Constructive Interval Disjunction

Gilles Trombettoni and Gilles Chabert

University of Nice-Sophia and COPRIN Project, INRIA, 2004 route des lucioles, 06902 Sophia.Antipolis cedex, B.P. 93, France
{trombe, gchabert}@sophia.inria.fr

Abstract. Shaving and constructive disjunction are two main refutation principles used in constraint programming. The shaving principle allows us to compute the singleton arc-consistency (SAC) of finite-domain CSPs and the 3B-consistency of numerical CSPs. Considering the domains as unary disjunctive constraints, one can adapt the constructive disjunction, proposed by Van Hentenryck et al. in the nineties, to provide another general-purpose refutation operator. One advantage over the shaving is that the partial consistency performed to refute values in the domains is not entirely lost.
This paper presents a new filtering operator for numerical CSPs, called `CID`, based on constructive disjunction, and a hybrid algorithm, called `3BCD`, mixing shaving and constructive disjunction. Experiments have been performed on 20 benchmarks. Adding `CID` to bisection, hull or box consistency, and interval Newton, produces a gain in performance of 1, 2 or 3 orders of magnitude on several benchmarks. `3BCD` and adaptive `CID` filtering algorithms with no additional parameters compare advantageously to the `3B-consistency` operator. Finally, the CID principle has led us to design a new splitting strategy.

1 Introduction

In constraint programming and operational research, *shaving* is based on a simple refutation principle. 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. This principle of refutation has two drawbacks. Contrarily to arc consistency, this consistency is not incremental [2]. Intuitively, the work of the underlying partial consistency algorithm on the whole subproblem explains why a single value can be removed. Thus, obtaining the *singleton arc consistency* on finite-domain CSPs requires an expensive fixed-point propagation algorithm where all the variables must be handled again every time a single value is removed [11]. SAC2 [1] and SAC-optim [2] and other SAC variants obtain better average or worst time complexity by managing heavy data structures for the supports of values (like with AC4) or by duplicating the CSP for every value. However, using these filtering operators inside a backtrack scheme is far from being competitive with the standard MAC algorithm in the current state of research. In its QuickShaving [7], Lhomme uses this shaving principle in a pragmatic way, i.e., with no overhead because the promising variables (i.e., those that can possibly produce gains with shaving in the future) are learnt during the search. Researchers and practitioners also use for a long time the shaving principle in scheduling problems. The variables are generally handled only once so that no fixed-point is reached neither. On numerical CSPs, the 2B-consistency is the refutation algorithm used by 3B-consistency [6] as arc-consistency is used to refute values in the SAC property. This weaker property limited to the bounds of intervals explains that 3B-consistency filtering can solve some systems very quickly (although it is counterproductive on a majority of numerical CSPs). The second drawback of shaving is that the pruning performed by the partial consistency operator to refute a given value is lost, which is not the case with constructive disjunction.

Constructive disjunction produces a significant filtering when dealing with disjunctions of constraints, and not only with conjunctions of constraints as in the standard CSP model [17]. The idea is to propagate independently every term of the disjunction, and to perform the union of the different pruned search spaces. In other terms, a value that is removed by every propagation process (run with one term/constraint of the disjunct) can be safely removed from the ground CSP. This idea is fruitful in several fields, such as in scheduling where a common constraint is that two given tasks cannot overlap, or in bin/strip packing optimization problems where two rectangles must not overlap.

It is known, but not so widespread, that constructive disjunction can also be used to handle the classical CSP model. Indeed, every variable domain can be viewed as a unary disjunctive constraint that imposes one value among the different possible ones ($x = v_1 \vee ... \vee x = v_n$, where x is a variable and $v_1, ..., v_n$ are the different values). In this specific case, similarly to shaving, the constructive disjunction principle can be applied as follows. Every variable in a domain is iteratively (and temporarily) assigned to a value (the other values are temporarily discarded), one computes a partial consistency on the corresponding subproblems and one computes the union of the resulting search spaces. This constructive "domain" disjunction is not very much exploited right now while it can sometimes produce impressive gains in performance. In particular, in addition to *all-diff* constraints [13], incorporating constructive domain disjunctions into the famous Sudoku problem (launched for instance when the variables/cases have only two remaining possible values/digits) often leads to a backtrack-free solving. The same phenomenon is observed with the shaving, but at an even higher cost [14].

This observation has precisely motivated the research described in this paper that studies how to apply constructive domain disjunction to numerical CSPs. In other terms, is the above intuition also true for numerical CSPs solved by interval solving techniques? The continuous nature of interval domains is particularly convenient for constructive domain disjunction. By splitting an interval into several smaller intervals, constructive domain disjunction leads in a straightforward way to the *constructive interval disjunction* (`CID`) filtering operator introduced in this paper.

After useful notations and definitions introduced in Section 2, Section 3 describes the *CID* partial consistency and the corresponding filtering operator. The experiments, presented in Section 7, have led us to design adaptive variants of the `CID` filtering operator with no additional parameters (see Section 4). A hybrid algorithm mixing shaving and CID is described in Section 5. Finally, a new CID-based splitting strategy is presented in Section 6.

2 Definitions

The algorithms presented in this paper aims at solving systems of equations.

Definition 1 *A* **numerical CSP** *(NCSP) $P = (X, C, B)$ contains a set of constraints C and a set X of n variables. Every variable $x_i \in X$ can take a real value in the interval $\mathbf{x_i}$ (the* **box** $\mathbf{B} = \mathbf{x_1} \times ... \times \mathbf{x_n}$*). A solution of P is an assignment of the variables in X such that all the constraints in C are satisfied.*

Because real numbers cannot be represented in computer architectures, the bounds of an interval $\mathbf{x_i}$ are floating-point numbers.

CID filtering performs a union operation between two boxes.

Definition 2 *Let B_l and B_r be two boxes corresponding to a same set C of constraints and a same set V of variables.*

The **hull** *(box) of B_l and B_r, denoted by* `Hull`*(B_l, B_r), is the minimal box including B_l and B_r.*

To compute a bisection point based on a new (splitting) strategy, we need calculate the *size* of a box. In this paper, the size of a box is given by its perimeter.

Definition 3 *Let $B = \mathbf{x_1} \times ... \times \mathbf{x_n}$ be a box. The* **size** *of B is $\sum_{i=1}^{n} \overline{\mathbf{x_i}} - \underline{\mathbf{x_i}}$, where $\overline{\mathbf{x_i}}$ and $\underline{\mathbf{x_i}}$ are respectively the upper and lower bounds of the interval $\mathbf{x_i}$.*

As mentioned in the introduction, the CID partial consistency has several common points with the well-known 3B-consistency partial consistency [6].

Definition 4 (3B-consistency)
Let $P = (X, C, B)$ be an NCSP.

Let $P_{\underline{x_i}}$ be P where the domain of $x_i \in X$ is reduced to its lower bound. Let $P_{\overline{x_i}}$ be P where the domain of $x_i \in X$ is reduced to its upper bound. Let $P'_{\underline{x_i}}$ be the closure of $P_{\underline{x_i}}$ by 2B-consistency. Let $P'_{\overline{x_i}}$ be the closure of $P_{\overline{x_i}}$ by 2B-consistency.

A variable x_i in X is **3B-consistent** *iff $P'_{\underline{x_i}}$ and $P'_{\overline{x_i}}$ are both non empty (i.e., iff the bounds cannot be refuted by 2B-consistency).*

The NCSP P is 3B-consistent iff all the variables in X are 3B-consistent.

The 2B-consistency (or hull-consistency) is a form of arc consistency restricted to the bounds of the domains [6].

For practical considerations, and contrarily to finite-domain CSPs, a partial consistency of an NCSP is generally obtained with a precision w [6]. This precision avoids a slow convergence to obtain the property. The definition above is generalized by considering respectively the intervals $[\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}} + w]$ and $[\overline{\mathbf{x_i}} - w, \overline{\mathbf{x_i}}]$ in $P_{\underline{x_i}}$ and $P_{\overline{x_i}}$.

When "subfiltering" is performed by *Box consistency*, instead of 2B-consistency, we obtain the so-called *Bound consistency* property [16].

3 CID-consistency

The CID-consistency is a new partial consistency that can be obtained on numerical CSPs. Following the principle given in the introduction, the CID(2)-consistency can be formally defined as follows.

Definition 5 (CID(2)-consistency)
Let $P = (X, C, B)$ be an NCSP. Let F be a partial consistency.

Let B_i^l the sub-box of B in which $\mathbf{x_i}$ is replaced by $[\underline{\mathbf{x_i}}, \check{\mathbf{x}}_\mathbf{i}]$ (where $\check{\mathbf{x}}_\mathbf{i}$ is the float in the middle of the interval $\mathbf{x_i}$). Let B_i^r the sub-box of B in which $\mathbf{x_i}$ is replaced by $[\check{\mathbf{x}}_\mathbf{i}, \overline{\mathbf{x_i}}]$.

A variable x_i in X is CID(2)-consistent w.r.t. P and F iff $B = Hull(F(X, C, B_i^l), F(X, C, B_i^r))$. The NCSP P is CID(2)-consistent iff all the variables in X are CID(2)-consistent.

For every dimension, the number of slices considered in the CID(2)-consistency is equal to 2. The definition can be generalized to the CID(s)-consistency in which every variable is split into s slices by `VarCID`.

In practice, like 3B-w-consistency, the CID consistency is obtained with a precision that avoids a slow convergence onto the fixed-point. We will consider that a variable is CID(2,w)-consistent if the hull of the corresponding left and right boxes resulting from sub-filtering reduces no variable more than w.

Definition 6 (CID(2,w)-consistency)
Let $P = (X, C, B)$ be an NCSP and $B' = Hull(F(X, C, B_i^l), F(X, C, B_i^r))$.

A variable x_i in X is CID(2,w)-consistent iff $\forall i \in [1..n], |\mathbf{x_i}| - |\mathbf{x'_i}| \leq w$, where $|\mathbf{x_i}|$ is the size of $\mathbf{x_i}$ in B and $|\mathbf{x'_i}|$ is the size of $\mathbf{x_i}$ in B'.
The NCSP P is CID(2,w)-consistent iff all the variables in X are CID(2,w)-consistent.

Algorithm `CID` details the CID(s,w)-consistency filtering algorithm. Like the 3B-consistency algorithm, `CID` iterates on all the variables until a stop criterion, depending on w, is reached. The procedure `VarCID` details the work on a given variable x_i. The interval of x_i is split into s slices of size $\frac{|\mathbf{x_i}|}{s}$ each by the procedure `SubBox`. The partial consistency operator F (e.g., 2B, Box-consistency) reduces the corresponding sub-boxes, and the union of the resulting boxes is computed by the `Hull` operator. Note that if the subfiltering operator F applied to a given subbox `sliceBox` detects an inconsistency, then `sliceBox'` is empty. This means that there is no use to perform the union of `sliceBox'` with the current box in construction.

```
Algorithm CID (s: number of slices, w: precision, in-out P = (X, C, B): an NCSP,
F: subfiltering operator and its parameters)
    repeat
        P_old ← P
        LoopCID (X, s, P, F)
    until StopCriterion(w, P, P_old)
end.
Procedure LoopCID (X, s, in-out P, F)
    for every variable xi ∈ X do
        VarCID (xi, s, P, F)
    end
end.
Procedure VarCID (xi, s, (X, C, in-out B), F)
    B' ← empty box
    for j ← 1 to s do
        sliceBox ← SubBox (j, s, xi, B) /* the j^th sub-box of B on xi */
        sliceBox' ← F(X, C, sliceBox) /* perform a partial consistency */
        B' ← Hull(B', sliceBox') /* Union with previous subboxes */
        B ← B'
    end
end.
```

The stop criterion related to w is given in Definition 6: the `Repeat` loop is interrupted when no variable interval has been reduced more than w. This stop criterion does not guarantee a convergence onto a *unique* fixed-point. Indeed, practionners of interval programming solvers know that, when a precision w is used, the final box depends on the order of the filtering operations. Moreover, from a theoretical point of view, in order that the stop criterion leads to a *non unique* fixed-point, it is necessary to return the box obtained just before the last filtering, i.e., the box in P_{old} in Algorithm `CID`. In the following, `CID(s,w)` denotes the `CID` operator obtaining the CID(s,w)-consistency.

Comparison with 3B-consistency

The `3B` algorithm follows a scheme similar to `CID`, in which `VarCID` is replaced by a shaving process, called `VarShaving` in this paper. In particular, both algorithms are not incremental, hence the outside repeat loop possibly reruns the treatment of all the variables, as shown in Algorithm `3B`.

The procedure `VarShaving` reduces the left and right bounds of variable x_i by trying to refute intervals with a width at least equal to ws. Starting from the interval of x_i, our implementation tries to iteratively refute the half part of the current interval (e.g., the left part when reducing the left bound) [6] if the left bound (i.e., a single float) can be removed by subfiltering.

The following proposition allows a better understanding of the difference between 3B filtering and CID filtering.

Algorithm `3B` *(w: stop criterion precision, ws : shaving precision, in-out* $P = (X, C, B)$*: an NCSP, F: subfiltering operator and its parameters)*

```
repeat
    for every variable xi ∈ X do
        VarShaving (xi, ws, P, F)
    end
until StopCriterion(w, P)
end.
```

Proposition 1 *Let* $P = (X, C, B)$ *be an NCSP. Let* F *be a partial consistency. Consider the box* B' *obtained by* `CID` *w.r.t.* F*, where* `VarCID` *splits intervals on every included float. Consider the box* B'' *obtained by* `3B` *w.r.t.* F*, where* `VarShaving` *splits intervals on every included float. Then,*

`CID` *filtering is stronger than* `3B` *filtering, i.e.,* B' *is included in or equal to* B''*.*

This theoretical property is based on the fact that, due to the hull operation in `VarCID`, the whole box B can be reduced on several, possibly all, dimensions. With `VarShaving`, the pruning effort can impact only x_i, losing all the temporary reductions obtained on the other variables by the different calls to F. [1]

In the general case however, if no assumption is made on ws and s, 3B-consistency and CID-consistency are not comparable. This has motivated the design of the hybrid `3BCD` operator decribed in Section 5. In practice, as shown by experiments, the number of slices s in `CID` and the precision ws in `3B` are specified such that the number of calls to F in `VarShaving` is generally greater to the number of calls to F in `VarCID`. In other words, on a given instance, the adequate parameters for the `CID` operator leads to a small number of slices. Since `CID` is often more efficient than `3B` both in terms of running time and pruning capacity (i.e., `CID` requires a smaller number of generated boxes to compute all the solutions), this clearly shows that it is better to perform a rough work on a given variable x_i (e.g., setting $s = 2$ or $s = 3$) and performing the union of the possible deductions than to perform a fine and more costly work on x_i and to forget the deductions on the other variables, as it is done by `3B`.

4 Adaptive variants of CID

We have presented a `CID(s,w)` operator with two additional parameters: the number of slices s and a fixed-point parameter w. Based on the feedback from experiments, we wanted to design filtering operators with no additional parameter. This section introduces such adaptive variants of `CID`.

[1] Note that optimized implementations of SAC reuse the domains obtained by subfiltering in subsequent calls to "`VarShaving`" [5].

4.1 Reaching a fixed-point is not fruitful

Although useful to compute the CID-consistency or 3B-consistency properties, the fixed-point repeat loop does not pay off in practice (see Section 7). In other words, running more than once `LoopCID` on all the variables is generally counterproductive, even if the value of w is finely tuned. We will call `CID(s)` (or `CID`(s,∞)) this simple variant of CID[2].

We have first envisaged that measuring the pruning effect on a single variable (with w) was not relevant. Let us remember that `LoopCID` is relaunched even if only one variable interval is reduced more than w. Indeed, as opposed to 2B or Box consistency, the CID-consistency and 3B-consistency are not incremental. There is thus no reason to apply the same stop criterion as in propagation algorithms (like 2B-consistency). That is why we have measured instead several dimensions simultaneously, i.e., the size of the box (perimeter or volume): a new `LoopCID` is run if the *reduction of box size* is sufficiently significant. Without detailing, the corresponding experimental results (not reported here) provide similar results as with the presented version, for `CID` as well as for `3B`. (We have also envisaged a second explanation that has led to another variant of CID whose detailed description will appear in the extended version of this paper [15].)

This analysis has led us to forget the w parameter in `CID`, i.e., to use `CID`(s,∞).

4.2 Increasing the number of slices is fruitful

As shown by the experiments, for a given instance, increasing the number s of slices often produces an additional pruning effect (clearly related to a smaller number of splits) until the induced overhead does not pay off w.r.t. running time. On the tested instances, small values for s (i.e., less than 8), often induce the best performance. However, even if the best number of slices is small, we have no idea of the specific value, so that the parameter s should be tuned.

As said above, another observation is that running the same `LoopCID` twice is generally counterproductive. This could mean that there is a few interest of performing a same `VarCID` (i.e., with the same number of slices) twice.

These two observations have led us to design a variant of `CID`(s,∞). In this variant, the number s of splits is modified between two splits: it is alternatively 2, 4 or 6 : $s = ((i \, modulo \, 3) + 1) \times 2)$, where i indicates the i^{th} call to `LoopCID`. It appears that `CID(2|4|6,`∞`)` generally outperforms `CID(`s_{best}`,`∞`)` called with the best number of slices (s_{best}), so that it seems not necessary in practice to tune the parameter s (see Section 7.4).

4.3 Adaptive CID-based strategies: `CID1` and `CID246`

The algorithms `CID1` and `CID246` are straightforward adaptations of the previous ideas. They use a splitting strategy, CID filtering and an interval Newton to find all the solutions of a numerical CSP. More precisely:

[2] It also appears that running only once `VarShaving` on all the variables in 3B is generally not so bad (see Section 7).

– The CID operator is `CID(2,∞)` in `CID1`, that is, `VarCID` performs only two slices on every variable, and `LoopCID` is called only once between two bisections.

  The CID operator is `CID(2|4|6,∞)` in `CID246` where `VarCID` performs 2, 4 or 6 slices alternatively (see Section 4.2).
– The subfiltering operator F is `Box` or `2B` with one fixed-point parameter `%w2B`.
– Between two bisections, two operations are run in sequence:
  1. a call to `CID`(2,∞) (or `CID(2|4|6,∞)`), that is, **one** call to `LoopCID`,
  2. and finally, a call to an Interval Newton operator.

Different combinations have been tried. Experiments (not reported here) have shown that the presented combination of operators is the best one, but it worthwhile noticing that several combinations are nearly as efficient as the chosen one. In particular, mixing 2B and an interval Newton in the sub-filtering F (i.e., inside CID filtering) produces interesting results as well, whereas such a combination with 3B-consistency is couterproductive.

In the following, we denote by `CID1(RR)`, `CID246(RR)` the algorithms described above when they are called with a *round-robin* splitting strategy. We denote by `CID1(LI)`, `CID246(LI)` these algorithms when the next variable to be split has the Largest Interval. `CID1(CIDBis)`, `CID246(CIDBis)` refer to a new bisection strategy based on CID filtering and described in Section 6.

5 The `3BCD` filtering algorithm

As mentioned above, the `3B-consistency` and `CID` filtering operators follow the same scheme, so that several hybrid algorithms can be imagined. One of them is presented below.

Until a stop criterion, related to a precision w, is fulfilled, `3BCD` performs two iterations. Constructive disjunction is first applied because it is not expensive (only s calls to subfiltering F per variable) and can filter on several variables simultaneously (in the same `VarCID` operation). The second iteration calls `VarShaving` on all the variables in order to perform left and right interval reductions on every variable.

The following property has motivated the design of `3BCD`.

Proposition 2 *Let $P = (X, C, B)$ be an NCSP obtained by a fixed-point* `3BCD(s,w)` *algorithm.*

P is both 3B(w)-consistent and CID(s,w)-consistent.

The proof is straightforward if we assume than the algorithm returns P_{old} and not P.

An alternative consists in calling `VarCID` and `VarShaving` in a unique `For` loop. Experimental results, not reported in this paper, have shown that this variant is less efficient, confirming thus that the CID principle produces a good filtering at a lower cost.

```
Algorithm 3BCD (s: number of slices, w: stop criterion and shaving precision, in-out
P=(X,C, in-out B): an NCSP, F: subfiltering operator and its parameters)
    repeat
        P_old ← P
        for every variable xi ∈ X do
            VarCID (xi, s, P, F)
        end
        for every variable xi ∈ X do
            VarShaving (xi, w, P, F)
        end
    until StopCriterion(w, P, P_old)
end.
```

More sophisticated variants could also be envisaged. An interesting hybridization would be to perform constructive interval disjunction "during" a shaving refutation. Unfortunately, the interval that is refuted by shaving and the complementary interval have generally very different sizes, making the approach not promising.

With 3BCD, the number of calls to F due to VarCID is in practice negligible as compared to the number of calls to F due to VarShaving (i.e., necessary to shave left and right bounds of intervals). Hence, 3BCD can be viewed as an improved version of 3B-consistency where constructive disjunction produces an additional pruning effect with a low overhead.

6 A new CID-based splitting strategy

There are three main splitting strategies (i.e., variable choice heuristics) used for solving numerical CSPs. The simplest one follows a *round-robin* strategy and loops on all the variables. Another heuristics selects the variable with the largest interval. A third one, based on the *smear function* [8], selects a variable x_i implied in equations whose derivative w.r.t. x_i is large.

The round-robin strategy ensures that all the variables are split in a branch of the search tree. Indeed, as opposed to finite-domain CSPs, note that a variable interval is generally split (i.e., instantiated) several times before finding a solution (i.e., obtaining a small interval of width less than the precision). The largest interval strategy also leads the solving process to not always select a same variable as long as its domain size decreases. The strategy based on the smear function sometimes splits always the same variables so that an interleaved schema with round-robin, or a preconditionning phase, is sometimes necessary to make it effective in practice.

We introduce in this section a new CID-based splitting strategy. Let us first consider different box sizes related to (and learnt during) the VarCID procedure applied to a given variable x_i :

– Let $\texttt{OldBox}_i$ be the box B just before the call to `VarCID` on x_i. Let $\texttt{NewBox}_i$ be the box obtained after the call to `VarCID` on x_i.
– Let $B_i^{l'}$ and $B_i^{r'}$ be the left and right boxes computed in `VarCID`, after a reduction by the F filtering operator, and before the `Hull` operation. Let B_i^{max} be the box with the maximal size among $B_i^{l'}$ and $B_i^{r'}$, and let B_i^{min} be the other box, i.e., the box with the smaller size.

The sizes of these boxes provide two interesting measures. A first ratio is `ratioCID`$= \frac{\text{Size(NewBox)}}{\text{Size(OldBox)}}$. This ratio measures the pruning obtained by `VarCID`. Although interesting in theory, this ratio only yields an indication concerning the past, i.e., concerning the pruning effect we have obtained. However, it provides no clear indication about the future (see [15]), so that we have not exploited `ratioCID`.

The second measure leads to an "intelligent" splitting strategy. The ratio `ratioBis`$= \frac{f(Size(B_i^{l'}), Size(B_i^{r'}))}{Size(NewBox)}$ in a sense computes the size lost by the (box) `Hull` operation of `VarCID`. In other words, $B_i^{l'}$ and $B_i^{r'}$ represent precisely the boxes one would obtain if one split the variable x_i (instead of performing the `hull` operation) immediatly after the call to `VarCID`; `NewBox` is the box obtained by the `Hull` operation used by `CID` to avoid a combinatorial explosion due to a choice point.

Thus, after a call to `LoopCID`, the CID principle allows us to learn about a good variable interval to be split: *one selects the variable having led to the lowest* `ratioBis`. This splitting strategy is called `CIDBis` (for CID-based Bisection). Although not related to disjunctive construction, similar strategies have been applied to finite-domain CSPs [3, 12].

In our experiments, we have chosen `ratioBis`$= \frac{\text{Size}(B_i^{max})+0.1\times\text{Size}(B_i^{min})}{\text{Size}(NewBox))}$. Indeed, solving a numerical CSP is NP-complete and, even in practice, the time does generally not grow linearly with the size. Thus, in case of bisection, the time is generally dominated by the time required for solving the largest subtree among the two ones. Because of the 0.1 factor, we have $\frac{\text{Size}(B_i^{max})}{\text{Size}(NewBox))} \leq$ `ratioBis` $\leq 1.1 \times \frac{\text{Size}(B_i^{max})}{\text{Size}(NewBox)}$. This allows the splitting strategy to break ties in case two variables have approximately the same maximum box size (i.e., when the difference in size between both is less than 10%).

7 Experiments

We have performed a lot of comparisons and tests on a sample of 20 instances. These tests have helped us to confirm several intuitions and to design efficient variants of CID filtering.

7.1 Benchmarks and tuned parameters

Twenty benchmarks are briefly presented in this section. Five of them are sparse systems found in [9]. They are challenging for general-purpose interval-based techniques, but the algorithm IBB can efficiently exploit a preliminary decomposition of the systems into small subsystems [9]. The other benchmarks have been found in the Web page of the COPRIN research team or in the COCONUT Web page where the reader can find more details about them [10]. All the selected instances can be solved in an acceptable amount of time by a standard algorithm in order to make possible comparisons between different variants. No selected benchmark has been discarded for any other reason!

Name	n	**#sols**	**Ref.**	**Precision**	**Subflt.**	**%w2B**	**%w2B in CID**	ws	w
BroydenTri	32	2	[10]	1e-08	2B	10%	10%	1e-02	1e-02
Hourglass	29	8	[9]	1e-08	2B	20%	5%	1e-02	1e-02
Tetra	30	256	[9]	1e-08	2B	5%	10%	1e-03	1e-03
Tangent	28	128	[9]	1e-08	2B	30%	50%	10	10
Reactors	20	38	[10]	1e-08	2B	10%	10%	1e-01	1
Trigexp1	30	1	[10]	1e-08	2B	20%	20%	1	1
Discrete25	27	1	[10]	1e-08	2B	0.1%	1%	1e-01	1e-01
I5	10	30	[10]	1e-08	2B	5%	5%	1e-03	1e-03
Transistor	12	1	[10]	1e-08	2B	10%	10%	1e-01	1
Ponts	30	128	[9]	1e-08	2B	10%	10%	10	10
Yamamura8	8	7	[10]	1e-08	Box+2B	1%	1%	1e-02	1e-02
Design	9	1	[10]	1e-08	2B	10%	10%	1e-01	1e-01
D1	12	16	[10]	1e-08	2B	10%	10%	1e-01	1
Mechanism	98	448	[9]	5e-06	2B	0.5%	1%	1e-01	1e-01
Kinematics1	6	16	[10]	1e-08	2B	10%	10%	1	10
Hayes	8	1	[10]	1e-08	2B	1%	50%	1e-02	1e-02
Eco9	8	16	[10]	1e-08	2B	20%	20%	1	1
Trigexp2	5	1	[10]	1e-08	2B	10%	10%	1	10
Bellido	9	8	[10]	1e-08	2B	10%	10%	1e-01	1e-01
Caprasse	4	18	[10]	1e-08	2B	5%	5%	1	10

Table 1. The tested benchmarks. n is the number of variables; #sols is the number of solutions; **Ref.** indicates the reference in which the reader can get a more precise description of the system; **Precision** is the size of interval under which a variable interval is not split; **Subflt.** designs the filtering algorithm used in 3B or in CID; %w2B is a user-defined parameter used by 2B or Box: a constraint is not pushed in the propagation queue if the projection on its variables has reduced the corresponding intervals less than %w2B (percentage of the interval width); %w2B **in CID** indicates the same parameter when 2B or Box is used as subfiltering inside CID; ws is the width parameter used in VarShaving while w is used to stop the outside loop. w is also used by CID(s, w).

Note that CID filtering generally uses a smaller precision in subfiltering, i.e., a larger parameter %w2B, than 2B alone.

Although not reported in this paper, it appears that %w2B does not need to be finely tuned in CID. For instance, setting %w2B to 10% always produces good results.

7.2 Interval-based solver

All the tests have been performed on a `Pentium IV 2.66 Ghz` using the interval-based library in `C++` developed by the second author. This new solver provides the main standard interval operators such as `Box` filtering, `2B-consistency` filtering, interval Newton [8]. The solver provides a round-robin, a largest-interval and a `CIDBis` splitting strategies. Although recent and under developement, the library seems competitive with up-to-date solvers like `RealPaver` [4]. The reader can refer to [9] to have a first evaluation of it on several sparse equation systems.

The implementation of `3B-consistency` filtering is rather sophisticated. It uses splitting and interleaves refutation tests on floats and slices. In our implementation, we distinguish two precision parameters: ws used in `VarShaving` and w used to interrupt the outside loop.

For all the presented solving techniques, including `3B` and `3BCD`, an interval Newton is called just before a splitting operation iff the width of the largest variable interval is less than $1e-2$.

Note that the parameter `%w2B` used in the subfiltering operator has been finely tuned to offer the best performance for 2B/Box+Newton and CID.

7.3 Comparing `CID` and shaving

The main conclusions deduced from Table 2 are the following:

- The drastic reduction in the number of required bisections (often several orders of magnitude) clearly underlines the filtering power of CID: Best CID or 3BCD always obtains the lowest number of splits. 3B is rather competitive with Best CID or 3BCD on only six benchmarks.
- 3BCD always outperforms 3B, except on `Yamamura8`, `Trigexp2` and `Caprasse`. In this case, the loss in running time is not significant.
- `CID(2,w)` is always worse than `CID1(2,`∞`)`, except for `Trigexp1` and `D1`. This explains why we have removed the w fixed-point parameter in subsequent variants of CID.
- This phenomenon is generally true for 3BCD and sometimes also for 3B.
- Best CID outperforms almost all the other algorithms. Only 3B or 3BCD are slightly better on `BroydenTri` and `Trigexp1`. Moreover, 2B+Newton (see column 1) is slightly better on `Bellido` and `Trigexp2`, and better on `Caprasse`. Note that these benchmarks, especially `Trigexp2` and `Caprasse`, have a very small number of variables. Thus, a non expensive filtering algorithm is fruitful because there is no combinatorial explosion due to bisection.
- Impressive gains in running time are obtained by Best CID for the benchmarks on the top of the table.

Note that the three algorithms behind Best CID have no additional parameter, as opposed to 3B with its ws (and/or w) parameter.

Name	2B/Box	3B(w)	3B(∞)	3BCD(w)	3BCD(∞)	CID(2,w)	CID1(2,∞)	**Best** CID
BroydenTri	2910	**0.14**	**0.14**	0.15	**0.14**	6.67	0.39	0.26
	2e+07	**2**	5	**2**	**2**	1102	168	42
Hourglass	29	5.3	4	1.71	1.09	0.81	0.54	**0.44**
	81134	1684	1416	**26**	31	132	156	62
Tetra	433	129	83	30.3	20.5	19.2	14.3	**12.4**
	9e+05	12352	11019	1362	1405	1728	2458	**804**
Tangent	43.1	81.6	81.8	13.8	24.6	34.4	32.5	**3.47**
	2e+05	1e+05	1e+05	5506	10738	15955	15113	**645**
Reactors	131	60	50	34	30	27	23	**9.3**
	7e+05	57896	47302	3250	3893	5090	7144	**1361**
Trigexp1	3.4	0.24	0.23	**0.18**	0.19	0.19	0.27	0.2
	5025	**1**	**1**	**1**	2	**1**	20	7
Discrete25	6.5	3.76	4.28	2.92	3.68	3.95	1.9	**0.98**
	1923	**1**	3	**1**	2	38	78	18
I5	708	616	454	597	427	399	251	**142**
	3e+06	57900	59204	**10795**	14434	53633	97605	24541
Transistor	137	216	221	152	155	152	147	**35**
	6e+05	3e+05	3e+05	48018	49011	62787	61019	**6970**
Ponts	11.4	11.1	9.51	5.38	4.14	4.33	3.7	**2.75**
	23818	4315	4142	333	415	488	710	**170**
Yamamura8	13.2	12.32	10.68	16.78	12.22	22.88	17.27	**5.27**
	1032	197	307	49	79	117	142	**44**
Design	444	994	896	665	591	455	434	**234**
	3e+06	1e+06	1e+06	2e+06	2e+06	2e+05	2e+05	**63454**
D1	4.54	6.87	7.24	3.17	3.15	**2.33**	2.57	2.48
	34888	19716	20623	1689	1626	1426	1624	**682**
Mechanism	111	390	333	227	227	195	187	**83.9**
	24538	6819	6781	**1811**	2299	2647	3229	5386
Kinematics1	94	248	248	116	116	94	94	**76**
	7e+05	7e+05	7e+05	9160	9160	9180	9180	**4060**
Hayes	155	191	197	141	151	126	126	**124**
	3e+05	2e+05	2e+05	1e+05	1e+05	1e+05	1e+05	**79780**
Eco9	25.3	53.5	43.7	51.8	40.9	29.7	27.6	**24.9**
	3e+05	70982	71833	**19383**	21043	22881	25195	24091
Bellido	**92**	239	194	214	169	143	126	98.6
	7e+05	3e+05	3e+05	69678	78473	98137	1e+05	**42724**
Trigexp2	**3.45**	5.9	5.9	8.6	8.6	6.77	6.77	4.46
	11428	11322	11322	5871	5871	3445	3445	**2030**
Caprasse	**2.68**	6.71	6.98	7.09	7.3	5.16	5.16	5.14
	31196	21649	21525	8801	**8701**	9416	9416	9000

Table 2. First comparison between 2B, 3B, 3BCD and CID. The column 2B/Box reports the results obtained by filtering with 2B or 2B+Box, followed by a call to an interval Newton and a round-robin splitting. A similar combination is used with 3B or 3BCD in the following four columns. A parameter $w = \infty$ means that only one LoopCID (or "LoopShaving") is called between two bisections in 3B, 3BCD or CID. Best CID reports the best result obtained by CID1(CIDBis), CID246(CIDBis) or CID246(RR) (see Table 4). Every cell contains two values: the CPU time in seconds to compute all the solutions (top), and the number of required bisections (bottom). For every benchmark, the best result is bold-faced.

Name	CID(2,∞)	CID(3,∞)	CID(4,∞)	CID(8,∞)	CID(16,∞)	CID(2|4|6, ∞)	**Sensitivity**
BroydenTri	0.39	0.74	0.46	**0.23**	0.24	0.31	Yes
	168	130	50	16	10	35	
Hourglass	0.54	**0.52**	0.54	0.56	0.81	0.56	No
	156	90	74	38	28	76	
Tetra	14.3	**13.1**	14.7	22.2	34.7	14.7	No
	2458	1492	1306	1122	1026	1336	
Tangent	32.4	5.1	3.7	4.7	6.3	**3.5**	Very
	15113	1333	653	485	357	645	
Reactors	22.9	17.3	**17.2**	17.4	26.1	18.1	Yes
	7144	3157	2127	1079	808	2311	
Trigexp1	0.26	0.34	0.16	**0.11**	0.12	0.25	Yes
	20	16	4	1	1	8	
Discrete25	1.9	1.33	**0.8**	0.9	1.5	0.99	Yes
	78	35	14	5	3	20	
I5	250	177	148	142	174	**141**	Yes
	97605	41905	25339	11805	7294	24541	
Transistor	147	115	89	**72**	77	82	Yes
	61019	28420	15838	6179	3312	15171	
Ponts	3.73	3.22	3.21	4.35	6.54	**3.04**	Yes
	710	364	304	270	266	304	
Yamamura8	17.3	**12.8**	13.7	16.6	22	14.1	Yes
	142	70	54	28	20	55	
Design	435	340	317	310	412	**297**	Yes
	229545	112742	77022	36502	24139	74244	
D1	2.57	**1.80**	**1.80**	2.70	3.18	2.48	Yes
	1624	672	484	298	190	682	
Mechanism	186	**171**	184	181	194	181	No
	3229	1993	2012	1323	986	1963	
Kinematics1	94	95	90	100	122	**85**	Yes
	9180	5837	4049	2246	1356	3965	
Hayes	**124**	134	126	141	193	**124**	No
	138310	108233	79609	47989	33848	79780	
Eco9	27.6	**26.2**	27.3	35.8	55.8	28.5	No
	25195	15781	12145	7995	6231	13133	
Bellido	126	110	**105**	116	159	107	Yes
	110713	61657	43908	24324	16922	45823	
Trigexp2	6.9	7.5	6.7	7.9	6.7	8.9	No
	3445	2760	1825	1023	446	2494	
Caprasse	**5.19**	**5.19**	5.38	6.78	10.49	5.53	No
	9416	6572	5300	3456	2704	5608	

Table 3. Performance of CID1 with a round-robin splitting strategy and various numbers s of slices in CID filtering. CID246(RR) is CID246 with a round-robin splitting strategy. The column **Sensitivity** indicates whether increasing the number of slices has a positive impact on running time.

7.4 Playing with slices

The following conclusions can be drawn from Table 3.

- Playing with the number of slices in `CID1` has a positive impact on 13 of the 20 instances. The impact on `Tangent` is significant.
- The "optimal" number of slices is generally small, typically 3 or 4. Selecting $s = 8$ is better for `BroydenTri`, `Trigexp1` and `Transistor`.
- `CID246` is a judicious variant of `CID1`. Its running time is rarely far from the time of `CID1` with the optimal s. Its running time is even sometimes better than the time of `CID1` with the optimal s (6 instances among 20).

7.5 Comparison between splitting strategies

Table 4 applies the three available splitting strategies to `CID1` and `CID246`. We underline some observations.

- `CID1(RR)` never produces the best running CPU time (except for `Hayes` although `CID1(CIDBis)` turns out to be as efficient as it).
- `CID246` is better than `CID1` on 15 on 20 instances. `Tangent` and `Mechanism` have opposite behaviors.
- The new `CIDBis` splitting is better than the other strategies on 12 of the 20 instances. The largest interval strategy is the best on 4 instances. The round-robin strategy is the best on 4 instances.
- On the 8 instances for which `CIDBis` is not the best strategy, the loss in performance is significant on 4 of them: `BroydenTri`, `Tangent`, `I5` and `Hayes`.
- `CID1` is sometimes very bad with the largest interval strategy (see `Tangent` and `Hayes`).

8 Conclusion

This paper has introduced a new filtering operator based on the constructive disjunction principle exploited in combinatorial problems. This operator also opens the door to a new splitting strategy, called `CIDBis` learning from the work of CID filtering. The first experimental results are very promising and we believe that `CID1` or a variant has the potential to become a standard operator in interval constraint solvers. Note that the number of additional user-defined parameters is null. More precisely, we can consider that this paper has brought a new "metaCID" operator with two possible CID-based filtering (`CID1` or `CID246`) and an additional splitting strategy (`CIDBis`). The `%w2B` parameter used in CID subfiltering can be arbitrarily set to 10% with no significant loss in performance.

Bisection is a combinatorial way to make an assumption about the variable values. On the opposite, constructive interval disjunction, like 3B-consistency, can be viewed as a polynomial way to do it. This explains the fine analysis that should be performed about CID filtering and about a relevant splitting strategy. Thus, a future work is to better exploit the result of a `VarCID` operation.

Filtering	**2B/Box**	**3B**	CID1	CID246	CID1	CID246	CID1	CID246
Splitting	RR	RR	RR	RR	Largest I.	Largest I.	CID based	CID based
BroydenTri	2910	0.14	0.39	0.31	7.06	**0.18**	9.34	0.26
	2e+07	2	168	35	3120	31	4305	42
Hourglass	29	4	0.54	0.56	2.69	0.52	0.79	**0.44**
	81134	1416	156	76	888	56	270	62
Tetra	433	83	14.3	14.7	51.3	25.7	17.85	**12.42**
	9e+05	11019	2458	1336	10564	2008	3406	804
Tangent	43.1	81.6	32.4	**3.5**	2388	22	329	15.2
	2e+05	1e+05	15113	645	1.5e+06	4840	207430	3148
Reactors	131	50	23	18	14.9	12.3	17	**9.3**
	7e+05	47302	7144	2311	5413	1793	6183	1361
Trigexp1	3.4	0.23	0.26	0.25	0.21	0.22	0.27	**0.20**
	5025	1	20	8	13	6	23	7
Discrete25	6.5	3.76	1.9	0.99	1.22	1.52	1.55	**0.98**
	1923	1	78	20	37	21	58	18
I5	708	454	250	**141**	742	416	389	245
	3e+06	59204	97605	24541	297843	78527	156399	46389
Transistor	137	216	147	82	91	**33**	61	35
	6e+05	3e+05	62787	15171	41755	6895	27886	6970
Ponts	11.4	9.51	3.73	3.04	7.69	5.48	3.53	**2.75**
	23818	4142	710	304	1852	520	652	170
Yamamura8	13.2	10.68	17.3	14.1	**4.88**	6.91	5.27	8.03
	1032	307	142	55	39	24	44	27
Design	444	896	435	297	809	337	454	**234**
	3e+06	1e+06	229545	74244	425011	94924	244930	63454
D1	4.54	6.87	2.57	**2.48**	3.97	2.87	3.48	2.52
	34888	19716	1624	682	2424	728	2110	626
Mechanism	111	333	186	181	**81**	152	84	157
	24538	6781	3229	1963	4406	1011	5386	1340
Kinematics1	94	248	94	85	96	79	91	**76**
	69398	69034	9180	3965	9454	4127	8958	4060
Hayes	155	191	**124**	**124**	1198	567	554	501
	3e+05	2e+05	138310	79780	949034	285626	420747	252542
Eco9	25.3	43.7	27.6	28.5	29.3	31.1	**24.9**	28.4
	3e+05	71833	25195	13133	28185	15298	24091	13870
Bellido	92	194	126	107	110	103	103.7	**98.6**
	7e+05	3e+05	110713	45823	94957	43082	91484	42724
Trigexp2	3.45	5.87	6.77	8.92	5.41	5.74	**4.46**	4.96
	11428	11322	3445	2494	2345	1399	2030	1342
Caprasse	2.68	6.71	5.19	5.53	5.47	5.93	**5.14**	5.40
	31196	21649	9416	5608	9120	5300	9000	5400

Table 4. Comparison between CID1 and CID246 with three splitting strategies: round-robin (RR), largest interval (LI) and the new CID-based strategy (CIDBis).

According to the CID pruning obtained on a given variable x_i, when should we filter again x_i (with CID)? Is x_i a good candidate for the next bisection? A first tool, based on a variant of CID, called ACID, is presented in the extended paper to investigate these questions [15].

A near-term future work is of course to use CID techniques to solve challenging benchmarks.

Acknowledgements

Special thanks to Olivier Lhomme for useful comments on the paper. Also thanks to the anonymous reviewers.

References

1. R. Barták and R. Erben. A new Algorithm for Singleton Arc Consistency. Proc. FLAIRS'04. 2004.
2. C. Bessière and Debruyne R. Optimal and Suboptimal Singleton Arc Consistency Algorithms. Proc. IJCAI'05. pages 54–59, 2005.
3. P.A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *Proc. ECAI'92*, pages 31–35, 1992.
4. L. Granvilliers. Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, , Accepted for publication.
5. C. Lecoutre and S. Cardon. A Greedy Approach to Establish Singleton Arc Consistency. In *Proc. of IJCAI'05*, pages 199–204, 2005.
6. O. Lhomme. Consistency Tech. for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.
7. O. Lhomme. Quick Shaving. pages 411–415, 2005.
8. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
9. B. Neveu, G. Chabert, and G. Trombettoni. When Interval Analysis helps Interblock Backtracking. In *Proc. CP'06, LNCS 4204*, pages 390–405, 2006.
10. Web page of COPRIN: www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html COCONUT benchs: www.mat.univie.ac.at/ neum/glopt/coconut/Benchmark/Benchmark.html.
11. Debruyne R. and C. Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. Proc. IJCAI'97. pages 412–417, 1997.
12. P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proc. CP'04, LNCS 3258*, pages 557–571, 2004.
13. J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. Proc. AAAI'94. pages 362–367, 1994.
14. H. Simonis. Sudoku as a Constraint Problem. In CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems. pages 13–27, 2005.
15. G. Trombettoni and G. Chabert. Constructive Interval Disjunction. Technical report, INRIA, 2006. Extended version of this paper, in preparation.
16. P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica : A Modeling Language for Global Optimization*. MIT Press, 1997.
17. P. Van Hentenryck, V. Saraswat, and Deville Y. Design, Implementation, and Evaluation of the Constraint Language CC(FD). *J. Logic Programming*, 37(1–3):139–164, 1994.