Constraints (2015) 20:452–467
DOI 10.1007/s10601-015-9180-3

Adaptive constructive interval disjunction: algorithms and experiments

Bertrand Neveu · Gilles Trombettoni · Ignacio Araya

Published online: 5 February 2015

Abstract An operator called `CID` and an efficient variant `3BCID` were proposed in 2007. For the numerical CSP handled by interval methods, these operators compute a partial consistency equivalent to Partition-1-AC for the discrete CSP. In addition to the constraint propagation procedure used to refute a given subproblem, the main two parameters of `CID` are the number of times the main `CID` procedure is called and the maximum number of subintervals treated by the procedure. The `3BCID` operator is state-of-the-art in numerical CSP, but not in constrained global optimization, for which it is generally too costly. This paper proposes an adaptive variant of `3BCID` called `ACID`. The number of variables handled is auto-adapted during the search, the other parameters are fixed and robust to modifications. On a representative sample of instances, `ACID` appears to work efficiently, both with the `HC4` constraint propagation algorithm and with the state-of-the-art `Mohc` algorithm. Experiments also highlight that it is relevant to auto-adapt only a number of handled variables, instead of a specific set of selected variables. Finally, `ACID` appears to be the best interval constraint programming operator for solving and optimization, and has been therefore added to the default strategies of the `Ibex` interval solver.

Keywords Interval methods · Adaptive algorithms · Strong consistency

B. Neveu
Imagine LIGM Université Paris–Est, Paris, France
e-mail: Bertrand.Neveu@enpc.fr

G. Trombettoni (✉)
LIRMM, University of Montpellier, CNRS, Montpellier, France
e-mail: gilles.trombettoni@lirmm.fr

I. Araya
Pontificia Universidad Católica de Valparaíso, Valparaiso, Chile
e-mail: rilianx@gmail.com

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 able to handle domains such as robotics design and kinematics [16], dynamic systems in robust control or autonomous robot localization [12], or proofs of conjectures [24].

A filtering/contracting operator for *numerical constraint networks* (CNs) called *Constructive Interval Disjunction* (in short CID) has been proposed in [23]. CID is based on a shaving/singleton process. The shaving principle is used to compute the *Singleton Arc Consistency* (SAC) of the finite domain CSP [10] and the 3B-consistency of the numerical CSP [14]. It is also at the core of the SATZ algorithm [20] used to prove the satisfiability of Boolean formula. Shaving works as follows on discrete constraint networks. A value is temporarily assigned to a variable (the other values are temporarily discarded) and a partial consistency is computed on the remaining subproblem. If an inconsistency is obtained then the value can be safely removed from the domain of the variable. Otherwise, the value is kept in the domain.

Contrarily to arc consistency, this consistency cannot be achieved in an incremental way [10]. Indeed, the work of the underlying refutation procedure on the *whole* subproblem is the reason why a *single value* can be removed. Thus, obtaining the singleton arc consistency of finite-domain CNs requires an expensive fixed-point algorithm where all the variables must be handled again every time a single value is removed [10]. The remark still holds for the improved version SAC-Opt [7]. A similar shaving principle can be followed on numerical CNs by roughly splitting intervals into sub-intervals/slices, as we will show in Section 3, giving the algorithm CID and an efficient variant 3BCID [23].

Applied first to continuous constraint satisfaction problems handled by interval methods, 3BCID has been more recently applied to constrained global optimization problems. This algorithm is state-of-the-art for constraint satisfaction, but is generally dominated by constraint propagation algorithms like HC4 for optimization. The main practical contribution of this paper is to show that an adaptive version of CID becomes efficient for both real-valued satisfaction and optimization problems, while needing no additional parameter value from the user.

After a recall of the interval methods used for tackling numerical CSP in Section 2, we describe in Section 3 the algorithms 3B, CID and 3BCID at the base of the new ACID operator introduced in Section 4. Sections 5, 6, 7 and 8 show experiments highlighting the practical interest of ACID in continuous constraint solving and constrained global optimization, and justifying the auto-adaptation policy behind ACID.

2 Numerical CSP

A *numerical constraint network* (numerical CN) is defined by a tuple $P = (X, [X], C)$, where X denotes a n-set of numerical, real-valued variables ranging in a domain $[X]$. We denote by $[x_i] = [\underline{x_i}, \overline{x_i}]$ the interval/domain of variable $x_i \in X$, where $\underline{x_i}, \overline{x_i}$ are floating-point numbers (allowing interval algorithms to be implemented on computers). A solution of P is an n-vector in $[X]$ satisfying all the constraints in C. The constraints defined are numerical. They are equations and inequalities using mathematical operators like $+, \bullet, /, exp, log, sin$.

A Cartesian product of intervals like the domain $[X] = [x_1] \times ... \times [x_n]$ is called a (parallel-to-axes) *box*. $w(x_i)$ denotes the *width* $\overline{x_i} - \underline{x_i}$ of an interval $[x_i]$. The width of a

box is given by the width $\overline{x_m} - \underline{x_m}$ of its largest dimension x_m. 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.

Numerical CNs can be solved by a Branch & Contract interval strategy:

- **Branch:** a variable x_i is chosen and its interval $[x_i]$ is split into two sub-intervals, thus making the whole process combinatorial.
- **Contract:** a filtering process allows contracting the intervals (i.e., improving interval bounds) without loss of solutions.

The process starts with the initial domain $[X]$ and stops when the leaves (boxes) of the search tree reach a width inferior to a precision given as input. These leaves yield an approximation of all the solutions of the numerical CN.

Several contraction algorithms have been proposed. Let us mention the constraint propagation algorithm called `HC4` [5, 17], an efficient implementation of 2B [14], that can enforce the optimal local consistency (called *hull-consistency*) only if strong hypotheses are met (in particular, each variable must occur at most once in a same constraint). The 2B-Revise procedure works with all the *projection functions* of a given constraint. Informally, a projection function isolates a given variable occurrence within the constraint. For instance, consider the constraint $x + y = z.x; x \leftarrow z.x - y$ is a projection function (among others) that aims at reducing the domain of variable x. Evaluating the projection function with interval arithmetics on the domain $[x]\times[y]\times[z]$ (i.e., replacing the variable occurrences of the projection function by their domains and using the interval counterpart of the involved mathematical operators) provides an interval that is intersected with $[x]$. Hence a potential domain reduction. A constraint propagation loop close to that of `AC-3` [15] is used to propagate reductions obtained for a given variable domain to the other constraints in the system.

3 Shaving algorithms for numerical CSP

Stronger consistencies for numerical CSP have also been proposed.

3.1 3B algorithm

3B-consistency [14] is a theoretical consistency similar to SAC for CSP although limited to the bounds of the domains. Consider the $2n$ subproblems of the studied numerical CN where each interval $[x_i]$ ($i \in \{1..n\}$) is reduced to its lower bound $\underline{x_i}$ (resp. upper bound $\overline{x_i}$). 3B-consistency is enforced iff each of these $2n$ subproblems is hull-consistent.

In practice, the `3B`(w) algorithm splits the intervals in several sub-intervals, also called slices, of width w, which gives the accuracy: the 3B(w)-consistency is enforced iff the slices at the bounds of the handled box cannot be eliminated by `HC4`. Let us denote `var3B` the procedure of the `3B` algorithm that shaves one variable interval $[x_i]$ and s_{3b} its parameter, a positive integer specifying a number of sub-intervals: $w = w(x_i)/s_{3b}$ is the width of a sub-interval.

3.2 CID

Constructive Interval Disjunction (CID) is a partial consistency stronger than 3B-consistency [23]. CID-consistency is close to Partition-1-AC (P-1-AC) in finite domain CSP [6]. P-1-AC is strictly stronger than SAC [6].

The main procedure `varCID` handles a single variable x_i. The main parameters of `varCID` are x_i, a number s_{cid} of sub-intervals (accuracy) and a contraction algorithm ctc like `HC4`. $[x_i]$ is split into s_{cid} slices of equal width, each corresponding subproblem is contracted by the contractor ctc and the hull of the different contracted subproblems is finally returned, as shown in Algorithm 1.

Algorithm 1 The main `VarCID` procedure of the `CID` operator shaving a given variable x_i.

```
Procedure VarCID (x_i, s_cid, (X, C, in-out [X]), ctc)
    [X]' ← empty box
    for j ← 1 to s_cid do
        /* The jth sub-box of [X] on x_i is handled: */
        sliceBox ← SubBox (j, x_i, [X])
        /* Enforce a partial consistency on the sub-box: */
        sliceBox' ← ctc(X, C, sliceBox)
        /* "Union" with previous sub-boxes: */
        [X]' ← Hull([X]', sliceBox')
    [X] ← [X]'
```

Intuitively, `CID` generalizes `3B` because a sub-box that is eliminated by `var3B` is also discarded by `varCID`. In addition, contrary to `var3B`, `varCID` can also contract $[X]$ along *several* dimensions.

Note that in the actual implementation the `for` loop can be interrupted earlier, when $[X]'$ becomes equal to the initial box $[X]$ in all the dimensions except x_i.

3.3 3BCID

`var3BCID` is a hybrid and operational variant of `varCID`.

1. Like `var3B`, it first tries to eliminate sub-intervals at the bounds of $[x_i]$ of width $w = w(x_i)/s_{3b}$ each. We store the left box $[X_l]$ and the right box $[X_r]$ that are not excluded by the contractor ctc (if any).
2. Second, the remaining box $[X]'$ is handled by `varCID` that splits $[X]'$ into s_{cid} sub-boxes. The sub-boxes are contracted by ctc and hulled, giving $[X_{cid}]$.
3. Finally, we return the hull of $[X_l]$, $[X_r]$ and $[X_{cid}]$.

Figure 1 illustrates the contraction task achieved by the main procedure of 3B, `CID` and 3BCID.

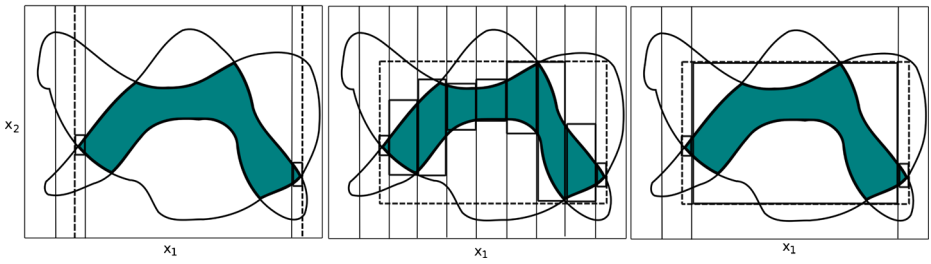

Fig. 1 Task of the var3B (*left*), varCID (*center*) and `var3BCID` (*right*) procedures applied to the interval $[x_1]$, with parameter s_{3b} set to 10 and s_{cid} set to 1. The *darkened* region corresponds to the solution set of the two constraints. The boxes in *dotted* lines are returned by the respective procedures

var3BCID comes from the wish of managing different widths (accuracies) for s_{3b} and s_{cid}. Indeed, the best choice for s_{3b} generally belongs to {5..20} while s_{cid} should always be set to 1 or 2 (implying a final hull of 3 or 4 sub-boxes). The reason is that the actual time cost of the shaving part is smaller than the one of the constructive domain disjunction. Indeed, if no sub-interval is discarded by var3B, only two calls to ctc are performed, one for each bound of the handled interval; if varCID is applied, the subcontractor is often called s_{cid} times.

The procedure var3BCID has been deeply studied and experimented in the past. The number and the order in which calls to var3BCID are achieved is a harder question studied in this paper.

4 Adaptive CID: learning the number of handled variables

Like for SAC or 3B, a quasi fixed-point in terms of contraction can be reached by 3BCID (or CID) by calling var3BCID inside two nested loops. An inner loop calls var3BCID on each variable x_i. An outer loop calls the inner loop until no interval is contracted more than a predefined (width) precision (thus reaching a quasi-fixed point). Let us call 3BCID-fp (fixed-point) this historical version.

Two reasons led us to radically change this policy. First, as said above, var3BCID can contract the handled box in several dimensions. One significant advantage is that the fixed-point in terms of contraction can thus be reached in a small number of calls to var3BCID. On most of the instances in satisfaction or optimization, it appears that a quasi fixed-point is reached in less than n calls. In this case, 3BCID is clearly too expensive. Second, the varCID principle is close to a branching point in a search tree. The difference is that a hull is achieved at the end of the sub-box contractions. Therefore an idea is to use a standard branching heuristic to select the next variable to be "varcided". We will write in the remaining part of the paper that a variable is *varcided* when the procedure var3BCID is called on that variable to contract the current box.

To sum up, the idea for rendering 3BCID even more efficient in practice is to replace the two nested loops by a single loop calling *numVarCID* times var3BCID and to use an efficient variant of the Smear function branching heuristic for selecting the variables to be varcided (called SmearSumRel in [22]). Informally, the Smear function favors variables having a large domain and a high impact on the constraints – by measuring interval partial derivatives.

A first idea is to fix *numVarCID* to the number n of variables. We call 3BCID-n this version. This gives good results in satisfaction but is dominated by pure constraint propagation in optimization. As said above, it is too time costly when the right *numVarCID* is smaller than n (which is often the case in optimization), but can also have a very bad impact on performance if a bigger effort brought a significantly greater filtering.

The goal of Adaptive CID (ACID) is precisely to compute dynamically during search the value of the *numVarCID* parameter. Several auto-adaptation policies have been tested and we report three interesting versions. All the policies measure the decrease in search space size after each call to var3BCID. They measure a *contraction ratio* of a box $[X]^b$ over another box $[X]^a$ as an average relative gain in all the dimensions:

$$\texttt{gainRatio}([X]^b, [X]^a) = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \frac{w(x_i^b)}{w(x_i^a)}\right)$$

4.1 ACID0: auto-adapting numVarCID during search

The first version ACID0 adapts the number of shaved variables dynamically at each node of the search tree. First, the variables are sorted by their impact, computed by the same formula as the SmearSumRel function (used for branching). Variables are then varcided until the cumulative contraction ratio during the last nv calls to var3BCID becomes less than $ctratio$. This algorithm has thus 2 parameters nv and $ctratio$, and it was difficult to tune them. We experimentally found that $ctratio$ could be fixed to 0.001 and nv should depend on the number of variables n of the problem. Setting nv to 1 is often a bad choice, and fixing it with the formula $nv = max(3, \frac{n}{4})$ experimentally gave the best results. The experimental results are not bad but this policy prevents numVarCID from reaching 0, i.e. from calling only constraint propagation. This is a significant drawback when a simple constraint propagation is the most efficient approach.

4.2 ACID1: interleaving learning and exploitation phases

A more sophisticated approach avoids this drawback. ACID1 interleaves learning and exploitation phases for auto-adapting the *numVarCID* value. Depending on the node number, the algorithm is in a learning or in an exploitation phase.

The behavior of ACID1, shown in Algorithm 2, is the following:

- The variables are first sorted according to their impact measurement (using the SmearSumRel heuristic).
- During a learning phase (during *learnLength* nodes), we then analyze how the contraction ratio evolves from a var3BCID call to the next one, and store the number *kvarCID* of varcided variables necessary to obtain most of the possible filtering.

  More precisely, 2.*numVarCID* variables are varcided at each node (with a minimum value equal to 2, in case *numVarCID*$=0$). In the first learning phase, we handle n variables.

  At the current node, the lastSignificantGain function returns the rank (*kvarCID*) of the last varcided variable giving a significant improvement (drop in domain size). In other words, after the *kvarCID*th call to var3BCID, the gain in current box size from a var3BCID call to the next one (computed by the gainRatio formula) never exceeds a small given ratio, called *ctratio*. This analysis starts from the last varcided variable to ensure we capture the last drop in domain size. (For the readibility of the pseudo-code, we omit the parameters of the var3BCID procedure, i.e. s_{3b}, s_{cid}, the constraints C and the contractor ctc.)
- During the exploitation phase following the previous learning phase, the average of the different *kvarCID* values (obtained in the nodes of the learning phase) provides the new value of *numVarCID*. This value will be used by 3BCID during the exploitation phase. Compared to the previous value (previous call to an exploitation phase), note that this new value can at most double, but can also drastically decrease.

Every cycleLength nodes in the search tree, both phases are called again.

Numerous variants of this schema were tested. In particular, it is counterproductive to learn *numVarCID* only once or, on the contrary, to memorize the computations from a learning phase to another one.

We fixed experimentally the 3 parameters of the ACID1 procedure *learnLength*, *cycleLength* and *ctratio*, respectively to 50, 1000 and 0.002. ACID1 becomes then a parameter

Algorithm 2 Algorithm `ACID1`

```
Procedure ACID1 (X, n, in-out [X], in-out call, in-out numVarCID)
    learnLength ← 50
    cycleLength ← 1000
    ctratio ← 0.002
    /* Sort the variables according to their impact */
    X ← smearSumRelSort (X)
    if call % cycleLength ≤ learnLength then
        /* Learning phase */
        nvarCID ← max(2, 2 . numVarCID)
        for i from 1 to nvarCID do
            [X]^old ← [X]
            var3BCID (X[i%n], [X], ...)
            ctcGains[i] ← gainRatio( [X], [X]^old)
        kvarCID[call] ← lastSignificantGain (ctcGains[], ctratio, nvarCID)
        if call % cycleLength = learnLength then
            /* End of learning phase */
            numVarCID ← average (kvarCID[])
    else
        /* Exploitation Phase */
        if numVarCID > 0 then
            for i from 1 to numVarCID do
                var3BCID (X[i % n], [X], ...)
    call ← call + 1
```

```
Function lastSignificantGain(ctcGains[ ], ctratio, nvarCID)
    for i from nvarCID downto 1 do
        if (ctcGains[i] > ctratio) then
            return i
    return 0
```

free procedure. With these parameter values, the overhead of the learning phases (where we double the previous *numVarCID* value) remains small.

4.3 ACID2: taking into account the level in the search tree

A criticism against `ACID1` is that we average *kvarCID* values obtained at different levels of the search tree. This drawback is partially corrected by the successive learning phases of `ACID1`, where each learning phase corresponds to a part of the search tree.

In order to go further in that direction, we designed a refinement of `ACID1` for which each learning phase tunes at most 10 different values depending on the width of the studied box. A value corresponds to one order of magnitude in the box width. For example, we store a *numVarCID* value for the boxes with a width comprised between 1 and 0.1, another one for the boxes with a width comprised between 0.1 and 0.01, etc. However, this approach, called `ACID2`, gave in general results similar to those of `ACID1` and appeared to be less

robust. Indeed, only a few nodes sometimes fall at certain width levels, which renders the statistics not significant.

5 Experiments

All the algorithms were implemented in the C++ interval library Ibex (Interval Based EXplorer), version 2.0 [8]. All the experiments were run on the same computer (Intel X86 3GHz). We tested the algorithms on square numerical CSP and on constrained global optimization. The square numerical CSP consists in finding all the solutions of a square system of n nonlinear equations with n real-values variables with bounded domains. Global optimization consists in finding the global minimum of a function over n variables subject to constraints (equations and inequalities), the objective function and/or the constraints being non-convex.

5.1 Experiments in constraint satisfaction

We selected from the COPRIN benchmark[1] all the systems that were solved by one of the tested algorithms in a time comprised between 2 s and 3,600 s. The timeout was fixed to 10,000 s. The required precision on the solution is 10^{-8}. Some of these problems are scalable. In this case, we selected the problem with the greatest number of variables that was solved by one of the algorithms in less than one hour.

We compared our ACID method and its variants with the well known filtering techniques: a simple constraint propagation HC4, 3BCID-n (see Section 4) and 3BCID-fp (fixed-point) in which a new iteration on all the variables is run when a variable domain width is reduced by more than 1 %. At each node of the search tree, we used the following sequence of contractors : HC4, *shaving*, Interval-Newton [11], and X-Newton [3]. *shaving* denotes a variant of ACID, 3BCID-n, 3BCID-fp or nothing when only HC4 is tested.

For each problem, we used the best bisection heuristics available (among two variants of the Smear function [22]). The main parameter *ctratio* of ACID1 and ACID2, measuring a stagnation (the last drop) in the filtering as long variables are varcided, was empirically fixed to 0.002. The var3BCID parameters s_{3b} and s_{cid} were fixed to the default settings, respectively 10 and 1, proposed in [23]. Experiments on the selected instances confirm that these settings are relevant and robust to variations. In particular, setting s_{3b} to 10 gives results better than with smaller values ($s_{3b} = 5$) and with greater values. (For 21 over the 26 instances, $s_{3b} = 20$ gives worse results.) As shown in Table 1, ACID1 appears to be often the best one, or close to the best one. In only 4 problems on 26, it was more than 10 % slower than the best. *The number of varcided variables was tuned close to 0 in the problems where HC4 was sufficient, and more than the number of variables in the problems where 3BCID-fp appeared to be the best method.*

In the left part of Table 2, we summarize the results obtained by the three variants of ACID and their competitors. It appears that only ACID1 could solve the 26 problems in 1 hour, while HC4 could solve only 21 problems in 10,000s. The gains in cpu time obtained by ACID1 w.r.t. competitors are sometimes significant (see the line *max gain*), while its losses remain weak. ACID0 with its two parameters was more difficult to tune, and it was not interesting to run the more complex algorithm ACID2. ACID1 obtains better gains w.r.t

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

Table 1 Continuous CSP solving: ACID1 results

	#var	ACID1 time	ACID1 #nodes	ACID1 #varcids	best	worst	Speedup $\frac{\text{ACID1}}{best}$	Speedup $\frac{\text{ACID1}}{worst}$
Bellido	9	3.45	518	5	ACID1	HC4	1	0.89
Brown-7	7	396	540,730	4.5	ACID1	HC4	1	0.82
Brent-10	10	17.63	3,104	9	ACID1	HC4	1	0.14
Butcher8a	8	981	204,632	9	3BCID-n	HC4	1.03	0.49
Butcher8b	8	388	93,600	10.8	ACID1	HC4	1	0.31
Design	9	29.22	5,330	11	3BCID-n	HC4	1.07	0.37
Dietmaier	12	926	82,364	26.3	ACID1	HC4	1	0.19
Directkin	11	32.73	2,322	7	ACID1	3BCID-fp	1	0.84
Disc.integralf2-16	32	592	58464	0.4	HC4	3BCID-fp	1.02	0.52
Eco-12	11	3156	297,116	12	ACID1	HC4	1	0.32
Fredtest	6	25.17	11,480	0.8	HC4	3BCID-fp	1.04	0.91
Fourbar	4	437	183,848	0.1	ACID1	3BCID-n	1	0.85
Geneig	6	178.2	83,958	2.9	HC4	3BCID-fp	1.02	0.82
Hayes	7	3.96	1,532	7.5	3BCID-n	HC4	1.14	0.77
I5	10	15.93	3,168	11.5	ACID1	HC4	1	0.13
Katsura-25	26	691	5396	10.4	ACID1	3BCID-fp	1	0.67
Pramanik	3	23.1	23,696	0.2	ACID1	HC4	1	0.69
Reactors-42	42	1,285	23,966	134	3BCID-fp	HC4	1.07	0.13
Reactors2-30	30	1,220	38,136	90	3BCID-n	HC4	1.14	0.12
Synthesis	33	356	7,256	53.8	3BCID-fp	HC4	1.15	0.25
Trigexp2-23	23	2,530	227,136	39.4	3BCID-fp	HC4	1.26	0.25
Trigo1-18	18	2,625	37,756	6.1	ACID1	3BCID-fp	1	0.80
Trigo1sp-35	36	2,657	70,524	2.4	ACID1	3BCID-fp	1	0.41
Virasoro	8	1,592	266,394	0.6	3BCID-n	3BCID-fp	1.08	0.28
Yamamura1-16	16	2,008	68,284	0.37	3BCID-n	HC4	1.02	0.86
Yamamura1sp-500	501	1,401	146	144	ACID1	HC4	1	0.14

For each instance, we present its number of variables and the results obtained by ACID1: the CPU time, the number of branching nodes in the search tree, the average number of varcided variables (tuned by ACID1 dynamically). We also report the best and the worst methods among ACID1, HC4, 3BCID-fp, and 3BCID-n, the cpu time ratio of ACID1 over the best method and over the worst method

3BCID-n in total time than on average because the best gains were obtained on difficult instances with more variables. In the right part of the table, we report the solving time ratios obtained when X-Newton is removed (¬ XN) from the contractor sequence (4 problems could not be solved in 10,000s). The only ACID variant studied was ACID1. ACID1 and 3BCID-n obtain globally similar results, better than 3BCID-fp, but with a greater dispersion (i.e., standard deviation) than with X-Newton since the shaving takes a more important part in the contraction.

Table 2 Numerical CSP: Solving time gain ratios

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2	ACID1 ¬ XN	3BCID-fp ¬ XN	3BCID-n ¬ XN
#solved $<3{,}600$	26	20	23	24	25	24	20	16	20
#solved $<10{,}000$	26	21	26	26	26	26	22	21	22
Average gain	1	0.7	0.83	0.92	0.96	0.91	1	0.78	1.02
Maximum gain	1	0.13	0.26	0.58	0.45	0.48	1	0.18	0.38
Maximum loss	1	1.04	1.26	1.14	1.23	1.05	1	2.00	1.78
Stand. dev. gain	0	0.32	0.23	0.15	0.15	0.19	0	0.34	0.28
Total time	23,594	>72,192	37,494	27,996	26,380	30,428	29,075	50,181	31,273
Total gain	1	<0.33	0.63	0.84	0.89	0.78	1	0.58	0.93

We report the number of problems solved before 3,600 s and before 10,000 s, and different statistics on the CPU time gain ratio of ACID1 over each competitor C_i (one per column): the average, maximum, minimum and standard deviation values of this ratio $\frac{acid1\ time}{C_i\ time}$

5.2 Experiments in constrained global optimization

We selected in the series 1 of the Coconut constrained global optimization benchmark[2] all the 40 instances that ACID or a competitor could solve in a CPU time comprised between 2 s and 3,600 s. The time out was fixed to 3,600s. We used the IbexOpt strategy of Ibex that performs a Best First Branch & Bound. The experimental protocol is the same as the numerical CSP experimental protocol, except that we do not use Interval-Newton that is only implemented for square systems.

For each instance, we use the best bisection heuristics (the same for all methods) among largestFirst, roundRobin and variants of the *Smear* function. The precision required on the objective is 10^{-8}. Each equation is relaxed by two inequalities with a precision 10^{-8}.

Table 3 reports the same columns as Table 1, plus a column indicating the number of constraints in the instance. For the constraint programming part of IbexOpt, HC4 is state of the art and 3BCID is rarely needed in optimization. Therefore, we report in the penultimate column a comparison between ACID1 and HC4. The number of varcided variables was indeed tuned by ACID1 to a value comprised between 0 and the number of variables. Again, we can see that ACID1 is robust and is the best, or at most 10 % worse than the best, for 34 among 40 instances. Table 4 shows that we obtained an average gain of 10 % over HC4. It is significant because the CP contraction is only a part of the IbexOpt algorithm [22] (linear relaxation and the search of feasible points are other important parts, not studied in this paper and set to their default algorithms in IbexOpt). ACID0 shaves a minimum of 3 variables, which is often too much. ACID2 obtains results slightly worse than ACID1, rendering this refinement not promising in practice.

[2] www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html

Table 3 Optimization problems: ACID1 results

	#var	#ctr	ACID1 time	ACID1 #nodes	ACID1 #varcids	best	worst	Speedup $\frac{\text{ACID1}}{best}$	Speedup $\frac{\text{ACID1}}{\text{HC4}}$	Speedup $\frac{\text{ACID1}}{worst}$
Ex2_1_7	20	10	8.75	465	3	HC4	3BCID-fp	1.03	1.03	0.7
Ex2_1_8	24	10	6.18	200	0	HC4	3BCID-fp	1.06	1.06	0.91
Ex2_1_9	10	1	10.09	1,922	0.75	HC4	3BCID-fp	1.04	1.04	0.9
Ex5_4_4	27	19	915	23,213	0.8	ACID1	3BCID-n	1	0.96	0.91
Ex6_1_1	8	6	60.85	13,071	8.9	HC4	3BCID-fp	1.21	1.21	0.73
Ex6_1_3	12	9	297	29,154	11.7	HC4	3BCID-fp	1.19	1.19	0.63
Ex6_1_4	6	4	1.99	505	6	ACID1	3BCID-fp	1	0.97	0.8
Ex6_2_6	3	1	106.8	46,687	0	HC4	3BCID-fp	1.02	1.02	0.74
Ex6_2_8	3	1	48.21	21,793	0.1	HC4	3BCID-fp	1.01	1.01	0.72
Ex6_2_9	4	2	51.92	19,517	0.1	HC4	3BCID-fp	1.02	1.02	0.72
Ex6_2_10	6	3	2248	569,816	0	ACID1	3BCID-fp	1	0.99	0.64
Ex6_2_11	3	1	29.32	13,853	0.3	HC4	3BCID-fp	1.05	1.05	0.73
Ex6_2_12	4	2	21.57	7,855	0.1	HC4	3BCID-fp	1.02	1.02	0.8
Ex7_2_3	8	6	19.41	4,596	4.4	3BCID-n	HC4	1.07	0.17	0.17
Ex7_2_4	8	4	36.79	5,606	4.2	3BCID-fp	HC4	1.04	0.66	0.66
Ex7_2_8	8	4	37.98	6,792	4.1	3BCID-n	HC4	1.09	0.71	0.71
Ex7_2_9	10	7	78.02	14,280	9.3	3BCID-n	HC4	1.07	0.48	0.48
Ex7_3_4	12	17	2.95	366	3	3BCID-n	3BCID-fp	1.23	0.99	0.89
Ex7_3_5	13	15	4.59	894	6	3BCID-n	HC4	1.05	0.38	0.38
Ex8_4_4	17	12	1738	46,082	0.9	ACID1	3BCID-fp	1	0.99	0.87
Ex8_4_5	15	11	772	25,454	4.8	HC4	3BCID-fp	1.03	1.03	0.75
Ex8_5_1	6	5	9.67	2,138	2.75	ACID1	3BCID-fp	1	0.84	0.82
Ex8_5_2	6	4	32.46	5,693	0.8	ACID1	3BCID-fp	1	0.9	0.87
Ex8_5_6	6	4	32.38	10,790	1.8	HC4	3BCID-fp	1.02	1.02	0.76
Ex14_1_7	10	17	665	95,891	3.3	3BCID-n	HC4	1.03	0.61	0.61
Ex14_2_3	6	9	2.01	360	2	HC4	3BCID-fp	1.17	1.17	0.69
Ex14_2_7	6	9	49.88	5,527	0	HC4	3BCID-n	1.47	1.47	0.48
alkyl	14	7	3.95	714	4	HC4	3BCID-fp	1.2	1.2	0.91
bearing	13	12	11.58	1,098	13	3BCID-n	HC4	1.01	0.53	0.53
hhfair	28	25	26.59	3,151	10	3BCID-n	HC4	1.12	0.58	0.58
himmel16	18	21	188	21,227	15.5	3BCID-n	3BCID-fp	1.1	0.94	0.88
house	8	8	62.8	27,195	3.25	HC4	3BCID-fp	1.09	1.09	0.79
hydro	30	24	609	32,933	0	ACID1	3BCID-fp	1	0.88	0.78
immun	21	7	4.17	1,317	2.5	ACID1	3BCID-fp	1	0.55	0.28
launch	38	28	107	2,516	21	ACID1	3BCID-n	1	0.79	0.43
linear	24	20	751	27,665	0.25	ACID1	3BCID-n	1	0.98	0.65
meanvar	7	2	2.43	370	2	HC4	3BCID-fp	1.04	1.04	0.84
process	10	7	2.61	611	8	HC4	3BCID-fp	1.08	1.08	0.77
ramsey	31	22	164.1	4,658	4.3	ACID1	3BCID-fp	1	0.85	0.68
srcpm	38	27	160	6,908	0.5	ACID1	3BCID-fp	1	0.62	0.33

Table 4 Optimization problems: gain ratio in solving time: time ACID1/time xxx

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2
#solved instances	40	40	40	40	40	40
Average gain	1	0.9	0.77	0.88	0.91	0.97
Maximum gain	1	0.17	0.28	0.35	0.62	0.28
Maximum loss	1	1.47	1.04	1.23	1.18	1.19
Stand. dev. gain	0	0.25	0.16	0.18	0.12	0.14
Total time	9,380	10,289	12,950	11,884	11,201	9,646
Total gain	1	0.91	0.72	0.79	0.84	0.97

6 Experiments with Ibex 2.1

We have extended our sample of instances in constrained global optimization to the series 2 of the Coconut benchmark. Among the 266 instances belonging to the series 1 (studied in the previous section) and the 727 instances of the series 2, we first discarded the unconstrained ones and the linear ones. We also discarded too difficult instances (having more than 50 variables or reaching a timeout of one hour – about 20 instances for the series 1 and 30 instances for series 2). Finally, we discarded the 70 "easy" instances from the series 1 (less than 0.5 second for all the competitors) and easy instances from the series 2 (having less than 6 variables or solved in less than 0.5 second). Overall, are remaining 43 instances in the series 1 and 32 instances in the series 2.

The sample was proceeded with the latest version of Ibex (2.1). This version is endowed with the same implementation of ACID1, but offers several other features. Let us mention an improvement of the polyhedral convex relaxation component. Instead of relaxing each inequality constraint with only a specific convex interval Taylor expanded at a vertex/corner of the box [3], as done in Ibex 2.0, a constraint is also relaxed using *affine arithmetic* that recursively applies on every operator in the expression [18, 19]. Both relaxed polyhedral forms are introduced in the same polytope. As a result, the part of the pure constraint programming in the total strategy is even lowered, although about the same gain of the strategy using ACID1(HC4) is observed w.r.t. that using only HC4 (see Section 6.1). Another significant feature is the state-of-the-art Mohc constraint propagation algorithm [1, 9] that was developed in Ibex 1 but not yet reimplemented in Ibex 2.0.

6.1 Performance on series 1 and 2 of the Coconut benchmark

Table 5 shows the results obtained by the IbexOpt strategy 2.1 using ACID1(HC4) (instead of HC4) on the new sample of 75 hard instances.

Although the version 2.1 of the optimization strategy takes more time on polyhedral relaxation, the gains of ACID1 w.r.t. HC4 remain similar. The average gain is an average of the gains in time or number of nodes obtained on each instance while the total gain is a ratio between the two total times spent for the whole benchmark.

The gain in number of nodes is very important in global optimization where the Branch and Bound process performs a best first search and thus may require an exponential memory.

Table 5 Results on the series 1 and 2 of Coconut

	ACID1	HC4	Remark
#solved instances	75	75	
Average time gain	1	0.95	
Average node gain	1	0.81	
Maximum time gain	1	0.15	obtained on ex_7_2_3
Maximum time loss	1	1.31	obtained on ex_6_1_1
Total time gain	1	0.91	
Total node gain	1	0.59	

Gains and losses are expressed as ratios ACID1 time/HC4 time and ACID1 #nodes/HC4 #nodes

6.2 Results obtained with Mohc

The main parameter of ACID1 is of course the constraint propagation algorithm used to refute or filter a given sub-box. HC4 is a constraint propagation algorithm often used in continuous constraint programming solvers [13] and even in global optimizers mainly based on mathematical programming algorithms like Baron [21]. However, other constraint propagation algorithms have been designed, like Box [5, 25] and Mohc [1, 9] (MOnotonic Hull Consistency). Mohc is an efficient constraint propagation algorithm exploiting the monotonicity of functions. Roughly, for a given constraint, the Mohc-Revise procedure brings an optimal reduction of the domain when all the corresponding functions are monotonic w.r.t. every involved variable. The contraction is also interesting as soon as one variable becomes monotonic. Note that monotonicity of functions increases with the decrease in domain size and thus more likely occurs lower in the search tree.

We have compared the performances obtained with ACID1(Mohc) and Mohc. It appears that the gain brought by ACID1 w.r.t. Mohc is about as interesting as w.r.t. HC4: 0.86 in total time and 0.79 in total number of nodes.

These results make ACID1(Mohc) a good candidate to belong to the default optimization strategy available in Ibex.

7 Justification of the parameters of ACID

This section gives a brief justification of important choices made in the implementation of ACID1: some alternative measures of the domain size used to capture the last drop in filtering as long as variables are varcided, and the way of aggregating in *numVarCID* the different numbers *kVarCID* of varcided variables at the different nodes of the learning phase.

7.1 Capturing the last drop in filtering

Determining the last drop in filtering requires one to measure the difference between two successive domain sizes as long as variables are varcided. Section 4 describes this contraction measure as an average of the size gains in all dimensions, i.e. a type of normalized perimeter ratio. However, two other ratios can be used instead:

- a ratio of box *volumes* (the volume being defined as the multiplication of the different interval sizes), as tested on discrete CSP [4],
- a ratio of the *maximum interval sizes* of the compared boxes.

Experimental results performed on our sample of 75 instances in global optimization showed no significant difference between the different policies.[3]

Of course, the best value chosen for `ctratio` (the value of the drop) depends on the size measure. The best value of `ctratio` was set to 0.002 in `ACID1`, while it is 1 % for the volume criterion and 5 % for the maximum dimension size criterion.

7.2 Aggregation of the learned *numVarCID* value with percentile?

As proposed in [4], we tried to aggregate with a percentile p %, and not a mean value, the different numbers *kVarCID* of varcided variables computed at the different nodes of the learning phase. Therefore the aggregated rank r is chosen such that p % of the different ranks are less than r.

Again, our experiments with percentiles equal to 40 %, 50 % (i.e., median), 60 %, 70 % show no difference with a simple averaged value.

8 Learning a set of variables to varcid?

In this section, we study an important question. The auto-adaptive policy of `ACID1` mainly adapts a number (*numVarCID*) of variables varcided at each search node. However, the specific set of variables that is varcided is given by a heuristic, namely the `smearSumRel` branching heuristic. So it is important to know whether a better heuristic could identify a better set of variables to varcid. Two experiments lead to a surprising conclusion.

8.1 Success-based impact heuristic

We have designed a new heuristic learning the (`CID`) impact of each variable. The impact of a variable x_i is computed by the (relative) number of times in which the shaving of x_i reaches a *significant gain in contraction* (see the function `lastSignificantGain` called in Algorithm 2).

The success-based impact ($sImpact_i$) of a variable x_i is updated after every call to `var3BCID`(x_i): The formula below is a weighted mean value depending on the previous mean value and on the current "success" (1 if a significant contraction gain is obtained, 0 otherwise).

$$sImpact_i = (1 - w) * sImpact_i + w * success$$

with $w = 0.02 = (1/learnLength)$.

The initial value of the $sImpact_i$'s is set to 1.0.

Although this simple type of learning process gave good results on an adaptive version of `Mohc` [2], it led here to performance results similar to that of the `SmearSumRel` heuristic.

We then tried more exhaustively the standard branching heuristics for ordering the variables varcided by `ACID1`: the different smear-based variants, largest domain, round-robin. This shows a slight advantage of `SmearSumRel` over its competitors (except compared to round-robin). This slight advantage led us to an informative and last experiment.

[3] Differences of less than 2 % between the three strategies have been measured, which is not representative.

8.2 Random heuristic

A random heuristic used to order the variables varcided by `ACID1` gave results very similar to `SmearSumRel`. Its average gain w.r.t. `SmearSumRel` in number of nodes is 0.99 while the average time gain is 0.95. The advantage in CPU time could be explained by the relative expensive cost of `SmearSumRel` that must compute a Jacobian matrix.

A conclusion suggested by these two experiments is that the number of variables varcided accounts more than the specific set of variables handled. A reason could be related to the fact that the variables are in a sense interchangeable. The removal of a sub-interval of $[x_i]$ during a shaving of $[x_i]$ (leading to an empty domain in $[x_j]$) could also be caused by the `varCID` operation on $[x_j]$. This could explain that $[x_i]$ or $[x_j]$ could be varcided indifferently.

9 Conclusion

We have presented in this paper an adaptive version of the `3BCID` contraction operator used by interval methods and close to `partition-1-AC` for the finite domain CSP. The best variant of this Adaptive `CID` operator (`ACID1` in the paper) interleaves learning phases and exploitation phases to auto-adapt the number of variables handled. All the parameters used for the adaptation are fixed and robust to modifications.

Overall, `ACID1` adds no parameter to the solving or optimization strategies. It offers the best results on average and is the best or close to the best on every tested instance, even in presence of the best `Ibex` devices (`Interval-Newton`, `X-Newton`). Therefore `ACID1` has been added to the `Ibex` default solving and optimization strategies.

Acknowledgments Ignacio Araya is supported by the Fondecyt Project 11121366.

References

1. Araya, I., Trombettoni, G., Neveu, B. (2010). Exploiting monotonicity in interval constraint propagation. In *Proceeding of AAAI* (pp. 9–14).
2. Araya, I., Trombettoni, G., Neveu, B. (2010). Making adaptive an interval constraint propagation algorithm exploiting monotonicity. In *Proceedings of CP, Constraint Programming, LNCS 6308* (pp. 61–68). Springer.
3. Araya, I., Trombettoni, G., Neveu, B. (2012). A Contractor based on convex interval Taylor. In *CPAIOR* 2012, no. 7298 in LNCS (pp. 1–16).
4. Balafrej, A., Bessiere, C., Bouyakhf, E., Trombettoni, G. (2014). Adaptive singleton-based consistencies. In *AAAI* (pp. 2601–2607). AAAI Press.
5. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F. (1999). Revising hull and box consistency. In *Proceedings of ICLP* (pp. 230–244).
6. Bennaceur, H., & Affane, M.S. (2001). Partition-k-AC: an efficient filtering technique combining domain partition and arc consistency. In *Proceedings of CP* (pp. 560–564).
7. Bessiere, C., & Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI* (pp. 54–59).
8. Chabert, G., & Jaulin, L. (2009). Contractor programming. *Artificial Intelligence*, *173*, 1079–1100.
9. Chabert, G., & Jaulin, L. (2009). Hull consistency under monotonicity. In *Proceedings of CP*, LNCS 5732 (pp. 188–195).
10. Debruyne, R., & Bessiere, C. (1997). Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI* (pp. 412–417).
11. Hansen, E. (1992). *Global optimization using interval analysis*. Marcel Dekker Inc.
12. Kieffer, M., Jaulin, L., Walter, E., Meizel, D. (2000). Robust autonomous robot localization using interval analysis. *Reliable Computing*, *3*(6), 337–361.

13. Lebbah, Y., Michel, C., Rueher, M., Daney, D., Merlet, J. (2005). Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis*, *42*(5), 2076–2097.
14. Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *IJCAI* (pp. 232–238).
15. Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, *8*, 99–118.
16. Merlet, J.P. (2007). Interval analysis and robotics. In *Symposium of Robotics Research*.
17. Messine, F. (1997). Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes. Ph.D. thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse.
18. Messine, F. (2002). Extensions of affine arithmetic: application to global optimization. *Journal of Universal Computer Science*, *8*(11), 992–1015.
19. Messine, F., & Laganouelle, J.L. (1998). Enclosure methods for multivariate differentiable functions and application to global optimization. *Journal of Universal Computer Science*, *4*(6), 589–603.
20. Min Li, C. (1997). Anbulagan: Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI* (pp. 366–371).
21. Tawarmalani, M., & Sahinidis, N.V. (2005). A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, *103*(2), 225–249.
22. Trombettoni, G., Araya, I., Neveu, B., Chabert, G. (2011). Inner regions and interval linearizations for global optimization. In *AAAI* (pp. 99–104).
23. Trombettoni, G., & Chabert, G. (2007). Constructive interval disjunction. In *Proceedings of CP*, LNCS 4741 (pp. 635–650).
24. Tucker, W. (2002). A rigorous ODE solver and smale's 14th problem. *Foundations of Computational Mathematics*, *2*, 53–117.
25. Van Hentenryck, P., Michel, L., Deville, Y. (1997). *Numerica : a modeling language for global optimization*: MIT Press.