Adaptive Constructive Interval Disjunction
Bertrand Neveu, Gilles Trombettoni

▶ To cite this version:

HAL Id: hal-00936654
https://hal-enpc.archives-ouvertes.fr/hal-00936654
Submitted on 27 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
**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. For 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 (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 $[x_i] = [x_i, \overline{x}_i]$ the interval/domain of variable $x_i \in X$, where $x_i, \overline{x}_i$ 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 $+, \cdot, /, \exp, \log, \sin$.

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 $\overline{x}_i - x_i$ of an interval $[x_i]$. The width of a box is given by the width $\overline{x}_m - x_m$ 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 2\( n \) subproblems of the studied NCSP where each interval \([x_i]\) (\(i \in \{1..n\}\)) is reduced to its lower bound \(x_i\) (resp. upper bound \(\bar{x_i}\)). 3B-consistency is enforced iff each of these 2\( n \) 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.

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 killed, 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 \(3\text{BCID-fp}\) (fixed-point) this historical version.

Two reasons led us to radically change this policy. First, as said above, \(\text{var3BCID}\) 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 \(\text{var3BCID}\). 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, \(3\text{BCID}\) is clearly too expensive. Second, the \(\text{var3CID}\) principle is very 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 \(\text{varcided}\) when the procedure \(\text{var3BCID}\) is called on that variable to contract the current box.

To sum up, the idea for rendering \(3\text{BCID}\) even more efficient in practice is to replace the two nested loops by a single loop calling \(\text{numVarCID}\) times \(\text{var3BCID}\) and to use an efficient variant of the Smear function branching heuristic for selecting the variables to be varcided (called \(\text{SmearSumRel}^\ast\) 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 \(\text{numVarCID}\) to the number \(n\) of variables. We call \(3\text{BCID-n}\) 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 \(\text{numVarCID}\) 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 (\(\text{ACID}\)) is precisely to compute dynamically during search the value of the \(\text{numVarCID}\) 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 \(\text{var3BCID}\). They measure a contraction ratio of a box \([X]^b\) over another box \([X]^a\) as an average relative gain in all the dimensions:

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

A. \(\text{ACID0}\): auto-adapting \(\text{numVarCID}\) during search

The first version \(\text{ACID0}\) 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 \(\text{SmearSumRel}\) function (used for branching). Variables are then varcided until the cumulative contraction ratio during the last \(nv\) calls to \(\text{var3BCID}\) 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 \(nv = \max(3, \frac{n}{3})\) experimentally gave the best results. The experimental results are not bad but this policy prevents \(\text{numVarCID}\) 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. \(\text{ACID1}\): interleaving learning and exploitation phases

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

The behavior of \(\text{ACID1}\), shown in Algorithm 2, is the following:

- The variables are first sorted according to their impact measurement (using the \(\text{SmearSumRel}\) heuristic).
- During a learning phase (during \(\text{learnLength}\) nodes), we then analyze how the contraction ratio evolves from a \(\text{var3BCID}\) call to the next one, and store the number \(kvarCID\) of varcided variables necessary to obtain most of the possible filtering.
- More precisely, \(2\text{numVarCID}\) variables are varcided at each node (with a minimum value equal to 2, in case \(\text{numVarCID}\) = 0). In the first learning phase, we handle \(n\) variables. At the current node, the \(\text{lastSignificantGain}\) function returns the number \(kvarCID\) of varcided variables giving the last significant improvement. After the \(kvarCID\)th call to \(\text{var3BCID}\), the gain in current box size from a \(\text{var3BCID}\) call to the next one, computed by the \(\text{gainRatio}\) formula, never exceeded a small given ratio, called \(ctratio\). This analysis starts from the last varcided variable. (For the readability of the pseudo-code, we omit the parameters of the \(\text{var3BCID}\) procedure, i.e. \(s^b, s^c\), the constraints \(C\) and the contractor etc.)
- 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 \(\text{numVarCID}\). This value will be used by \(3\text{BCID}\) 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 \(\text{cycleLength}\) nodes in the search tree, both phases are called again.

Numerous variants of this schema were tested. In particular, it is counterproductive to learn \(\text{numVarCID}\) 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 free procedure. With these parameter values, the overhead of the learning phases (where we double the number of contractors) was run when a variable domain width is reduced (fixed-point) in which a new iteration on all the variables is run. 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 $s_{4id}$ 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.)

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\footnote{www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html} 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 $s_{4id}$ 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
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 ACID1 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 3BCID(Mohc) is not yet reimplemented in Ibex and 3BCID-fp is rarely needed in optimization.

B. Experiments in constrained global optimization

We selected in the series 1 of the Coconut constrained global optimization benchmark2 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

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

3In 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...
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


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 CI (one per column): the average, maximum, minimum and standard deviation values of this ratio ACID time

<table>
<thead>
<tr>
<th></th>
<th>ACID1</th>
<th>HC4</th>
<th>3BCID-fp</th>
<th>3BCID-n</th>
<th>ACID0</th>
<th>ACID2</th>
<th>ACID1</th>
<th>3BCID-fp</th>
<th>3BCID-n</th>
</tr>
</thead>
<tbody>
<tr>
<td>#solved instances &lt; 3600</td>
<td>26</td>
<td>20</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>#solved instances &lt; 10000</td>
<td>26</td>
<td>21</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>22</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Average gain</td>
<td>1</td>
<td>0.7</td>
<td>0.83</td>
<td>0.92</td>
<td>0.96</td>
<td>0.91</td>
<td>1</td>
<td>0.78</td>
<td>1.02</td>
</tr>
<tr>
<td>Maximum gain</td>
<td>1</td>
<td>0.13</td>
<td>0.26</td>
<td>0.58</td>
<td>0.45</td>
<td>0.48</td>
<td>1</td>
<td>0.18</td>
<td>0.38</td>
</tr>
<tr>
<td>Maximum loss</td>
<td>1</td>
<td>1.04</td>
<td>1.26</td>
<td>1.14</td>
<td>1.23</td>
<td>1.05</td>
<td>1</td>
<td>2.00</td>
<td>1.78</td>
</tr>
<tr>
<td>Standard deviation gain</td>
<td>0</td>
<td>0.32</td>
<td>0.23</td>
<td>0.15</td>
<td>0.15</td>
<td>0.19</td>
<td>0</td>
<td>0.34</td>
<td>0.28</td>
</tr>
<tr>
<td>Total time</td>
<td>23594</td>
<td>&gt;72192</td>
<td>37484</td>
<td>27996</td>
<td>26380</td>
<td>30428</td>
<td>29075</td>
<td>50181</td>
<td>31273</td>
</tr>
<tr>
<td>Total gain</td>
<td>1</td>
<td>0.63</td>
<td>0.84</td>
<td>0.89</td>
<td>0.78</td>
<td>1</td>
<td>0.58</td>
<td>0.93</td>
<td></td>
</tr>
</tbody>
</table>
Table III

<table>
<thead>
<tr>
<th>#var</th>
<th>#ct</th>
<th>ACID1</th>
<th>ACID1</th>
<th>ACID1</th>
<th>best</th>
<th>worst</th>
<th>Time ratio</th>
<th>Time ratio</th>
<th>Time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table IV**

<table>
<thead>
<tr>
<th>#solved instances</th>
<th>ACID1</th>
<th>HC4</th>
<th>3BCID-fp</th>
<th>3BCID-n</th>
<th>ACID0</th>
<th>ACID1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Average gain</td>
<td>1</td>
<td>0.9</td>
<td>0.77</td>
<td>0.88</td>
<td>0.91</td>
<td>0.97</td>
</tr>
<tr>
<td>Maximum gain</td>
<td>1</td>
<td>0.17</td>
<td>0.28</td>
<td>0.35</td>
<td>0.62</td>
<td>0.28</td>
</tr>
<tr>
<td>Maximum loss</td>
<td>1</td>
<td>1.47</td>
<td>1.04</td>
<td>1.23</td>
<td>1.18</td>
<td>1.19</td>
</tr>
<tr>
<td>Standard deviation gain</td>
<td>0</td>
<td>0.25</td>
<td>0.16</td>
<td>0.18</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>Total time</td>
<td>9380</td>
<td>10289</td>
<td>12950</td>
<td>11884</td>
<td>11201</td>
<td>9646</td>
</tr>
<tr>
<td>Total gain</td>
<td>1</td>
<td>0.91</td>
<td>0.72</td>
<td>0.79</td>
<td>0.84</td>
<td>0.97</td>
</tr>
</tbody>
</table>