Adaptive Constructive Interval Disjunction

Bertrand Neveu
LIGM
Université Paris Est
Marne-la-Vallée, France
Email: Bertrand.Neveu@enpc.fr

Gilles Trombettoni
LIRMM
Université Montpellier II
Montpellier, France
Email: Gilles.Trombettoni@lirmm.fr

***Abstract*—An operator called `CID` and an efficient variant `3BCID` were proposed in 2007. For numerical CSPs handled by interval methods, these operators compute a partial consistency equivalent to Partition-1-AC for discrete CSPs. The two main parameters of `CID` are the number of times the main `CID` procedure is called and the maximum number of sub-intervals treated by the procedure. The `3BCID` operator is state-of-the-art in numerical CSP solving, but not in constrained global optimization.**

This paper proposes an adaptive variant of `3BCID`. 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 be the best approach in solving and optimization, and has been added to the default strategies of the `Ibex` interval solver.

I. Constructive interval disjunction (CID)

A filtering/contracting operator for *numerical CSPs* called *Constructive Interval Disjunction* (in short `CID`) has been proposed in [13]. Applied first to continuous constraint satisfaction problems handled by interval methods, it has been more recently applied to constrained global optimization problems. This algorithm is state-of-the-art in constraint satisfaction, but is generally dominated by constraint propagation algorithms like `HC4` in optimization. The main practical contribution is 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.

A. Shaving

The shaving principle is used to compute the *Singleton Arc Consistency* (SAC) of finite domain CSPs [7] and the 3B-consistency of numerical CSPs [9]. It is also at the core of the SATZ algorithm [11] used to prove the satisfiability of Boolean formula. Shaving works as follows. 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 is not incremental [7]. 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 on finite-domain CSPs requires an expensive fixed-point algorithm where all the variables must be handled again every time a single value is removed [7]. The remark still holds for the improved version `SAC-Opt` [5]. A similar idea can be followed on numerical CSPs (NCSPs).

B. Numerical CSP

An NCSP 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 in an NCSP 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.

NCSPs are generally 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 NCSP.

Several contraction algorithms have been proposed. Let us mention the constraint propagation algorithm called `HC4` [3], [10], an efficient implementation of 2B [9], 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 AC3 is used to propagate reductions obtained for a given variable domain to the other constraints in the system.

C. 3B algorithm

Stronger interval partial consistencies have also been proposed. 3B-consistency [9] is a theoretical partial consistency similar to SAC for CSP although limited to the bounds of the domains. Consider the $2n$ subproblems of the studied NCSP 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.

D. CID

Constructive Interval Disjunction (CID) is a partial consistency stronger than 3B-consistency [13]. CID-consistency is similar to Partition-1-AC (P-1-AC) in finite domain CSPs [4]. P-1-AC is strictly stronger than SAC [4].

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.

Intuitively, CID generalizes 3B because a sub-box that is eliminated by var3B can also be 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.

var3BCID is a hybrid and operational variant of varCID.

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

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

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 $[X_{cid}]$.

The var3BCID process is illustrated in Figure 1.

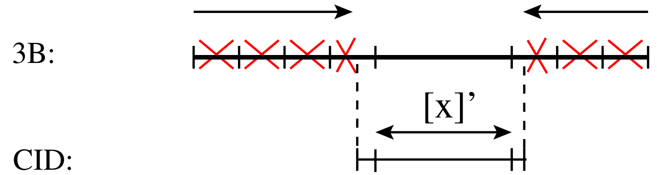

Figure 1. Task of the var3BCID procedure. The parameter s_{3b} is set to 10 and s_{cid} is set to 1.

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.

II. 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 [12]). Informally, the Smear function favors variables having a large domain and a high impact on the constraints – 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:

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

A. 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.

B. 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 number *kvarCID* of varcided variables giving the last significant improvement. 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 exceeded a small given ratio, called *ctratio*. This analysis starts from the last varcided variable. (For the readibility of the pseudocode, 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

```
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
```

Algorithm 2: Algorithm ACID1

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

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

C. 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.

III. Experiments

All the algorithms were implemented in the C++ interval library Ibex (Interval Based EXplorer) [6]. All the experiments were run on the same computer (Intel X86 3GHz). We tested the algorithms on square NCSP solving and constrained global optimization. NCSP solving 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.

A. 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 3600 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 II) 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 [8], and X-Newton [2]. *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 [12]). The main parameter *ctratio* of ACID1 and ACID2, measuring a stagnation in the filtering while variables are varcided, was fixed to 0.002. The var3BCID parameters s_{3b} and s_{cid} were fixed to the default settings, respectively 10 and 1, proposed in [13]. 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

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

Table I
CONTINUOUS CSP SOLVING: ACID1 RESULTS. FOR EACH PROBLEM, 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.

	#var	ACID1	ACID1	ACID1	best	worst	Time ratio	Time ratio
		time	#nodes	#varcids			ACID1/best	ACID1/worst
Bellido	9	3.45	518	5	ACID1	HC4	1	0.89
Brown-7	7	396	540730	4.5	ACID1	HC4	1	0.82
Brent-10	10	17.63	3104	9	ACID1	HC4	1	0.14
Butcher8a	8	981	204632	9	3BCID-n	HC4	1.03	0.49
Butcher8b	8	388	93600	10.8	ACID1	HC4	1	0.31
Design	9	29.22	5330	11	3BCID-n	HC4	1.07	0.37
Dietmaier	12	926	82364	26.3	ACID1	HC4	1	0.19
Directkin	11	32.73	2322	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	297116	12	ACID1	HC4	1	0.32
Fredtest	6	25.17	11480	0.8	HC4	3BCID-fp	1.04	0.91
Fourbar	4	437	183848	0.1	ACID1	3BCID-n	1	0.85
Geneig	6	178.2	83958	2.9	HC4	3BCID-fp	1.02	0.82
Hayes	7	3.96	1532	7.5	3BCID-n	HC4	1.14	0.77
I5	10	15.93	3168	11.5	ACID1	HC4	1	0.13
Katsura-25	26	691	5396	10.4	ACID1	3BCID-fp	1	0.67
Pramanik	3	23.1	23696	0.2	ACID1	HC4	1	0.69
Reactors-42	42	1285	23966	134	3BCID-fp	HC4	1.07	0.13
Reactors2-30	30	1220	38136	90	3BCID-n	HC4	1.14	0.12
Synthesis	33	356	7256	53.8	3BCID-fp	HC4	1.15	0.25
Trigexp2-23	23	2530	227136	39.4	3BCID-fp	HC4	1.26	0.25
Trigo1-18	18	2625	37756	6.1	ACID1	3BCID-fp	1	0.8
Trigo1sp-35	36	2657	70524	2.4	ACID1	3BCID-fp	1	0.41
Virasoro	8	1592	266394	0.6	3BCID-n	3BCID-fp	1.08	0.28
Yamamura1-16	16	2008	68284	0.4	3BCID-n	HC4	1.02	0.86
Yamamura1sp-500	501	1401	146	144	ACID1	HC4	1	0.14

shown in Table I, 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 II, 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 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 ($\neg$ 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.

B. *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 3600 s.

The time out was fixed to 3600s. We used the IbexOpt strategy of Ibex that performs a Best First Branch & Bound. The experimental protocol is the same as the NCSP 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 III reports the same columns as Table I, plus a column indicating the number of constraints of the instance. For the constraint programming part of IbexOpt, HC4 is state of the art and 3BCID is rarely needed in optimization.[3]

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

[3]In fact, the more recent Mohc constraint propagation algorithm [1] is better than HC4. Mohc is not yet reimplemented in Ibex 2.0. However, 3BCID(Mohc) shows roughly the same gains w.r.t. Mohc than 3BCID(HC4) does w.r.t. HC4...

Table II

NCSP: SOLVING TIME GAIN RATIOS. WE REPORT THE NUMBER OF PROBLEMS SOLVED BEFORE 3600 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 $\frac{acid1\ time}{C_i\ time}$

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2	ACID1	3BCID-fp	3BCID-n
							¬ XN	¬ XN	¬ XN
#solved instances $<$ 3600	26	20	23	24	25	24	20	16	20
#solved instances $<$ 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
Standard deviation gain	0	0.32	0.23	0.15	0.15	0.19	0	0.34	0.28
Total time	23594	>72192	37494	27996	26380	30428	29075	50181	31273
Total gain	1		0.63	0.84	0.89	0.78	1	0.58	0.93

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 IV 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 [12] (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.

IV. 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. 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. These variables are selected by an efficient branching heuristic and all the other parameters 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.

REFERENCES

[1] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI*, pages 9–14, 2010.

[2] I. Araya, G. Trombettoni, and B. Neveu. A Contractor Based on Convex Interval Taylor. In *Proc. CPAIOR*, volume 7298 of *LNCS*, pages 1–16. Springer, 2012.

[3] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, volume 5649 of *LNCS*, pages 230–244. Springer, 1999.

[4] H. Bennaceur and M.-S. Affane. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proc. CP*, volume 2239 of *LNCS*, pages 560–564. Springer, 2001.

[5] C. Bessiere and R. Debruyne. Optimal and Suboptimal Singleton Arc Consistency Algorithms. In *Proc. IJCAI*, pages 54–59, 2005.

[6] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173:1079–1100, 2009.

[7] R. Debruyne and C. Bessiere. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. IJCAI*, pages 412–417, 1997.

[8] E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker inc., 1992.

[9] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proc. IJCAI*, pages 232–238, 1993.

[10] F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. PhD thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse, 1997.

[11] C. Min Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. IJCAI*, pages 366–371, 1997.

[12] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert. Inner Regions and Interval Linearizations for Global Optimization. In *Proc. AAAI*, pages 99–104, 2011.

[13] G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP*, volume 4741 of *LNCS*, pages 635–650. Springer, 2007.

Table III
OPTIMIZATION PROBLEMS: ACID1 RESULTS

	#var	#ctr	ACID1 time	ACID1 #nodes	ACID1 #varcids	best	worst	Time ratio ACID1/best	Time ratio ACID1/HC4	Time ratio 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	1922	0.75	HC4	3BCID-fp	1.04	1.04	0.9
Ex5_4_4	27	19	915	23213	0.8	ACID1	3BCID-n	1	0.96	0.91
Ex6_1_1	8	6	60.85	13071	8.9	HC4	3BCID-fp	1.21	1.21	0.73
Ex6_1_3	12	9	297	29154	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	46687	0	HC4	3BCID-fp	1.02	1.02	0.74
Ex6_2_8	3	1	48.21	21793	0.1	HC4	3BCID-fp	1.01	1.01	0.72
Ex6_2_9	4	2	51.92	19517	0.1	HC4	3BCID-fp	1.02	1.02	0.72
Ex6_2_10	6	3	2248	569816	0	ACID1	3BCID-fp	1	0.99	0.64
Ex6_2_11	3	1	29.32	13853	0.3	HC4	3BCID-fp	1.05	1.05	0.73
Ex6_2_12	4	2	21.57	7855	0.1	HC4	3BCID-fp	1.02	1.02	0.8
Ex7_2_3	8	6	19.41	4596	4.4	3BCID-n	HC4	1.07	0.17	0.17
Ex7_2_4	8	4	36.79	5606	4.2	3BCID-fp	HC4	1.04	0.66	0.66
Ex7_2_8	8	4	37.98	6792	4.1	3BCID-n	HC4	1.09	0.71	0.71
Ex7_2_9	10	7	78.02	14280	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	46082	0.9	ACID1	3BCID-fp	1	0.99	0.87
Ex8_4_5	15	11	772	25454	4.8	HC4	3BCID-fp	1.03	1.03	0.75
Ex8_5_1	6	5	9.67	2138	2.75	ACID1	3BCID-fp	1	0.84	0.82
Ex8_5_2	6	4	32.46	5693	0.8	ACID1	3BCID-fp	1	0.9	0.87
Ex8_5_6	6	4	32.38	10790	1.8	HC4	3BCID-fp	1.02	1.02	0.76
Ex14_1_7	10	17	665	95891	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	5527	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	1098	13	3BCID-n	HC4	1.01	0.53	0.53
hhfair	28	25	26.59	3151	10	3BCID-n	HC4	1.12	0.58	0.58
himmel16	18	21	188	21227	15.5	3BCID-n	3BCID-fp	1.1	0.94	0.88
house	8	8	62.8	27195	3.25	HC4	3BCID-fp	1.09	1.09	0.79
hydro	30	24	609	32933	0	ACID1	3BCID-fp	1	0.88	0.78
immun	21	7	4.17	1317	2.5	ACID1	3BCID-fp	1	0.55	0.28
launch	38	28	107	2516	21	ACID1	3BCID-n	1	0.79	0.43
linear	24	20	751	27665	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	4658	4.3	ACID1	3BCID-fp	1	0.85	0.68
srcpm	38	27	160	6908	0.5	ACID1	3BCID-fp	1	0.62	0.33

Table IV
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
Standard deviation gain	0	0.25	0.16	0.18	0.12	0.14
Total time	9380	10289	12950	11884	11201	9646
Total gain	1	0.91	0.72	0.79	0.84	0.97