_n)$ of variables varying in a box $[x] \in \mathbb{IR}^n$ and a set of constraints c. c involves equality and inequality arithmetic constraints. A *solution* $x' \in [x]$ of S satisfies all of the constraints in c.

When interval-based methods are used to compute the set of all of the solutions of an NCSP, they generally find a set of *atomic* boxes B_ϵ (i.e., boxes smaller than a required precision), such that all of the solutions of the problem belong to at least one of these boxes.

Without loss of generality, consider a set of less-than constraints $c = \{f_1(x) \leqslant 0, \ldots, f_m(x) \leqslant 0\}$, where $x = (x_1, \ldots, x_n)$ corresponds to a vector of variables. Algorithm 1 shows the basic skeleton of an NCSP algorithm. The search tree is implemented using a stack L, i.e., nodes are removed from and inserted to the front of L. This implies that the search follows a *depth-first search strategy*.

Algorithm 1. The NCSP solver

```
Data: c, x
procedure NCSPsolver([x], ϵ, contractors); out: B_ϵ
    L ← {[x]};
    while L ≠ ∅ do
        [x] ← pop(L);
        ([x^r], [x^l]) ← bisect([x]);
        for all [x] ∈ {[x^l], [x^r]} do
            //contraction phase:;
            for all ctc ∈ contractors do
                [x] ← ctc([x], c);
            end
            if [x] ≠ ∅ then
                if is-atomic-box? ([x], ϵ) then
                    B_ϵ ← B_ϵ ∪ {[x]};
                else
                    push (L, [x]);
                end
            end
        end
    end
end.
```

In each iteration, a node is taken from the front of L. Each node is treated in two phases: branching/bisection and contraction. The `bisect` method divides the current box into two sub-boxes by splitting the domain of one variable. The new boxes are then contracted by a set of contractors. Contractors from the `contractors` list are applied one by one to the box $[x]$ and the set of constraints c. They attempt to filter the boxes by eliminating inconsistent values from their bounds without loss of solutions. If all the values in a particular domain are filtered (i.e., a *wipeout* is produced), an empty box is returned.

After contraction, empty boxes are discarded. Non-empty boxes are inserted in L only if they are not atomic boxes; otherwise, they are inserted in B_ϵ. At the end of the search, B_ϵ contains the set of all of the solutions of the problem.

3. Contractors and active constraints

Among the primary and most expensive components in NCSP solvers are the contractors. In general, a contractor is applied to a set of constraints c and a box $[x]$. Then, the contractor attempts to contract the box $[x]$ by removing values from the bounds of the intervals that do not satisfy one or more constraints in c. When a contractor filters a box, we denote *active constraints* as those constraints used actively for filtering the box. In other words, the active constraints are those constraints in which the removed values were detected as inconsistent by the corresponding contractor. Active constraints are of crucial importance to our approach;

thus, we explain how to obtain them from different types of contractors in this section.

3.1. Constraint propagation contractors

Constraint propagation contractors treat constraints independently. They attempt to enforce the *hull consistency* on each constraint of the system. Enforcing the hull consistency is equivalent to finding the smallest box containing all the solutions related to a single constraint. This is an intractable problem in general, and several algorithms have been proposed thus far (e.g., Araya et al., 2010; Benhamou et al., 1999; Lhomme, 1993). All of them perform an AC3-like propagation loop for propagating the domain changes among constraints until a fix point is reached.

Algorithm 2. Basic structure of a constraint propagation contractor

```
procedure propag_ctc([x], c); out: [x]
    q ← c;
    while q ≠ ∅ do
        c_j ← pop(q);
        revise([x], c_j);
        if is_empty([x]) then
            return;
        else
            for variable x_i involved in c_j was significantly reduced
                do //c_j is an active constraint;
                for c_k ∈ c such that x_i is involved in c_j and c_k ≠ c_j do
                    push(q, c_k);
                end
            end
        end
    end
end.
```

Algorithm 2 shows the basic structure. It works as follows. A propagation queue q is first initialized with all the constraints of the system. An iterative process then handles every constraint in q until q becomes empty. A `revise` procedure performs the contraction of the box by considering a single constraint $c_j \in q$. If the contraction returns an empty box, the procedure finishes. Otherwise, each variable x_i involved in c_j, significantly reduced by the revision, propagates the changes to other constraints c_k involving x_i: these constraints are pushed into the propagation queue to be handled in subsequent iterations.

For this type of contractor, we consider active constraints to be those constraints causing a significant reduction in some variable domain (c_j in the for-loop).

3.2. 3B-like contractors

Stronger consistency algorithms similar to `3B` (e.g., `3BCID`, `ACID`) treat one variable at a time. The domain of the variable $[x_i]$ is split into several slices. Each corresponding subproblem is contracted by a constraint propagation contractor. Finally, the hull of the different contracted subproblems is returned. Generally, when some reduction is reached, the most reduced interval corresponds to that of the treated variable. Thus, when x_i is treated, we consider active constraints to be those constraints helping to remove slices from the bounds of $[x_i]$. The active constraints are obtained from the calls to the constraint propagation contractor that removed the slices.

3.3. Linear relaxation-based contractors

These methods usually work on a linear relaxation of the entire system or a part of it. Then, the Simplex algorithm is used to narrow the domain of each variable by using the relaxed system, i.e., the problem of minimizing x_i (resp. maximizing x_i) is solved to find a new lower bound (resp. upper bound) for each interval $[x_i]$. In this type of contractor, we consider active constraints to be the same active constraints found by Simplex at the optimal solution. These constraints can be found through the dual solution vector returned by the algorithm. Each value in this vector is associated with one constraint in the linear system, and each value different from zero corresponds to an active constraint. Of course, we map the linear active constraint used by Simplex to the nonlinear constraint in the original problem. The active constraints are considered only if the domain of the related variable has been reduced by more than a user-defined ratio τ.

4. The adaptive contraction approach

In this section, we present our approach. The basic idea consists of *monitoring* the search, i.e., applying costly contractors from time to time (e.g., once every 4 nodes), in order to detect clusters. Clusters are regions in the search space where contractors are effective. We consider that a cluster has been detected when a contractor `ctc` reaches a significant contraction in the node. When this event occurs, we exploit this cluster by consecutively calling the same contractor `ctc` with its subset of active constraints. `ctc` is called until the search leaves the cluster, i.e., until `ctc` ceases to be successful for filtering. When a contractor is applied in a box, it returns the filtered box and the set of active constraints c_{act}.

We consider a *filter* to be a pair $(\texttt{ctc}, c)$, where `ctc` is a contractor and c is a set of constraints. The algorithm uses two hash table lists. The periodicity list $\mathbf{T}$ keeps the set of filters F mapped to its periodicity of application $\mathbf{T}[F]$. That is, F should be used at least once every $\mathbf{T}[F]$ treated nodes. This list is fixed and may be defined by the user before starting the search. For instance, it can contain 3 filters: $(\texttt{HC4}, c), (\texttt{3BCID}, c)$ and $(\texttt{LRC}, c)$, where c is the set of all the constraints in the system. Periods can be set by hand according to the cost of the filters. We know that, in general, `LRC` is more expensive than `3BCID` and that `3BCID` is more expensive than `HC4`. Thus, we could set, for instance: $T[(\texttt{LRC}, c)] = 8, T[(\texttt{3BCID}, c)] = 4$ and $T[(\texttt{HC4}, c)] = 1$. The choice of powers of two is no coincidence. In this manner, we ensure that when it is the turn of an expensive contractor `ctc` to be applied, all the contractors cheaper than `ctc` have already been applied (contractors are applied from the cheapest to the most expensive).

The cluster list $\mathbf{C}$ is dynamic and contains the set of clusters currently detected by the monitoring. Each cluster is represented by a filter F, which is mapped to an integer value $\mathbf{C}[F]$. This value corresponds to the number of times the filter F will be applied consecutively in order to exploit the filtering in the cluster. Each time a filter of the cluster is applied, $\mathbf{C}[F]$ decreases in 1. Filters for which $\mathbf{C}[F] = 0$ are removed from the list.

Algorithm 3. Adaptive filtering

```
procedure adaptive_contraction([x], c, L, T, C); out: [x]
    iters ← iter + 1;
    filters ← {};
    for each F ∈ T do
        if iters%T[F] = 0 then add(filters, F)
    end
    for each F ∈ C do
        add(filters, F);
        C[F] ← C[F] − 1;
        if C[F] = 0 then remove(C, F);
    end
    for each (ctc, c) ∈ filters do
        c_act ← ctc([x], c);
        if c_act ≠ {} then
            // a cluster has been detected;
            C[(ctc, c_act)] ← L;
        end
    end
end.
```

Algorithm 3 shows the pseudo-code of our approach. This procedure replaces the contraction phase in Algorithm 1. The method `adaptive_contraction` first generates a set of filters *filters* to be used for contracting the current box $[x]$. Filters F whose turns have come, i.e., $iters\%\mathbf{T}[F] = 0$, are selected from the list $\mathbf{T}$. Additionally, all the filters from the cluster list are selected, decreasing the values $\mathbf{C}[F]$ in 1. In the second part of the algorithm, the filters are applied in turn to contract $[x]$. If a significant contraction is reached in the box (i.e., $c_{act} \neq \{\}$), then the active constraints c_{act} and the contractor `ctc` form a new filter, which is added to the cluster list. In the experiments, constraints are considered "active" when they help to contract one interval by at least 10%. L corresponds to the minimal number of consecutive calls that will be performed to the filter in order to exploit the cluster.

It makes little sense to apply a contractor to different sets of constraints. It is probably better to apply the contractor to the union of these sets. The cost is similar, but the second option offers, in general, a more powerful contraction. That is why, in the *filters* list, each contractor is associated with only one filter. Each time a new filter $F_1 = (\texttt{ctc}, c^1)$ is added to *filters*, if there is a filter with the same contractor $F_2 = (\texttt{ctc}, c^2)$, then F_2 updates its set of constraints to $c^1 \cup c^2$ and F_1 is removed.

4.1. A Stergiou-based adaptive contraction

In order to offer a more interesting comparison, we adapted the finite-domain oriented mechanism proposed in Stergiou (2008, 2012) to interval-based contractors.

The authors propose a mechanism for dynamically switching between a weak (W_{ctc}) and a strong (S_{ctc}) propagation method (propagator) for individual constraints during the search. The two main proposed heuristics are as follows:

- H_1: The propagator W_{ctc} is applied in the constraint until a wipeout is produced. If this is the case, the propagator S_{ctc} is applied in the constraint until no wipeouts are produced in L consecutive revisions of the constraint.
- H_2: The propagator W_{ctc} is applied in the constraint until at least one value is filtered from some domain. If this is the case, the propagator S_{ctc} is applied in the constraint until no filtered domains are produced in L consecutive revisions of the constraint.

We modified Algorithm 3 to adapt the heuristics to interval-based contractors. Algorithm 4 shows the contraction procedure applying the H_2 heuristic.

Algorithm 4. Adaptive filtering

```
procedure H2_contraction([x], c, L, W_ctc, S_ctc); out: [x]
    c' ← {};
    for each F = (S_ctc, c'') ∈ C do
        c' ← c' ∪ c'';
        C[F] ← C[F] − 1;
        if C[F] = 0 then remove(C, F);
    end
    c_act ← W_ctc([x], c/c');
    c_act ← c_act ∪ S_ctc([x], c' ∪ c_act);
    if c_act ≠ {} /*and [x] = ∅*/ then
        // a cluster has been detected;
        C[(S_ctc, c_act)] ← L;
    end
end.
```

The algorithm filters the current box using two contractors (or sets of contractors): W_{ctc} and S_{ctc} (e.g., `HC4` and `3BCID` respectively). First, it generates the set of constraints c' that will be used by the stronger contractor S_{ctc}. Constraints from the cluster list are selected, and the value of the corresponding filter is decreased by 1. Then, the algorithm applies W_{ctc} to the entire set of constraints minus the constraints in c' and applies S_{ctc} to the recently generated set of constraints c' plus the active constraints related to the recent application of the weaker contractor. The set of active constraints is obtained from both contractors. Finally, these active constraints generate a new cluster associated with the contractor S_{ctc}.

The heuristic H_1 is obtained by simply uncommenting the empty box condition of the last if statement and considering as active constraints only those constraints that cause a wipeout in the contractor.

5. Experiments

In order to validate our approach, we implemented the adaptive contraction algorithm into a state-of-the-art solver of the Interval-Based EXplorer library (Ibex (Chabert & Jaulin, 2009)). All the experiments were run on the same server (PowerEdge T420, with 2 quad-core Intel Xeon processors running at 2.20 GHz and 8 GB RAM).

The periodicity list $\mathbf{T}$ was initialized with four filters, which are applied to the entire system of constraints c: $(\texttt{HC4}, c), (\texttt{3BCID}, c), (\texttt{LRC}, c)$ and $(\texttt{IntervalNewton}, c)$. The periodicity of $(\texttt{HC4}, c)$ and $(\texttt{IntervalNewton}, c)$ was fixed to 1, i.e., these filters are applied in each node of the search tree. The former is utilized because it is the cheapest method and the standard filtering algorithm in interval solvers, and the latter is utilized because it is applied to a reduced set of instances (systems composed by n variables and n equations). For variable selection, we use the SmearSumRel heuristic, a variant of Kearfott's Smear function described in Trombettoni, Araya, Neveu, and Chabert (2011, 2013). Related to the extraction of active constraints from contractors, we considered that constraints are "active" when they help to contract one interval by at least 10%.

Our approach was compared with a *state-of-the-art solver* that simply replaces the adaptive contraction by a classical contraction phase. That is, the entire set of contractors is applied in each node of the search tree (see Algorithm 1).

A portion of the instances was selected from the COPRIN benchmark database.[2] We also transformed the constrained global optimization problems from series 1 and 2 of the COCONUT benchmark database[3] into unsatisfiable NCSPs. The instances were transformed just by adding the constraint $f_o(x) < f_o(x^*) - 10^{-8}$, where f_o is the minimization objective function and $f_o(x^*)$ is the optimal cost of the optimization problem. From the entire set of instances, we only considered instances that could be solved by the state-of-the-art solver in times ranging from 2 s to 3600 s (20 instances from COPRIN and 43 modified instances from COCONUT).

Definition 2 (*Average relative gain*). We define as relative time $t_r(a,b,\pi)$ the ratio between the time taken by an strategy a and the worst time taken by strategies a and b in solving an instance π, i.e., $t_r(a,b,\pi) = \frac{time(a,\pi)}{max(time(a,\pi),time(b,\pi))}$. Then, the *average relative gain* of an strategy a w.r.t. the state-of-the-art strategy (*SoA*) is given by $\frac{\sum_{\pi\in\Pi}(t_r(SoA,s,\pi))}{\sum_{\pi\in\Pi}(t_r(s,SoA,\pi))}$, where Π is the set of the considered instances.

Table 1
Comparison between the adaptive contraction approach and the state-of-the-art solver.

Conf.	All (63 instances)		$20 \leqslant t \leqslant 3600$ (35 instances)		$2 \leqslant t \leqslant 20$ (33 instances)	
	Av. gain	Tot. time	Av. gain	Tot. time	Av. gain	Tot. time
SoA	1.00	10757	1.00	10532	1.00	337
1,2	1.06	**10203**	1.06	**9990**	1.07	308
1,4	**1.08**	10210	**1.08**	9998	**1.12**	**295**
1,8	0.97	10893	1.03	10616	0.95	357
2,1	0.96	11335	0.93	11112	0.98	352
2,2	1.05	10352	1.04	10135	1.06	317
2,4	1.07	10264	1.07	10054	1.10	296
2,8	0.97	11338	1.01	11069	0.98	351
4,1	0.91	11884	0.87	11652	0.94	388
4,2	1.01	10682	0.99	10461	1.03	340
4,4	1.04	10586	1.03	10368	1.08	309
4,8	0.94	11630	0.97	11349	0.95	366
8,1	0.88	12769	0.82	12526	0.91	446
8,8	0.89	12574	0.93	12260	0.90	403

5.1. Comparison among different frequency configurations

Table 1 summarizes the results, comparing the state-of-the-art solver with our approach for different configurations of the periodicity list. Each row shows the results related to a configuration t_1, t_2, where $T[(\texttt{3BCID}, c)] = t_1$ and $T[(\texttt{LRC}, c)] = t_2$. In these experiments, we fixed L to 2. SoA denotes the state-of-the-art solver.

Columns show the average relative gain (av. gain) and the total time (tot. time) spent by configurations in different sets of instances: the entire set of instances (All); instances requiring between 20 and 3600 s for at least one configuration; and instances requiring between 2 and 20 s for at least one configuration.

Note that configurations (1,2), (1,4) and (2,4) offer the best gains, outperforming the state-of-the-art solver by up to 12%. These configurations also offer the lowest total times. Alternately, strategies with lower frequencies offer worse results. This is probably due to the fact that the reduction in contraction effort by a node is not enough to compensate for the increase in size of the search tree.

Table 2 reports detailed results for the best configurations and the subset of instances with the largest differences in time spent (>20%). For each strategy, we reported the spent CPU time and the size of the tree search as the number of treated nodes. The last row shows the average relative gain of each configuration for the subset of instances.

Observe that although the configuration (2,4) treats the largest number of nodes, it consumes the lowest total CPU time.

Related to this subset of instances, the best results are commonly obtained by configurations (1,4) and (2,4). The worst result is obtained in the instance *linear*, where the state-of-the-art solver outperforms all configurations by a factor between $[1.78, 3.30]$. The adaptive contraction approach outperforms by 18% the state-of-the-art solver on this subset of instances. The best results are obtained on the instances *creactor*, *ex7_3_5*, *himmel16* and *batch*, where the adaptive contraction outperforms the state-of-the-art solver by a factor of approximately 2.

5.2. Comparison among different values of L

In a second series of experiments, we compared the adaptive contraction approach for different values of L. Recall that L is a threshold related to the clustering effect. When a cluster is detected for a filter F (i.e., F has been successful in reducing domains), F is applied successively in the following nodes until it fails L consecutive times. $L = 0$ means that the clustering effect is not considered, i.e., each filter F is just applied periodically according to the periodicity table. Small values of L (e.g., $L = 1$) may imply that the algorithm incorrectly detects the end of a cluster, while large values of L may unnecessarily increase the filtering effort.

Table 3 summarizes the results. We took three of the best frequency configurations from the previous section $((1,4);(2,4);(4,4))$ and two bad configurations Average relative gain and set L to different values in $\{0,1,2,4,8\}$. Additionally, as an alternative, we considered L to be equal to the period associated to each filter ($L = T$ in the table). For instance, given the configuration $(4,8)$, we set $L = 4$ for the filter $(\texttt{3BCID}, c)$ and $L = 8$ for the filter $(\texttt{LRC}, c)$.

Columns show the average relative gain and total time spent by the strategies in the different sets of instances. Note that the approaches that do not take into account the clustering effect (i.e., $L = 0$) have the worst results, offering, on average, a gain lower than 0.5. Alternately, no large differences are noted for different values of $L > 0$. It seems that values in $[2,8]$ offer good results. Additionally, setting L automatically to the period of each filter seems to be a good choice, avoiding the task of setting this value by hand. Observe that the worst configurations greatly improve their performances by setting L to larger values.

Table 4 reports detailed results for the configuration $(1,4)$ and different values of L. We selected all the instances such that the ratio between the lowest and largest CPU time taken by the strategies (with the exception of $L = 0$) is larger than 1.2. For each strategy, we reported the spent CPU time and the size of the tree search in terms of the number of treated nodes. The last row shows the average relative gain of each configuration for the subset of instances.

The best performances for each instance are marked in bold. Note that in some instances, the best results are offered by the strategy without clustering. Those are instances in which the filter (LRC, c) is probably not useful. Strategies with $L > 0$ also offer good results in those instances. Alternately, when the strategy without clustering offers bad results, it most likely indicates that the filter (LRC, c) is useful. Additionally, in this case, strategies with $L > 0$ offer good results. Furthermore, strategies with $L > 0$ commonly offer results that are better than the strategy without clustering and the state-of-the-art solver (e.g., *synth*, *trig2-13*, *ex5_4_4*,

[2] http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html.
[3] http://www.mat.univie.ac.at/neum/glopt/coconut/Benchmark/Benchmark.html.

Table 2
Comparison between the adaptive contraction approach an the state-of-the-art solver in the subset of instances with the largest differences in performance.

Instance	State-of-the-art		1,2		1,4		2,4	
	Time	#Nodes	Time	#Nodes	Time	#Nodes	Time	#Nodes
Brent	20.7	2896	16.9	2940	**15.7**	3566	15.8	4576
Butcherb	449	70922	400	81542	**374**	98840	390	112334
Creactor	12.72	2716	9.28	2778	7.49	2834	**7.03**	2692
Eco9	21.1	2680	18.2	3262	**14.7**	3706	15.3	4622
i5	28.0	2976	22.0	3098	**17.8**	3210	19.8	4028
Kats14	10.4	188	10.9	288	12.8	368	**10.2**	686
Trig2-13	116	8308	98	8434	**82.8**	8542	89	10954
Dualc1	20.0	634	15.3	736	12.5	898	**10.1**	910
Ex5_4_4	263	3978	233	4914	226	6434	**218**	6262
Ex6_1_1	12.4	2336	12.1	2624	12.1	3324	**10.1**	3324
Ex6_1_3	65.7	5022	60.9	5804	60.6	7190	**51.1**	6912
Ex7_3_5	4.02	398	2.70	432	2.00	454	**1.84**	560
Ex8_5_1	7.48	7836	7.07	8568	5.30	2390	**5.26**	2698
Himmel16	47.1	3940	35.5	4326	**27.6**	4424	28.5	5648
Hydro	**8.36**	740	8.93	840	11.4	1190	11.7	1620
Linear	**52.0**	3858	93.6	6236	175	10412	108	9614
Batch	5.29	26	8.48	68	**3.14**	18	8.92	76
Hs119	**15.4**	592	16.4	726	19.8	1084	19.4	1412
Pentagon	2.22	1290	1.96	1322	**1.80**	1336	1.92	1672
Total	1162	121336	1071	138938	1083	160220	**1022**	180600
Av. gain	1.00		1.08		**1.19**		1.18	

Table 3
Comparison between different values of L.

Conf.	L	All (65 instances)		$20 \leqslant t \leqslant 3600$ (35 instances)		$2 \leqslant t \leqslant 20$ (33 instances)	
		Av. gain	Tot. time	Av. gain	Tot. time	Av. gain	Tot. time
1,4	0	0.36	108030	0.50	61280	0.30	50411
	1	1.06	10021	1.09	9793	1.07	311
	2	1.08	10210	1.07	9998	1.11	295
	4/T	1.08	10153	1.06	9946	1.11	295
	8	1.08	10205	1.06	10001	1.11	297
2,4	0	0.36	102854	0.48	60879	0.30	45645
	1	1.07	10241	1.07	10029	1.11	300
	2	1.07	10264	1.07	10054	1.10	296
	4	1.05	10280	1.06	10064	1.07	310
	8	1.06	10305	1.06	10096	1.08	305
	T	1.06	10353	1.05	10142	1.08	306
4,4	0	0.30	110632	0.38	66331	0.29	47987
	1	1.05	10504	1.03	10288	1.09	310
	2	1.04	10586	1.03	10368	1.08	309
	4/T	1.05	10520	1.03	10315	1.09	305
	8	1.05	10661	1.03	10458	1.08	303
4,8	2	0.94	11630	0.97	11349	0.95	366
	T	1.05	10586	1.03	10380	1.08	308
8,8	2	0.89	12574	0.93	12260	0.90	403
	8/T	1.01	10838	1.00	10618	1.03	321
SoA		1.00	10757	1.000	10532	1.000	337

ex8_5_1, *hydro*, *batch*, *disc2*). These results highlight the interest in controlling the filtering effort in branch and bound algorithms. Finally, note that when $L = 8$, the adaptive contraction strategy outperforms the state-of-the-art solver by 22% on this subset of instances.

Fig. 1 shows the performance profile. We compared the state-of-the-art solver and the adaptive contraction approach with the configuration (1,4) and $L = 8$. From the set of instances solved in $[2, 3600]$ seconds by at least one of the two strategies, each curve represents the percentage of instances solved by the corresponding strategy in less than *factor* times the CPU time spent by the best strategy for each instance. For instance, when *factor* = 1, each curve shows the percentage of instances in which the corresponding strategy gives the best results (in 78% of the instances, the adaptive strategy outperforms the state-of-the-art solver).

Observe that when *factor* is just 1.05, the adaptive contraction approach reaches 95% of solved instances. The same percentage is reached by the state-of-the-art solver only when the factor is equal to 1.50.

5.3. Results of the Stergiou-based approach

The Stergiou-based approach requires one weaker and cheap contractor and one stronger and expensive one. We decided to define `HC4` as the weaker contractor and `3BCID` + `LRC` (i.e., a contractor which applies `3BCID` first and then `LRC`) as the stronger

Table 4
Comparison of different values of L for a subset of instances.

	State-of-the-art		$L=0$ (no clustering)		$L=1$		$L=2$		$L=4$		$L=8$	
	Time	#Nodes	Time	#Nodes	Time	#Nodes	Time	#Nodes	Time	#Nodes	Time	#Nodes
Brent	20.7	2896	16.0	3096	16.0	3192	15.7	3566	15.6	3174	**15.5**	3172
Butcherb	449	70922	369	114280	**368**	100670	374	98840	388	86122	407	78796
Creactor	12.7	2716	**6.30**	2902	7.18	2850	7.49	2834	8.58	2790	9.52	2764
Eco9	21.1	2680	**12.8**	4050	14.3	3730	14.7	3706	17.4	3304	19.3	2730
i5	27.9	2976	**17.2**	3262	18.2	3190	17.8	3210	20.0	3110	21.0	3106
Kats14	**10.4**	188	21.9	1356	12.3	370	12.8	368	11.9	248	11.5	212
Synth	305	2602	277	9756	**247**	4398	264	4420	268	3250	290	2664
Trig2-13	115.9	8308	82.7	8542	**82.3**	8542	82.8	8542	83.9	8516	83.8	8528
Dualc1	**10.0**	634	>3600	>132874	13.4	1084	12.5	898	11.0	704	10.2	636
Ex5_4_4	264	3978	>3600	>769562	224	8666	227	6434	**221**	4618	239	4268
Ex7_3_5	4.02	398	**1.78**	466	1.80	458	2.00	454	2.02	446	2.51	442
Ex8_5_1	7.48	7836	350	222016	5.77	2982	**5.30**	2390	7.15	8110	5.88	2970
Himmel16	47.1	3940	**26.4**	5100	27.0	4502	27.6	4424	29.1	4222	32.1	4010
Hydro	8.36	740	>3600	>364394	13.0	1396	11.4	1190	9.00	916	**7.72**	758
Linear	**52.0**	3858	>3600	>276424	157	11748	175	10412	93.9	6284	76.0	4964
Batch	5.29	26	>3600	>596288	12.8	140	3.14	18	**3.05**	18	**3.05**	18
Disc2	2.08	4	>3600	>1060286	**1.70**	16	1.94	12	1.96	12	1.95	12
Hs119	**15.4**	592	>3600	251858	22.0	1482	19.8	1084	17.1	770	16.0	672
Pentagon	2.22	1290	**1.78**	1342	1.82	1340	1.80	1336	1.80	1336	1.84	1336
Total	1392	116584	26380	3827854	1246	160756	1277	154138	**1210**	137950	1253	122058
Av. gain	1.00		0.67		1.13		1.20		1.20		**1.22**	

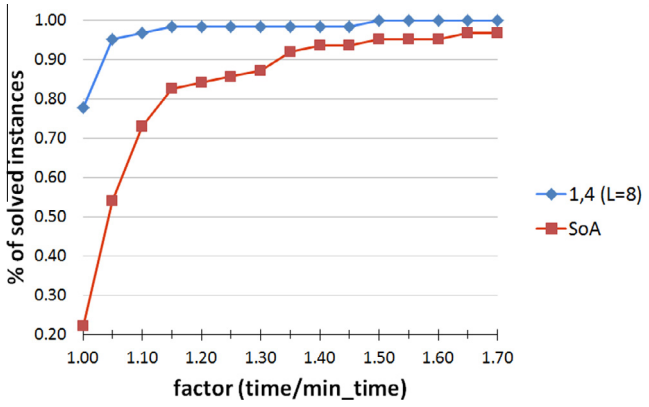

Fig. 1. Performance profile comparing the state-of-the-art solver and our approach using the best configuration.

Table 5
Comparison among our approach and the Stergiou-based approach.

L	(1,4), $L=8$		H_1		H_2	
	Av. gain	Tot. time	Av. gain	Tot. time	Av. gain	Tot. time
1	1.06	10021	0.38	53785	0.59	28917
2	1.08	10210	0.56	38417	0.75	20702
4	1.08	10153	0.69	29072	0.92	13696
8	1.08	10205	0.79	20296	1.00	10622

one. This decision was made because the contractors `3BCID` and `LRC` are comparably expensive and both of them are much more expensive than `HC4`.

Table 5 reports the average relative gains and the total time spent by of our approach with the configuration $(1,4)$ and $L=8$ and by the Stergiou-based approach (heuristics H_1 and H_2) for different values of the parameter L. The average gains are related to the state-of-the-art strategy.

Note that the Stergiou-based approach offers very bad results for low values of L. When L increases, the relative gains approach to 1.00, this is because the behavior of the adaptive mechanism approach to that of the state-of-the-art strategy, i.e., to apply the contractors in every node of the search tree.

The bad results are mainly due to the weaker contractor (i.e., `HC4`) has little to do with the stronger contractor (i.e., `3BCID` + `LRC`). We observed experimentally that, in many situations, `3BCID` or `LRC` *could have* filter a box, however they were not applied because `HC4` did not detect any cluster. The detection error of the monitoring mechanism may have a large impact in the solver performance because the size of the search tree may be increased enormously. In our approach, this situation does not occur mainly because each contractor is responsible for detecting its own clusters.

6. Conclusions

In this work, we present a mechanism for interval-based solvers that controls the filtering effort by adaptively selecting the set of contractors that will be applied in each node of the search tree. The mechanism was primarily conceived for solver users or designers to facilitate their tasks related to selecting the adequate set of contractors for solving a particular problem. Experiments also highlight that controlling the filtering during the search may result in performance improvements for state-of-the-art solvers, such as Ibex. By both improving the performance of interval-based solvers and facilitating the tasks of solver designers, our approach can help simplify the design and improve the performance of expert systems that require the solving of numerical systems of constraints.

The basic idea of the adaptive strategy consists of increasing the effort when it is more probable to contract a box while reducing the effort when it seems to be improbable to perform contraction. We assume that fruitful contractions occur in nearby revisions or clusters. The detection of these clusters is carried out by the same contractors. When a contractor performs a successful contraction, we consider a cluster to have been found. The cluster ends when the contractor ceases to be useful. Compared to an adaptation of the strategy proposed by Stergiou for interval methods, our approach offers more robust results. We believe that the better

performance is mainly due to how, in our approach, each contractor looks for clusters in which the *same* contractor may be useful. In contrast, in the Stergiou-based approach, one contractor (the weaker one) detects clusters for a second contractor (the stronger one); thus, an intrinsic relation between the contractors is required (for instance, a strict relation of consistency).

Our work opens the possibility of including more costly contractors in interval-based solvers, reducing the potential overhead. For instance, consider contractors that are useless in most cases but very useful in a few. Thanks to the monitoring mechanism, the cost of these contractors should be dissipated when they are useless. Alternately, when they are useful, clusters should be detected and exploited.

A drawback of our approach is that the user or designer has to define the monitoring period for each contractor. The monitoring period should depend on the cost and the effectiveness of the contractor, and in some cases, it may be difficult to choose its value. To address this problem, we plan to include an automatic and adaptive mechanism for setting the monitoring periods for each filter. This mechanism could be based on information extracted from the monitoring and/or information extracted from an initial phase of the search in which all the contractors are applied in each iteration.

Acknowledgments

Ignacio Araya is supported by the Fondecyt Project 11121366, Ricardo Soto is supported by the Fondecyt Project 11130459, Broderick Crawford is supported by the Fondecyt Project 1140897.

References

Araya, I., Trombettoni, G., Neveu, B. et al. (2010). Exploiting monotonicity in interval constraint propagation. In *AAAI*.

Araya, I., Neveu, B., & Trombettoni, G. (2012). An interval extension based on occurrence grouping. *Computing, 94*, 173–188.

Araya, I., Reyes, V., & Oreallana, C. (2013). More smear-based variable selection heuristics for ncsps. In *IEEE 25th international conference on tools with artificial intelligence (ICTAI)* (pp. 1004–1011). IEEE.

Araya, I., Trombettoni, G., & Neveu, B. (2012). A contractor based on convex interval taylor. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 1–16). Springer.

Baharev, A., Achterberg, T., & Rév, E. (2009). Computation of an extractive distillation column with affine arithmetic. *AIChE Journal, 55*, 1695–1704.

Baharev, A., & Rév, E. (2009). A complete nonlinear system solver using affine arithmetic. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference* (pp. 17–33).

Balafrej, A., Bessiere, C., Coletta, R., & Bouyakhf, E. H. (2013). Adaptive parameterized consistency. In *Principles and practice of constraint programming* (pp. 143–158). Springer.

Balafrej, A., Bessiere, C., Bouyakhf, E. H., & Trombettoni, G. (2014). Adaptive singleton-based consistencies. In *AAAI'14: Twenty-eighth conference on artificial intelligence* (pp. 2601–2607).

Belotti, P., Lee, J., Liberti, L., Margot, F., & Wächter, A. (2009). Branching and bounds tightening techniques for non-convex minlp. *Optimization Methods & Software, 24*, 597–634.

Benhamou, F., Goualard, F., Granvilliers, L., & Puget, J. -F. (1999). Revising hull and box consistency. In *International conference on logic programming*. Citeseer.

Chabert, G., & Jaulin, L. (2009). Contractor programming. *Artificial Intelligence, 173*, 1079–1100.

Gleixner, A. M., & Weltge, S. (2013). Learning and propagating lagrangian variable bounds for mixed-integer nonlinear programming. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 355–361). Springer.

Goldsztejn, A. (2005). *Définition et Applications des Extensions des Fonctions Réelles aux Intervalles Généralisés: Nouvelle Formulation de la Théorie des Intervalles Modaux et Nouveaux Résultats* (Ph.D. thesis). University of Nice Sophia Antipolis.

Goldsztejn, A., & Goualard, F. (2010). Box consistency through adaptive shaving. In *Proceedings of the 2010 ACM symposium on applied computing* (pp. 2049–2054). ACM.

Granvilliers, L., & Benhamou, F. (2006). Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software (TOMS), 32*, 138–156.

Hansen, E. (1992a). Bounding the solution of interval linear equations. *SIAM Journal on Numerical Analysis, 29*, 1493–1503.

Hansen, E. (1992b). *Global optimization using interval analysis*. Marcel Dekker inc.

Hladík, M., & Horáček, J. (2014). Interval linear programming techniques in constraint programming and global optimization. In *Constraint programming and decision making* (pp. 47–59). Springer.

Jaulin, L., Kieffer, M., Didrit, O., & Walter, E. (2001). *Applied interval analysis*. Springer.

Kieffer, M., Jaulin, L., Walter, É., & Meizel, D. (2000). Robust autonomous robot localization using interval analysis. *Reliable Computing, 6*, 337–362.

Lebbah, Y., Michel, C., & Rueher, M. (2007). An efficient and safe framework for solving optimization problems. *Journal of Computational and Applied Mathematics, 199*, 372–377.

Lebbah, Y., Michel, C., Rueher, M., Daney, D., & Merlet, J.-P. (2005). Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis, 42*, 2076–2097.

Lhomme, O. (1993). Consistency techniques for numeric csps. In *IJCAI*. Citeseer (Vol. 93, pp. 232–238).

Merlet, J.-P. (2011). Interval analysis and robotics. In *Robotics research* (pp. 147–156). Springer.

Misener, R., & Floudas, C. A. (2013). Antigone: Algorithms for continuous/integer global optimization of nonlinear equations. *Journal of Global Optimization*, 1–24.

Neumaier, A. (1990). *Interval methods for systems of equations* (Vol. 37). Cambridge university press.

Neveu, B., Trombettoni, G., & Araya, I. (2015). Adaptive constructive interval disjunction: Algorithms and experiments. *Constraints*, 1–16.

Ninin, J., Messine, F., & Hansen, P. (2014). A reliable affine relaxation method for global optimization. *4OR*. http://dx.doi.org/10.1007/s10288-014-0269-0. Issn: 1619-4500, 1614-2411, url: http://link.springer.com/10.1007/s10288-014-0269-0.

Paparrizou, A., & Stergiou, K. (2012). Evaluating simple fully automated heuristics for adaptive constraint propagation. *IEEE 24th international conference on tools with artificial intelligence (ICTAI)* (Vol. 1, pp. 880–885). IEEE.

Rohn, J. (1986). Inner solutions of linear interval systems. In *Proceedings of interval mathematics 1985, LNCS* (Vol. 212, pp. 157–158).

Sahinidis, N. V. (1996). Baron: A general purpose global optimization software package. *Journal of Global Optimization, 8*, 201–205.

Sandretto, J., Trombettoni, G., & Daney, D. (2013). Confirmation of hypothesis on cable properties for cable-driven robots. In *New trends in mechanism and machine science* (pp. 85–93). Springer.

Shary, S. (1995). Solving the linear interval tolerance problem. *Mathematics and Computers in Simulation, 39*, 53–85.

Stamatatos, E., & Stergiou, K. (2009). Learning how to propagate using random probing. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 263–278). Springer.

Stergiou, K. (2008). Heuristics for dynamically adapting propagation. In *ECAI* (pp. 485–489).

Trombettoni, G., & Chabert, G. (2007). Constructive interval disjunction. In *Principles and practice of constraint programming–CP 2007* (pp. 635–650). Springer.

Trombettoni, G., Araya, I., Neveu, B., & Chabert, G. (2011). Inner regions and interval linearizations for global optimization. In *AAAI*.

Tucker, W. (2002). A rigorous ode solver and smale's 14th problem. *Foundations of Computational Mathematics, 2*, 53–117.

Van Hentenryck, P., McAllester, D., & Kapur, D. (1997). Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis, 34*, 797–827.