

Improving Inter-Block Backtracking with Interval Newton

Bertrand Neveu, Gilles Trombettoni, Gilles Chabert

▶ **To cite this version:**

Bertrand Neveu, Gilles Trombettoni, Gilles Chabert. Improving Inter-Block Backtracking with Interval Newton. Constraints, 2010, 15 (1), pp.93-116. 10.1007/s10601-009-9072-5 . hal-00481324

HAL Id: hal-00481324
https://hal.science/hal-00481324

Submitted on 8 Oct 2013

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.

Constraints (2010) 15:93–116
DOI 10.1007/s10601-009-9072-5

Improving inter-block backtracking with interval Newton

Bertrand Neveu · Gilles Trombettoni · Gilles Chabert

Published online: 20 May 2009
© Springer Science + Business Media, LLC 2009

Abstract Inter-block backtracking (`IBB`) computes all the solutions of sparse systems of nonlinear equations over the reals. This algorithm, introduced by Bliek et al. (1998) handles a system of equations previously decomposed into a set of (small) $k \times k$ sub-systems, called blocks. Partial solutions are computed in the different blocks in a certain order and combined together to obtain the set of global solutions. When solutions inside blocks are computed with interval-based techniques, `IBB` can be viewed as a new interval-based algorithm for solving decomposed systems of nonlinear equations. Previous implementations used Ilog Solver and its `IlcInterval` library as a black box, which implied several strong limitations. New versions come from the integration of `IBB` with the interval-based library `Ibex`. `IBB` is now reliable (no solution is lost) while still gaining at least one order of magnitude w.r.t. solving the entire system. On a sample of benchmarks, we have compared several variants of `IBB` that differ in the way the contraction/filtering is performed *inside* blocks and is shared *between* blocks. We have observed that the use of interval Newton inside blocks has the most positive impact on the robustness and performance of `IBB`. This

B. Neveu
INRIA CERTIS, 2004 route des lucioles, BP 93, 06902 Sophia Antipolis Cedex, France
e-mail: Bertrand.Neveu@sophia.inria.fr

G. Trombettoni (✉)
INRIA Université de Nice-Sophia, 2004 route des lucioles, BP 93,
06902 Sophia Antipolis Cedex, France
e-mail: trombe@sophia.inria.fr

G. Chabert
LINA Ecole des Mines de Nantes, 2004 route des lucioles, BP 93,
06902 Sophia Antipolis Cedex, France
e-mail: Gilles.Chabert@emn.fr

modifies the influence of other features, such as intelligent backtracking. Also, an incremental variant of inter-block filtering makes this feature more often fruitful.

Keywords Intervals · Decomposition · Solving sparse systems

1 Introduction

Interval techniques are promising methods for computing all the solutions of a system of nonlinear constraints over the reals. They are general-purpose and are becoming more and more efficient. They are having an increasing impact in several domains such as robotics [21] and robust control [12]. However, it is acknowledged that systems with hundreds (sometimes tens) nonlinear constraints cannot be tackled in practice.

In several applications made of nonlinear constraints, systems are sufficiently sparse to be decomposed by equational or geometric techniques. CAD, scene reconstruction with geometric constraints [26], molecular biology and robotics represent such promising application fields. Different techniques can be used to decompose such systems into $k \times k$ *blocks*. Equational decomposition techniques work on the *constraint graph* made of variables and equations [2, 15]. The simplest equational decomposition method computes a maximum matching of the constraint graph. The strongly connected components (i.e., the cycles) yield the different blocks, and a kind of triangular form is obtained for the system. When equations model geometric constraints, more sophisticated geometric decomposition techniques generally produce smaller blocks. They work directly on a geometric view of the entities and use a rigidity property [10, 13, 15].

Once the decomposition has been obtained, the different blocks must be solved in sequence. An original approach of this type has been introduced in 1998 [2] and improved in 2003 [23]. *Inter-Block Backtracking* (`IBB`) follows the partial order between blocks yielded by the decomposition, and calls a solver to compute the solutions in every block. `IBB` combines the obtained partial solutions to build the solutions of the problem.

1.1 Contributions

The new versions of `IBB` described in this paper make use of our new interval-based library called `Ibex` [4, 5].

- `Ibex` allows `IBB` to become reliable (no solution is lost) while still gaining one or several orders of magnitude w.r.t. solving the system as a whole.
- An extensive comparison on a sample of decomposed numerical CSPs allows us to better understand the behavior of `IBB` and its interaction with interval analysis.
- The use of an interval Newton operator inside blocks has the most positive impact on the robustness and performance of `IBB`. Interval Newton modifies

the influence of other features, such as intelligent backtracking and filtering on the whole system (inter-block filtering—IBF).
- An incremental implementation of inter-block filtering leads to a better performance.
- It is counterproductive to filter inside blocks with `3B` [19] rather than with `2B`. However, first experiments show that using `3B` only inside large blocks might be fruitful.

2 Assumptions

We assume that the systems have a finite set of solutions. This condition also holds on every sub-system (block), which allows `IBB` to combine together a finite set of partial solutions. Usually, to produce a finite set of solutions, a system must contain as many equations as variables. In practice, the problems that can be decomposed are often under-constrained and have more variables than equations. However, in existing applications, the problem is made square by assigning an initial value to a subset of variables called *input parameters*. The values of input parameters may be given by the user (e.g., in robotics, the degrees of freedom, determined during the design of the robot, serve to pilot it), read on a sketch, or are given by a preliminary process (e.g., in scene reconstruction [26]).

3 Description of `IBB`

`IBB` works on a **Directed Acyclic Graph** of blocks (in short **DAG**) produced by any decomposition technique. A **block** i is a sub-system containing equations and variables. Some variables in i, called **input variables** (or parameters), are replaced by values when the block is solved. The other variables are called **(output) variables**. There exists an arc from a block i to a block j iff an equation in j involves at least one input variable assigned to a "value" in i. The block i is called a parent of j. The `DAG` implies a partial order in the solving process.

3.1 Example

To illustrate the principle of `IBB`, we take the 2D mechanical configuration example introduced in [2] (see Fig. 1). Various points (white circles) are connected with rigid rods (lines). Rods impose a distance constraint between two points. Point h (black circle) is attached to the rod $\langle g, i\rangle$. The distance from h to i is one third of the distance from g to i. Finally, point d is constrained to slide on the specified line. The problem is to find a feasible configuration of the points so that all constraints are satisfied. An equational decomposition method produces the `DAG` shown in Fig. 1-right. Points a, c and j constitute the input parameters (see Section 2).

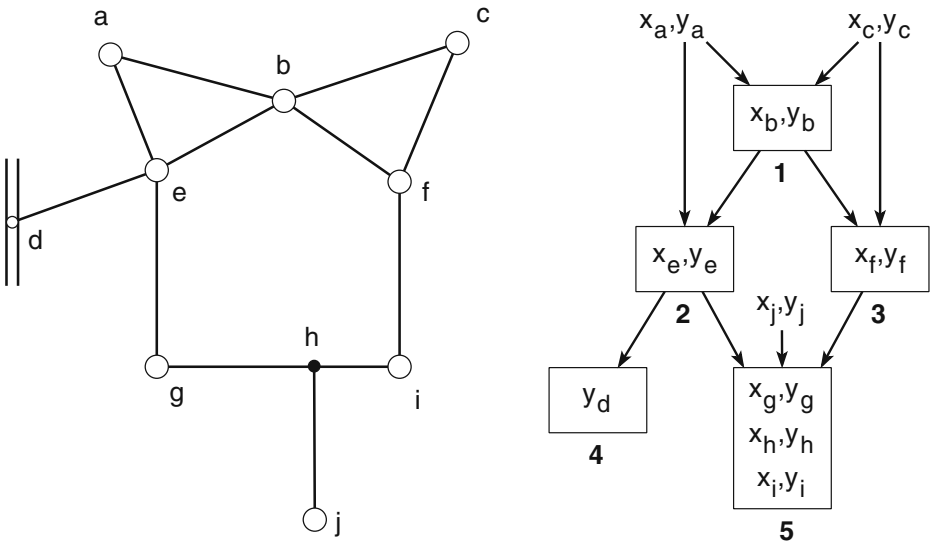


Fig. 1 Didactic problem and its DAG

3.2 Description of `IBB[BT]`

The algorithm `IBB[BT]` is a simple version of `IBB` based on a chronological backtracking (BT). It uses several arrays:

- `solutions[i,j]` is the j^{th} solution of block i.
- `#sols[i]` is the number of solutions in block i.
- `solIndex[i]` is the index of the current solution in block i (between 0 and `#sols[i]` -1).
- `assignment[v]` is the current value assigned to variable v.

Respecting the order of the DAG, `IBB[BT]` follows one of the induced total orders, yielded by the list `blocks`. The blocks are solved one by one. The procedure `BlockSolve` computes the solutions of `blocks[i]`. It stores them in `solutions` and computes `#sols[i]`, the number of solutions in block i. The found solutions are assigned to block i in a combinatorial way. (The procedure `assignBlock` instantiates the variables in the block: it updates `assignment` with the values given by `solutions [i, solIndex[i] ]`.) The process proceeds recursively to the next block $i+1$ until a solution for the last block is found: the values in `assignment` are then stored by the procedure `storeTotalSolution`. Of course, when a block has no (more) solution, we have to backtrack, i.e., the next solution of block $i-1$ is chosen, if any.

```
Algorithm IBB_c[BT] (blocks: a list of blocks, #blocks: the number of blocks)
    i ← 1
    recompute ← true
    while i ≥ 1 do
        if recompute then
            BlockSolve (blocks, i, solutions, #sols)
            solIndex[i] ← 0
        end
        if solIndex[i] ≥ #sols[i] /* all solutions of block i have been explored */ then
            i ← i − 1
            recompute ← false
        else
            /* solutions [i, solIndex[i] ] is assigned to block i */
            assignBlock (i, solIndex[i], solutions, assignment)
            solIndex[i] ← solIndex[i] +1
            if (i == #blocks) /* total solution found */ then
                storeTotalSolution (assignment)
            else
                i ← i + 1
                recompute ← true
            end
        end
    end
end.
```

The reader should notice a significant difference between `IBB[BT]` and the chronological backtracking schema used in finite-domain CSPs. The domains of variables in a CSP are static, whereas the set of solutions of a given block may change every time it is solved. Indeed, the system of equations itself may change from a call to another because the input variables, i.e., the parameters of the equation system, may change. This explains the use of the variable `recompute` set to *true* when the algorithm goes to a block downstream.

Let us emphasize this point on the didactic example. `IBB[BT]` follows one total order, e.g., block 1, then 2, 3, 4, and finally 5. Calling `BlockSolve` on block 1 yields two solutions for x_b. When one replaces x_b by one of its two values in the equations of subsequent blocks (2 and 3), these equations have a different coefficient x_b. Thus, in case of backtracking, block 2 must be solved twice, and with different equations, one for each value of x_b.

4 `IBB` with interval-based techniques

`IBB` can be used with any type of solver able to compute all the solutions of a system of equations (over the real numbers). In a long term, we intend to use `IBB` for solving systems of geometric constraints in CAD applications. In such applications, certain blocks will be solved by interval techniques while others, corresponding to theorems of geometry, will be solved by parametric hard-coded procedures obtained (off-line) by symbolic computation. In this paper, we consider only interval-based solving techniques, and thus view `IBB` as an interval-based algorithm for solving decomposed systems of equations.

4.1 Background in interval-based techniques

We present here a brief introduction of the most common interval-based operators used to solve a system of equations. The underlying principles have been developed in interval analysis and in constraint programming communities.

The whole system of equations, as well as the different blocks in the decomposition, are viewed as numerical CSPs.

Definition 1 A numerical CSP $P = (X, C, B)$ 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 $\mathbf{x_i}$ ($B = \mathbf{x_1} \times ... \times \mathbf{x_n}$). A solution of P is an assignment of the variables in V such that all the constraints in C are satisfied.

The n-set of intervals B is represented by an n-dimensional parallelepiped called a **box**. Since real numbers cannot be represented in computer architectures, the bounds of an interval $\mathbf{x_i}$ should actually be defined as floating-point numbers. A solving process reduces the initial box until a very small box is obtained. Such a box is called an **atomic box** in this paper. In theory, an interval could be composed by two consecutive floats in the end. In practice, the process is interrupted when all the intervals have a width less than `w_biss`, where `w_biss` is a user-defined parameter. It is worthwhile noting that an atomic box does not necessarily contain a solution. Indeed, evaluating an equation with interval arithmetic may prove that the equation has no solution (when the image of the corresponding box does not contain 0), but cannot assert that there exists a solution in the box. However, several operators from interval analysis can often certify that there exists a solution inside an atomic box.

Our interval-based solver `Ibex` uses interval-based operators to handle the blocks (`BlockSolve`). In the most sophisticated variant of `IBB`, the following three steps are iteratively performed. The process stops when an atomic box of size less than `w_biss` is obtained.

1. *Bisection:* One variable is chosen and its domain is split into two intervals (the box is split along one of its dimensions). This yields two smaller sub-CSPs which are handled in sequence. This makes the solving process combinatorial.
2. *Filtering/propagation:* Local information is used on constraints handled individually to reduce the current box. If the current box becomes empty, the corresponding branch (with no solution) in the search tree is cut [17, 19, 25].
3. *Interval analysis/unicity test:* Such operators use the first and/or second derivatives of the functions. They produce a "global" filtering on the current box. If additional conditions are fulfilled, they may ensure that a unique solution exists inside the box, thus avoiding further bisection steps.

4.2 Filtering/propagation

Propagation is performed by an AC3-like fixed-point algorithm. Several types of filtering operators reduce the bounds of intervals (no gap is created in the current box). The `2B-consistency` (also known as Hull-consistency—HC) and the `Box-consistency` [25] algorithms both consider one constraint at a time (like AC3) and reduce the bounds of the involved variables. `Box-consistency` uses an iterative process to reduce the bounds while `2B-consistency` uses projection

functions. The more expensive job performed by `Box-consistency` may pay when the equations contain several occurrences of a same variable. This is not the case with our benchmarks which are mostly made of equations modeling distances between 2D or 3D points, and of other geometric constraints. Hence, `Box-consistency` has been discarded. The `3B-consistency` [19] algorithm uses `2B-consistency` as a sub-routine and a refutation principle (shaving; similar to the Singleton Arc Consistency [6] in finite domains CSPs) to reduce the bounds of every variable iteratively. On the tested benchmarks our experiments have led us to use the `2B-consistency` operator (and sometimes `3B-consistency`) combined with an interval Newton.

4.3 Interval analysis

We have implemented in our library the interval Newton (`I-Newton`) operator [22]. `I-Newton` is an iterative numerical process, based on the first derivatives of equations, and extended to intervals. Without detailing this algorithm, it is worth understanding the output of `I-Newton`. Applied to a box B_0, `I-Newton` provides three possible answers:

1. When the Jacobian matrix is not strongly regular, the process is immediatly interrupted and B_0 is not reduced [22]. This necessarily occurs when B_0 contains several solutions. Otherwise, different iterations modify the current box B_i to B_{i+1}.
2. When B_{i+1} exceeds B_i in at least one dimension, B_{i+1} is intersected with B_i before the next iteration. No existence or unicity property can be guaranteed.
3. When the box B_{i+1} is included in B_i, then B_{i+1} is guaranteed to contain a unique solution (existence and unicity test).

In the last case, when a unique solution has been detected, the convergence onto an atomic box of width `w_biss` in the subsequent iterations is very fast, i.e., quadratic. Moreover, the width of the obtained atomic box is often very small (even less than `w_biss`), which highlights the significant reduction obtained in the last iteration (see Table 3).

4.4 Interval techniques and block solving

Let us stress a characteristic of the systems corresponding to blocks when they are solved by interval-based techniques: the equations contain coefficients that are not punctual but (small) intervals. Indeed, the solutions obtained in a given block are atomic boxes and become parameters of subsequent blocks. For example, the two possible values for x_b in block 1 are replaced by atomic boxes in block 2. This characterictic has several consequences.

The precision sometimes decreases as long as blocks are solved in sequence. A simple example is the a 1×1 block $x^2 = p$ where the parameter p is $[0, 10^{-10}]$. Due to interval arithmetics, solving the block yields a coarser interval $[-10^{-5}, 10^{-5}]$ for x. Of course, these pathological cases related to the proximity to 0, occur occasionally and, as discussed above, interval analysis renders the problem more seldom by sometimes producing tiny atomic boxes.

The second consequence is that it has no sense to talk about a unique solution when the parameters are not punctual and can thus take an infinite set of possible real values. Fortunately, the unicity test of `I-Newton` still holds. Generalizing the unicity test to non punctual parameters has the following meaning: if one takes *any* punctual real value in the interval of every parameter, it is ensured that exactly one point inside the atomic box found is a solution. Of course, this point changes according to the chosen punctual values. Although this proposition has not been published (to our knowledge), this is straightforward to extend the "punctual" proof to systems in which the parameters are intervals.

Remark In the old version using the `IlcInterval` library of Ilog Solver [23], the unicity test was closely associated to the `Box-consistency`. We now know that the benefit of mixing `Box` and `2B` was not due to the `Box-consistency` itself, but to the unicity test that avoided bisection steps in the bottom of the search tree. It was also due to the use of the *centered form* of the equations that produced additional pruning.

4.5 Inter-block filtering `(IBF)`

In all the variants of `IBB`, it is possible to add an *inter-block filtering* (*IBF*) process: instead of limiting the filtering process (e.g., `2B`) to the current block i, we apply the filtering process to the entire system.

In Section 6.2, we will detail a more incremental version of *IBF*, called *IBF+*, in which the filtering process is applied to only certain blocks, called *friend blocks* (those that can be filtered).

5 Different versions of `IBB`

Since 1998, several variants of `IBB` have been implemented [2, 23]. We can classify them into three main categories from the simplest one to the most sophisticated one. They differ in the way they manage, during the search for solutions, the current block (with procedure `BlockSolve`) and the other blocks of the system.

The third version $\texttt{IBB}_c$ is the most sophisticated one. It corresponds to the pseudo-code described in Section 3.2. $\texttt{IBB}_c$ defines one system per block in the decomposition. Data structures are used to manage the inter-block backtracking: for storing and restoring solutions of blocks, domains of variables and so on.

The first two versions are sufficiently simple to be directly integrated into `Ibex`.

$\texttt{IBB}_a$ can be viewed as a new splitting heuristic in a classical interval-based solving algorithm handling the whole system.

- $\texttt{IBB}_a$ handles the entire system of equations as a numerical CSP. Most of the operations, such as `2B` or `I-Newton`, are thus applied to the whole system so that $\texttt{IBB}_a$ necessarily calls inter-block filtering.
- The decomposition (i.e., the DAG of blocks) is just used to feed the new splitting heuristic. $\texttt{IBB}_a$ can choose the next variable to be split only inside the current block i. The specific variable inside the block i is chosen with a standard *round robin* strategy.

The second version IBB$_b$ is a bit more complicated and manages two systems at a time: the whole system, like for IBB$_a$, but also the current block which is managed as an actual system of equations by Ibex. Like for IBB$_a$, bisections are done in the whole system (selecting one variable in the current block). Also, it is possible to run a filtering process on the whole system, implementing a simple version of inter-block filtering.

Contrarily to IBB$_a$:

- It is possible to de-activate interblock-filtering (in the whole system).
- It is possible to run 2B and/or I-Newton in the current block only.

When 2B is run both in the current block and in the entire system (IBF), the two filtering processes are managed as follows. First, 2B is run on the current block until a fixed-point is reached. Second, 2B is run on the entire system.

The experiments will show that IBB$_a$ is not competitive with the other two versions because IBB$_a$ cannot run I-Newton in the current block. They also will show that IBB$_c$ is more robust than IBB$_b$ and can incorporate the sophisticated features described below.

6 Advanced features in IBB$_c$

The following sections mention or detail how are implemented advanced features in IBB$_c$ (called IBB for simplicity): intelligent backtracking, the *recompute condition* which is a simple way to exploit the partial order between blocks provided by the decomposition, a sophisticated variant of inter-block filtering. We also detail the advantages of using interval-Newton inside blocks.

6.1 Exploiting the DAG of blocks

As shown in Section 3.2, IBB[BT] uses only the total order between blocks and forgets the actual dependencies between them. However, IBB$_c$ is flexible enough to exploit the partial order between blocks. Figure 1-right shows an example. Suppose block 5 had no solution. Chronological backtracking would go back to block 4, find a different solution for it, and solve block 5 again. Clearly, the same failure will be encountered again in block 5.

It is explained in [2] that the *Conflict-based Backjumping* and *Dynamic backtracking* schemes cannot be used to take into account the structure given by the DAG. Therefore, an intelligent backtracking, called IBB[GPB], was introduced, based on the *partial order backtracking* [2, 20]. In 2003, we have also proposed a simpler variant IBB[GBJ] [14] based on the *Graph-based BackJumping* (GBJ) proposed by Dechter [7].

However, there is an even simpler way to exploit the partial order yielded by the DAG of blocks: the **recompute condition**. This condition states that it is useless to recompute the solutions of a block with BlockSolve if the parent variables have not changed. In that case, IBB can reuse the solutions computed the last time the block has been handled. In other words, when handling the next block $i+1$, the variable recompute is not always set to *true* (see Section 3.2). This condition has

been implemented in IBB[GBJ] and in IBB[BT]. In the latter case, the variant is named IBB[BT+].

Let us illustrate how IBB[BT+] works on the didactic example. Suppose that the first solution of block 3 has been selected, and that the solving of block 4 has led to no solution. IBB[BT+] then backtracks on block 3 and the second position of point f is selected. When IBB[BT+] goes down again to block 4, that block should normally be recomputed from scratch due to the modification of f. But x_f and y_f are not implied in equations of block 4, so that the two solutions of block 4, which had been previously computed, can be reused. It is easy to avoid this useless computation by using the DAG: when IBB goes down to block 4, it checks that the parent variables x_e and y_e have not changed.

Remark Contrarily to IBB_a and IBB_b, the recompute condition can be incorporated into IBB_c thanks to the management of sophisticated data structures.

6.2 Sophisticated implementation of inter-block filtering (IBF+)

IBF is integrated into IBB_b and IBB_c in the following way. When a bisection is applied to a variable in a given block i, the filtering operators described above, i.e., 2B and I-Newton, are first called inside the block. Second, *IBF* is launched on the entire system.

In the latest versions of IBB_b and IBB_c, *IBF* is launched in a more incremental way. The underlying local filtering (e.g., 2B) is run with a propagation queue initially filled with only the variables inside the current block, which lowers the overhead related to *IBF* when it is not efficient.

To perform a more sophisticated implementation of *IBF*, called *IBF+*, before solving a block i, one forms a subsystem extracted from the *friend blocks* F'_i of block i. The filtering process will concern only the friend blocks, thus avoiding the management of the other ones. The friend blocks of i are extracted as follows:

1. take the set $F_i = \{i...\#blocks\}$ containing the blocks not yet "instantiated",
2. keep in F'_i only the blocks in F_i that are connected to i in the DAG.[1]

To illustrate *IBF+*, let us consider the DAG of the didactic example. When block 1 is solved, all the other blocks are considered by *IBF+* since they are all connected to block 1. Any interval reduction in block 1 can thus possibly perform a reduction for any variable of the system. When block 2 is solved, a reduction has potentially an influence on blocks 3, 4, 5 for the same reasons. (Notice that block 3 is a friend block of block 2 that is not downstream to block 2 in the DAG.) When block 3 is solved, a reduction can have an influence only on block 5. Indeed, once blocks 1 and 2 have been removed (because they are "instantiated"), block 3 and 4 do not belong anymore to the same connected component. Hence, no propagation can reach block 4 since the parent variables of block 5, which belong to block 2, have an interval of width at most w_biss and thus cannot be reduced further.

[1]The orientation of the DAG is forgotten at this step, that is, the arcs of the DAG are transformed into non-directed edges, so that the filtering can also be applied on friend blocks that are not directly linked to block i.

IBF+ implements only a local filtering on the friend blocks, e.g., `2B-consistency` on the tested benchmarks. It turns out that `I-Newton` is counterproductive in *IBF*. First, it is expensive to compute the Jacobian matrix of the whole system. More significantly, it is likely that `I-Newton` does not prune at all the search space (except when handling the last block) because it always falls in the singular case. As a rule of thumb, if the domain of one variable x in the last block contains two solutions, then the whole system will contain at least two solutions until x is bisected. This prevents `I-Newton` from pruning the search space. This explains why $\texttt{IBB}_a$ is not competitive with the two other versions of `IBB`.

The experiments confirm that it is always fruitful to perform a sophisticated filtering process inside blocks (i.e., `2B + I-Newton`), whereas *IBF* or *IBF+* (on the entire system) produces sometimes, but not always, additional gains in performance.

6.3 Mixing *IBF* and the recompute condition

Incorporating *IBF* or *IBF+* into $\texttt{IBB}_c\texttt{[BT]}$ is straightforward. This is not the case for the variants of `IBB` with more complicated backtracking schemes. Reference [14] gives guidelines for the integration of *IBF+* into `IBB[GBJ]`. More generally, *IBF+* adds in a sense some edges between blocks. It renders the system less sparse and complexifies the recomputation condition. Indeed, when *IBF+* is launched, the parent blocks of a given block i are not the only exterior causes of interval reductions inside i. The friend blocks of i have an influence as well and must be taken into account.

For this reason, when *IBF+* is performed, the recompute condition is more often true. Since the causes of interval reductions are more numerous, it is more seldom the case that all of them have not changed. This will explain for instance why the gain in performance of `IBB[BT+]` relatively to `IBB[BT]` is more significant than the gain of $\texttt{IBB}_c\texttt{[BT+,IBF+]}$ relatively to $\texttt{IBB}_c\texttt{[BT,IBF+]}$ (see experiments).

This remark holds even more for the simple *IBF* implementation where local filtering is run on the entire system (and not only on friend blocks). In this case, the recompute condition is simply always true, so that `BT+` becomes completely inefficient and avoids the recomputation of zero block. In other terms, $\texttt{IBB}_c\texttt{[BT,IBF]}$ and $\texttt{IBB}_c\texttt{[BT+,IBF]}$ are quasi-identical.

6.4 Discarding the non reliable midpoint heuristic

The integration of the Ibex solver underlies several improvements of `IBB`. As previously mentioned, using a white box allows us to better understand what happens. Also, $\texttt{IBB}_c$ is now reliable. The *parasitic solutions* problem has been safely handled (see Section 6.5) and an ancient **midpoint heuristic** is now abandoned. This heuristic replaced every parameter, i.e., input variable, of a block by the midpoint of its interval. Such a heuristic was necessary because the `IlcInterval` library previously used did not allow the use of interval coefficients. Ibex accepts non punctual coefficients so that no solution is lost anymore, thus making `IBB` reliable. The midpoint heuristics would however allow the management of sharper boxes, but the gain in running time would be less than 5% in our benchmarks. The price of reliability is not so high!

6.5 Handling the problem of *parasitic solutions*

With interval solving, *parasitic solutions* are obtained when:

- several atomic boxes are returned by the solver as possible solutions;
- these boxes are close one to each other;
- only one of them contains an actual solution and the others are not discarded by filtering.

Even when the number of parasitic solutions is small, `IBB` explodes because of the multiplicative effect of the blocks, i.e., because the parasitic partial solutions are combined together. In order that this problem occurs more rarely, one can reduce the precision (i.e., enlarge `w_biss`) or mix several filtering techniques together. The use of interval analysis operators like `I-Newton` is also a right way to fix most of the pathological cases (see experiments).

It appears that $\texttt{IBB}_a$ and $\texttt{IBB}_b$ are not robust against the parasitic solutions problem that often occurs in practice. $\texttt{IBB}_c$ handles this problem by taking the union of the close boxes (i.e., the hull of the boxes). $\texttt{IBB}_c$ considers the different blocks separately, so that all the solutions of a block can be computed before solving the next one. This allows $\texttt{IBB}_c$ to merge close atomic boxes together. Note that the previous implementations of $\texttt{IBB}_c$ might lose some solutions because no hull between close boxes was performed. Instead, only one of the close boxes was selected and might lead to a failure in the end when the selected atomic box did not contain a solution.

6.6 Certification of solutions

As mentioned in Section 4.1, certifying the existence and the unicity of a solution inside an atomic box returned by the solver requires interval analysis techniques. With `IBB`, we use an interval Newton to contract every block and to guarantee the solutions that are inside. `IBB` can thus often certify solutions of a decomposed system. Indeed, a straightforward induction ensures that *a total solution is certified iff all the corresponding partial solutions are certified in every block.*

Among the ten benchmarks studied below, only solutions of `Mechanism` and `Chair` have not been certified.

6.7 Summary: benefit of running `I-Newton` inside blocks

Finally, as shown in the experiments reported below, the most significant impact on `IBB` is due to the integration of an interval Newton inside the blocks:

- `I-Newton` has a good power of filtering, thus reducing time complexity.
- Due to its quadratic convergence, `I-Newton` often allows us to reach the finest precision, i.e. an even better precision than `w_biss`. This is of great interest because the solutions of a given block become coefficients (input parameters) in blocks that are downstream in the DAG. Thus, when no `I-Newton` is used, the precision may decrease during block solving, generally obtaining at the end a precision which is worse than `w_biss`.
  Second, with thinner input parameters, the loss in performance of `IBB`, as compared to the use of the discarded (non reliable) midpoint heuristic, is negligible.
- `I-Newton` often allows us to certify the solutions.

- Hence, the combinatorial explosion due to parasitic solutions is drastically limited.

Moreover, the use of `I-Newton` alters the comparison between variants of `IBB`. In particular, in the previous versions, we concluded that *IBF* was counterproductive, whereas it is not always true today. Also, the interest of intelligent backtracking algorithms is clearly put into question, which confirms the intuition shared by the constraint programming community that a better filtering (due to `I-Newton`) removes backtracks (and backjumps). Moreover, since `I-Newton` has a good filtering power, obtaining an atomic box requires less bisections. Hence, the number of calls to *IBF* is reduced in the same proportion.

7 Experiments

We have applied several variants of `IBB` on the benchmarks described above.

7.1 Benchmarks

Exhaustive experiments have been performed on 10 benchmarks made of geometric constraints. They compare different variants of `IBB` and show a clear improvement w.r.t. solving the whole system.

Some benchmarks are artificial problems, mostly made of quadratic distance constraints (Figs. 2 and 3). `Mechanism` and `Tangent` have been found in [16] and [3]. `Chair` is a realistic assembly made of 178 equations induced by a large variety of geometric constraints: distances, angles, incidences, parallelisms, orthogonalities [14].

The `DAGs` of blocks for the benchmarks have been obtained either with an equational method (abbrev. equ. in Table 1) or with a geometric one (abbrev. geo.). `Ponts` and `Tangent` have been decomposed by both techniques.

A problem defined with a domain of width 100 (see column 6 of Table 1) is generally similar to assigning $(-\infty, +\infty)$ to every domain. The intervals in `Mechanism` and `Sierp3` have been selected around a given solution in order to limit the total number of solutions. In particular, the equation system corresponding to `Sierp3` would have about 2^{40} solutions, so that the initial domains are limited to a width 1. `Sierp3` is the *Sierpinski* fractal at level 3, that is, 3 Sierpinski at level 2 (i.e., `Ponts`)

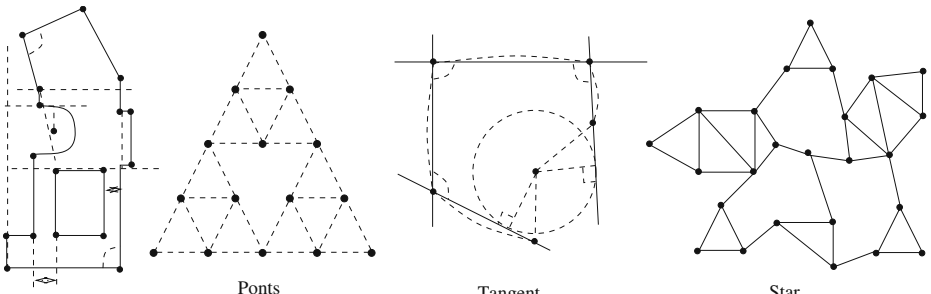


Fig. 2 2D benchmarks: general view

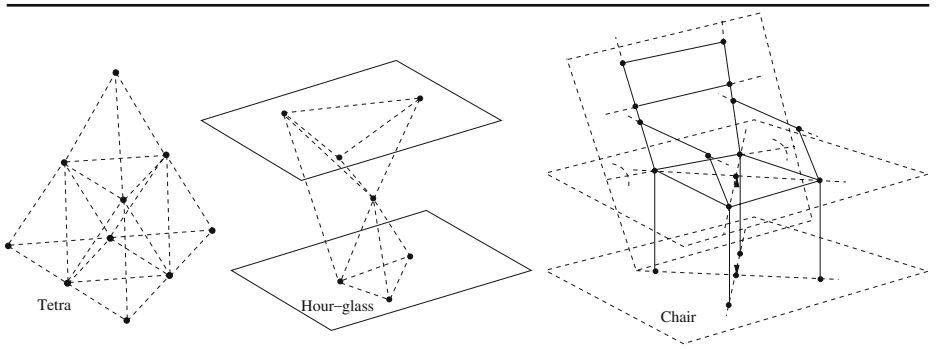


Fig. 3 3D benchmarks: general view

put together. The time spent for the equational and geometric decompositions is always negligible, i.e., a few milliseconds for all the benchmarks.

7.2 Brief introduction to `Ibex`

All the tests with `IBB` have been conducted using the interval-based library, called `Ibex`, implemented in `C++` by the third author [4, 5]. The hull consistency (i.e., `2B-consistency`) is implemented with the famous `HC4` that builds a syntactic tree for every constraint [1, 17]. A "width" parameter `r_hc4` must be tuned: a constraint is pushed in the propagation queue if the projection on one of its variables has reduced the corresponding intervals more than `r_hc4` (ratio of interval width). `I-Newton` is run when the largest interval in the current box has a width less than `ceiling_newton`. For using 3B-consistency [19], one must specify the width of the smallest interval that the algorithm tries to refute. This parameter `ratio_var_shave` (in short `r_vs`) is given as a ratio of interval width. Two atomic boxes are merged iff a unique solution has not been certified inside both and the boxes are sufficiently close to each other, that is, for every variable, there is a distance

Table 1 Details about the benchmarks

GCSP	Dim.	Dec.	Size	Size of blocks	Dom.	#sols	`w_biss`
`Mechanism`	2D	equ.	98	$98 = 1 \times 10, 2 \times 4, 27 \times 2, 26 \times 1$	10	448	5.10^{-6}
`Sierp3`		geo.	124	$124 = 44 \times 2, 36 \times 1$	1	198	10^{-8}
`PontsE`		equ.	30	$30 = 1 \times 14, 6 \times 2, 4 \times 1$	100	128	10^{-8}
`PontsG`		geo.	38	$38 = 13 \times 2, 12 \times 1$	100	128	10^{-8}
`TangentE`		equ.	28	$28 = 1 \times 4, 10 \times 2, 4 \times 1$	100	128	10^{-8}
`TangentG`		geo.	42	$42 = 2 \times 4, 11 \times 2, 12 \times 1$	100	128	10^{-8}
`Star`		equ.	46	$46 = 3 \times 6, 3 \times 4, 8 \times 2$	100	128	10^{-8}
`Chair`	3D	equ.	178	$178 = 1 \times 15, 1 \times 13, 1 \times 9, 5 \times 8, 3 \times 6, 2 \times 4, 14 \times 3, 1 \times 2, 31 \times 1$	100	8	5.10^{-7}
`Tetra`		equ.	30	$30 = 1 \times 9, 4 \times 3, 1 \times 2, 7 \times 1$	100	256	10^{-8}
`Hourglass`		equ.	29	$29 = 1 \times 10, 1 \times 4, 1 \times 3, 10 \times 1$	100	8	10^{-8}

Type of decomposition method (Dec.); number of equations (Size); Size of blocks: NxK means N blocks of size K; Interval widths of variables (Dom.); number of solutions (#sols); bisection precision, i.e., domain width under which bisection does not split intervals (`w_biss`)

dist less than 10^{-2} between the two boxes (10^{-4} for the benchmark Star). Most of the reported CPU times have been obtained on a Pentium IV 3 Ghz.

7.3 Interest of system decomposition

The first results show the dramatic improvement due to IBB as compared to four interval-based solvers. All the solvers use a round-robin splitting strategy. Ilog Solver [11] uses a filtering process mixing 2B-consistency, Box-consistency and an interval analysis operator for certifying solutions. The relatively bad CPU times simply show that the IlcInterval library has not been improved for several years. They have been obtained on a Pentium IV 2.2 Ghz.

On Table 2, Ibex uses a 3B+Newton filtering algorithm. RealPaver [8, 9] generally uses HC4+Newton filtering (better results are obtained by weak3B + Newton only for TangentE; the *weak 3B-consistency* is described in [8]) The 3B operator of Ibex behaves better than that of RealPaver on these benchmarks essentially because it manages a parameter r_vs which is a ratio of interval and not a fixed width. Note that RealPaver uses the default values of the different parameters. We have also applied the Quad operator [18] that is sometimes very efficient to solve polynomial equations. This operator appears to be very slow on the tested benchmarks.

7.3.1 Tuning parameters

We would like to stress that it is even easier to tune the parameters used by IBB with Ibex (i.e., r_hc4 and ceiling_newton) than the parameters used by Ibex applied to the entire system (i.e., r_hc4, ceiling_newton and ratio_var_shave). First, there is one less parameter to be tuned with IBB. Indeed, as shown in Section 7.7, this is counterproductive (w.r.t. CPU time) to use 3B-consistency with IBB. Second, the value of ceiling_newton can have a significant impact on CPU

Table 2 Interest of IBB

GCSP	Ibex	RealPaver	Ilog Solver	IBB_c [BT+]	IBB_b [BT, IBF]	$\frac{Ibex}{IBB}$	Precision
Mechanism	> 4000 (117)	XXX	> 4000	1.55	1.65	75	2.10^{-5}
Sierp3	36	> 4000	> 4000	4	1.08	33	4.10^{-11}
Chair	> 4000	XXX	> 128 eq.	0.36	1.36	$> 10^4$	10^{-7}
Tetra	18.2	42	> 4000	0.96	1.42	19	2.10^{-14}
PontsE	7.2	6.9	103	0.97	1.21	7	7.10^{-14}
PontsG	3.2	1.9	294	1.52	0.43	8	10^{-13}
Hourglass	0.25	0.32	247	0.019	0.029	13	10^{-13}
TangentE	9.6	22*	191	0.15	0.15	64	5.10^{-14}
TangentG	14.6	XXX	XXX	0.10	0.16	146	4.10^{-14}
Star	18.8	12	1451	0.18	0.15	125	2.10^{-11}

The columns in the left report the times (in seconds) spent by three interval-based solvers to handle the systems globally. The column IBB_c [BT+] and IBB_b [BT, IBF] report the CPU times obtained by two interesting variants of IBB. Gains of at least one order of magnitude are highlighted by the column *Ibex*/IBB. The last column reports the size of the obtained atomic boxes. The obtained precision is often better than the specified parameter w_biss (see Table 1) thanks to the use of I-Newton. An entry XXX means that the solver is not able to isolate solutions (see Section 6.5)

Table 3 Variants of IBB with HC4+Newton filtering inside blocks

GCSP	1	2	3	4	5	6	7
	IBB_b [BT]	IBB_c [BT]	IBB_a [BT, IBF]	IBB_b [BT, IBF]	IBB_c [BT, IBF+]	IBB_c [BT+]	IBB_c [BT+, IBF+]
Mechanism	146	160	11000	165	196	155	195
	22803	22803	35007	22111	22111	22680	22066
	1635	1635	–	1629	1629	1544	1156
Sierp3	317	628	29660	108	307	402	258
	35685	35685	2465	5489	5484	19454	4354
	21045	21045	–	4272	4272	12242	3455
Chair	94	97	1080	136	163	36	127
	7661	7661	6601	6825	6825	2368	4304
	344	344	–	344	344	97	148
Tetra	108	109	4370	142	142	96	127
	10521	10521	66933	9843	9843	9146	8568
	235	235	–	235	235	100	100
PontsE	99	100	706	121	123	97	118
	6103	6103	23389	5880	5880	5986	5776
	131	131	–	115	115	79	67

PontsG	147	233	224	43	71	152	71
	14253	14253	4505	2669	2669	7154	2669
	9283	9283	–	1155	1155	6585	1155
Hourglass	2.1	2.7	18	2.9	4.2	1.9	4.2
	59	59	341	59	59	48	59
	57	57	–	53	53	35	53
TangentE	11	15	5370	15	21	15	21
	313	313	40747	308	308	304	308
	427	427	–	427	427	423	427
TangentG	12	16	XXX	16	24	10	24
	817	817	–	817	816	102	817
	411	411	–	411	411	238	396
Star	31	35	2400	15	20	18	6
	2475	2475	5243	1347	1347	1534	192
	457	457	–	254	254	325	34

Every entry contains three values: (top) the CPU time for obtaining all the solutions (in hundredths of second); (middle) the total number of bisections performed by the interval solver; (bottom) the total number of times `BlockSolve` is called

time with a global solving while it is not the case with IBB. Overall, a user of IBB must finely tune only the parameter r_hc4 used by HC4.

Table 3 reports the main results we have obtained with several variants of IBB. Tables 4, 5 and 6 highlight specific points.

7.4 Poor results obtained by IBB_a

Table 3 clearly shows that IBB_a is not competitive with IBB_b and IBB_c. As shown in Table 2, the results in CPU time obtained by IBB_a are close to or better than those obtained by Ibex applied to the entire system (with 3B, I-Newton and a round-robin splitting strategy): Tables 2 and 6 underline that IBB_a is able to render Mechanism and Chair tractable.

Table 6 also reports the results obtained by IBB_a with 3B (i.e., *IBF* is performed by 3B). Recall that IBB_a can be viewed as a splitting strategy driven by the decomposition into blocks (i.e., a total order between blocks). Thus, these results mainly underline that the IBB_a splitting heuristic is better than round-robin. We see below that a relevant filtering *inside* blocks, performed in IBB_b and IBB_c, bring an even better performance.

7.5 IBB_b versus IBB_c

IBB_a is significantly less efficient than IBB_b and IBB_c because it does not use I-Newton inside blocks.

The comparison between IBB_b and IBB_c can be summed up in several points mainly deduced from Tables 3 and 5:

- All the variants of IBB_b and IBB_c obtain similar results on all the benchmarks (provided that HC4+Newton filtering is used inside blocks).
- IBB_b is a simpler implementation of IBB than IBB_c. The main reason is that no sophisticated data structures are used by IBB_b. This explains that the CPU times obtained by IBB_b[BT] are better than those obtained by IBB_c[BT] (see columns 1 and 2 in Table 3). Also, IBB_b[BT,IBF] is more efficient than IBB_b[BT,IBF] (compare column 4 of Table 3 and column 2 of Table 5).

Table 4 No interest of intelligent backtracking

GCSP	IBB_c [BT]	IBB_c [BT+]	IBB_c [GBJ]	IBB_b [BT,IBF]	IBBc [BT,IBF+]	IBB_c [BT+,IBF+]	IBB_c [GBJ,IBF+]
Sierp3	628	402	288	108	307	258	252
	35684	19454	13062	5489	5484	4354	4260
	21045	12242	8103	4272	4272	3455	3175
			BJ=2974				BJ=135
Star	35	18	17	15	20	6	6
	2474	1534	1500	1347	1346	192	192
	457	325	277	254	254	34	34
			BJ=6				BJ=0

The number of backjumps is drastically reduced by the use of IBF (6 → 0 on Star; 2974 → 135 on Sierp3). The times obtained with IBF are better than or equal to those obtained with intelligent backtracking schemes. Only a marginal gain is obtained by IBB_c[GBJ,IBF+] w.r.t. IBB_c[BT+,IBF+] for Sierp3

Table 5 Comparison between *IBF* and *IBF+* in $\texttt{IBB}_c$

GCSP	$\texttt{IBB}_c$ [BT+]	$\texttt{IBB}_c$ [BT(+),IBF]	$\texttt{IBB}_c$ [BT,IBF+]	$\texttt{IBB}_c$ [BT+,IBF+]
Mechanism	155	199	196	195
	22680	22111	22111	22066
	1544	1629	1629	1156
Sierp3	402	828	307	258
	19454	5484	5484	4354
	12242	4272	4272	3455
Chair	36	226	163	127
	2368	6840	6825	4304
	97	344	344	148
Tetra	96	149	142	127
	9146	9842	9843	8568
	100	235	235	100
PontsE	97	126	123	118
	5986	5880	5880	5776
	79	115	115	67
PontsG	152	126	71	71
	7154	2669	2669	2669
	6585	1155	1155	1155
Hourglass	1.9	5.5	4.2	4.2
	48	59	59	59
	35	53	53	53
TangentE	15	32	21	21
	304	308	308	308
	423	427	427	427
TangentG	10	39	24	24
	102	816	816	816
	238	411	411	396
Star	18	32	20	6
	1534	1347	1347	192
	325	254	254	34

Thus, a same IBB algorithm is better implemented by the $\texttt{IBB}_b$ scheme.

- Using the BT+ backtracking scheme (related to the recompute solution) is always better than using the standard BT. The overhead is negligible and it avoids solving some blocks (compare for instance the number of solved blocks in columns 2 and 6 of Table 3).
  This is a good argument in favor of the $\texttt{IBB}_c$ version.
- Tables 3 and 5 shows that the interest of *IBF* is not clear. However, it seems that *IBF* is useful for hard instances for which a lot of choice points lead to failure and backtracking. For instance, Chair has only 8 solutions and implies thrashing in the search tree.
  Table 4 reports the only two benchmarks for which backjumps actually occur with an intelligent backtracking. It shows that the gain obtained by an intelligent backtracking (IBB[GBJ])) is compensated by a gain in filtering with IBF.
  Note that *IBF* was clearly counterproductive in old versions of IBB that did not use I-Newton to filter inside blocks. Indeed, a smaller filtering power implied more bisections and thus a larger number of calls to *IBF*.

Overall, the four versions of IBB that are the most efficient are $\texttt{IBB}_b$[BT], $\texttt{IBB}_b$[BT,IBF], $\texttt{IBB}_c$[BT+], $\texttt{IBB}_c$[BT+,IBF+] (see Tables 2 and 5).

Table 6 Use of 3B

GCSP	Ibex-2B	Ibex-3B	IBB$_a$[BT,IBF-2B]	IBB$_a$[BT,IBF-3B]	IBB$_c$[BT+]-2B	IBB$_c$[BT+]-3B	IBB$_c$[BT+]-3B>8
Mechanism	>400000	>400000	11000	17300	155	389	156
	–	–	22680	35007	2435	4236	23452
	–	–	–	–	1544	1544	1544
Sierp3	>400000	3580	29660	3850	402	1028	–
	–	453	2465	551	19454	12138	–
	–	–	–	–	12242	12242	–
Chair	>400000	>400000	1080	4040	36	80	35
	–	–	6601	329	2368	144	1458
	–	–	–	–	97	97	97
Tetra	75900	1820	4370	1310	96	123	80
	3409073	1337	66933	1999	9146	350	570
	–	–	–	–	100	100	100
PontsE	1070	720	706	354	97	117	117
	34457	843	23389	383	5986	320	334
	–	–	–	–	79	79	79

PontsG	623	317	224	306	152	400	–
	26099	407	4505	561	7154	5662	–
	–	–	–	–	6585	6585	–
Hourglass	32	25	18	25	1.9	3.6	2.2
	285	27	341	35	48	20	46
	–	–	–	–	35	35	35
TangentE	17800	960	5370	260	15	39	–
	568189	2457	40747	325	304	254	–
	–	–	–	–	423	423	–
TangentG	XXX	1460	XXX	380	10	18	–
	–	543	–	389	102	38	–
	–	–	–	–	238	238	–
	5240	1880	2400	1950	18	42	–
Star	28445	615	5243	691	1534	1126	–
	–	–	–	–	325	324	–

The columns 2 and 3 report the results obtained by Ibex on the entire system with resp. 2B filtering and 3B filtering. Columns 4 and 5 report the results obtained by IBB_a with *IBF* performed resp. by 2B and 3B. For obtaining the last three columns, one investigated three different approaches for filtering inside blocks: with 2B, with 3B and with a mixed approach: 2B is used for blocks of size less than 9 and 3B is used for the largest blocks

Two other points must be considered for a fair comparison between these four versions of IBB.

Since $\texttt{IBB}_b$ [BT,IBF] uses an incremental *IBF* (i.e., pushing initially in the propagation queue only the variables of the current block), the overhead w.r.t. $\texttt{IBB}_b$ [BT] is small: it is less than 50% when *IBF* does not reduce anything. If you compare the numbers of bisections and solved blocks in columns 1 and 4 of Table 3, these numbers are close in the two columns for Mechanism, Chair, Tetra, PontsE, Hourglass, TangentE, TangentG, which indicates that *IBF* prunes nothing or only a few. However, the loss in performance of $\texttt{IBB}_b$ [BT, IBF] lies (only) between 15% and 50%. The gain for the three other instances is substantial.

This suggests than $\texttt{IBB}_b$ [BT,IBF] is more robust (w.r.t. the CPU time complexity) than $\texttt{IBB}_b$ [BT], making it more attractive.

The second point cannot be deduced from the tables because it is related to robustness.

Experiments with old versions of IBB without I-Newton inside blocks clearly showed the combinatorial explosion of $\texttt{IBB}_b$ involved by the parasitic solution problem. The use of I-Newton limits this problem, except for Mechanism. Instead of computing the 448 solutions, $\texttt{IBB}_b$ [BT] and $\texttt{IBB}_b$ [BT,IBF] compute 680 solutions because it is not endowed with the parasitic solutions merging. However, it is important to explain that the problem needed also to be fixed by manually selecting an adequate value for the parameter w_biss. In particular, $\texttt{IBB}_b$ undergoes a combinatorial explosion on Chair and Mechanism when the precision is higher (i.e., when w_biss is smaller). On the contrary, the $\texttt{IBB}_c$ version automatically adjusts w_biss in every block according to the width of the largest input variable (parameter) interval. $\texttt{IBB}_c$ is thus more robust than $\texttt{IBB}_b$ and can add sophisticated features such as solution merging, the recompute condition and a finer implementation of *IBF* (with friend blocks).

These observations lead us to recommend 2 (or 3) versions of IBB:

- $\texttt{IBB}_b$ [BT,IBF] which is simple to be implemented (available in Ibex) and is often efficient when I-Newton behaves well;
- the more sophisticated version $\texttt{IBB}_b$ [BT+] (generally without and sometimes with *IBF*+) that is very useful when the interval-based solver cannot isolate solutions.

The following sections detail some points leading to the above recommendation.

7.6 *IBF* versus *IBF*+

Table 5 clearly shows that the *IBF*+ implementation, that can only be used with $\texttt{IBB}_c$, leads to gains in performance w.r.t. the simple *IBF*.

One can also observe that *IBF*+ lowers the interest of BT+ w.r.t. to BT.

7.7 IBB and 3B

Table 6 yields some indications about the interest of 3B. The columns 2 and 3 suggest that 3B applied to the entire system seems fruitful on sparse systems. It is even generally better than $\texttt{IBB}_a$ (with 2B). This suggests that a strong filtering process (i.e., 3B) has about the same impact as a good splitting strategy (i.e., $\texttt{IBB}_a$).

The last three columns explain why we have chosen 2B and not 3B to filter inside blocks. The last column reports a new experiment in which 3B is used only on the largest blocks. Indeed, due to combinatorial considerations, we believe that 3B can seldom be efficient for handling small systems [24]. The first results are not very convincing, but experiments must be performed on more benchmarks.

Although not reported, we have also experimented $\texttt{IBB}_a$ and $\texttt{IBB}_b$ whose *IBF* is implemented with 3B. This variant is counterproductive, but seems to be more robust, that is, IBB can more easily isolate solutions (when I-Newton is not effective). For instance, it does not require merging close atomic boxes of Mechanism to find exactly 448 solutions.

8 Conclusion

In this article, we have proposed new versions of IBB that use the new interval-based library Ibex. Discarding the old midpoint heuristic has rendered IBB reliable.

The main impact on robustness and performance is due to the combination of local filtering (e.g., 2B) and interval analysis operators (e.g., interval Newton) inside blocks. Using 3B instead of 2B seems not promising except maybe for large blocks, as shown by first experiments.

Two other advanced features have shown their efficiency to limit choice points during search: inter-block filtering (*IBF*) and the recompute condition that avoids solving some blocks during search (BT+). We are now able to provide clear recommendations about these features.

- The best implementation of *IBF* (*IBF*+) is based on the computation of the subset of blocks that can actually be filtered when the current block is handled.
- It is not easy to know in advance which feature among *IBF*+ and BT+ has the greatest impact on time complexity. In addition, using *IBF*+ makes *BT*+ less effective.
- Inter-block backtracking schemes that are more sophisticated than BT+, such as GBJ and GPB, have not proven their efficiency, especially thanks to the use of *IBF* that removes most of the potential backjumps.

Thus, we recommend two versions of IBB. First, $\texttt{IBB}_b$ [BT,IBF] is a simple implementation directly available in Ibex [4, 5]. It is very simple, very fast (the overcost in CPU time related to the call to *IBF* is always less than 50% and sometimes pays off significantly). It works well when I-Newton inside blocks can isolate solutions. Second, $\texttt{IBB}_c$ [BT+] (or $\texttt{IBB}_b$ [BT,IBF+]) is a more sophisticated version that makes the approach more robust when I-Newton cannot certify or isolate solutions.

In addition to the dramatic gain in performance w.r.t. a global solving, IBB is simpler to be tuned. Indeed, only the parameter r_hc4 used by HC4 needs to be finely tuned.

Apart from minor improvements, IBB is now mature enough to be used in CAD applications. Promising research directions are the computation of sharper Jacobian matrices (because, in CAD, the constraints belong to a specific class) and the design of solving algorithms for equations with non punctual coefficients.

References

1. Benhamou, F., Goualard, F., Granvilliers, L., & Puget, J.-F. (1999). Revising hull and box consistency. In *ICLP* (pp. 230–244).
2. Bliek, C., Neveu, B., & Trombettoni, G. (1998). Using graph decomposition for solving continuous CSPs. In *Proc. CP'98, LNCS* (Vol. 1520, pp. 102–116).
3. Bouma, W., Fudos, I., Hoffmann, C. M., Cai, J., & Paige, R. (1995). Geometric constraint solver. *Computer Aided Design, 27*(6), 487–501.
4. Chabert, G. (2009). Ibex—An Interval based EXplorer. www.ibex-lib.org.
5. Chabert, G., & Jaulin, L. (2009). Contractor programming. *Artificial Intelligence*. Accessed 18 March 2009.
6. Debruyne, R., & Bessière, C. (1997). Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. of IJCAI* (pp. 412–417).
7. Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence, 41*(3), 273–312.
8. Granvilliers, L. (2003). `RealPaver` *user's manual, version 0.3.* University of Nantes. www.sciences.-univ-nantes.fr/info/perso/permanents/granvil/realpaver.
9. Granvilliers, L., & Benhamou, F. (2006). RealPaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software, 32*(1), 138–156.
10. Hoffmann, C., Lomonossov, A., & Sitharam, M. (1997). Finding solvable subsets of constraint graphs. In *Proc. constraint programming CP'97* (pp. 463–477).
11. ILOG, Av. Galliéni, Gentilly (2000). *Ilog solver V. 5, users' reference manual*.
12. Jaulin, L., Kieffer, M., Didrit, O., & Walter, E. (2001). *Applied interval analysis*. New York: Springer.
13. Jermann, C., Neveu, B., & Trombettoni, G. (2003). Algorithms for identifying rigid subsystems in geometric constraint systems. In *Proc. IJCAI* (pp. 233–38).
14. Jermann, C., Neveu, B., & Trombettoni, G. (2003). Inter-Block backtracking: Exploiting the structure in continuous CSPs. In *Proc. of 2nd int. workshop on global constrained optimization and constraint satisfaction (COCOS'03)*.
15. Jermann, C., Trombettoni, G., Neveu, B., & Mathis, P. (2006). Decomposition of geometric constraint systems: A survey. *International Journal of Computational Geometry and Applications (IJCGA), 16*(5–6), 379–414.
16. Latham, R. S., & Middleditch, A. E. (1996). Connectivity analysis: A tool for processing geometric constraints. *Computer Aided Design, 28*(11), 917–928.
17. Lebbah, Y. (1999). *Contribution à la résolution de contraintes par consistance forte*. Ph.D. thesis, Université de Nantes.
18. Lebbah, Y., Michel, C., Rueher, M., Daney, D., & Merlet, J. P. (2005) Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis, 42*(5), 2076–2097.
19. Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *IJCAI* (pp. 232–238).
20. McAllester, D. A. (1993). Partial order backtracking. Research note, artificial intelligence laboratory, MIT. ftp://ftp.ai.mit.edu/people/dam/dynamic.ps.
21. Merlet, J.-P. (2002). Optimal design for the micro parallel robot MIPS. In *Proc. of IEEE international conference on robotics and automation, ICRA '02, Washington DC, USA* (Vol. 2, pp. 1149–1154).
22. Neumaier, A. (1990). *Interval methods for systems of equations*. Cambridge: Cambridge University Press.
23. Neveu, B., Jermann, C., & Trombettoni, G. (2005). Inter-Block backtracking: Exploiting the structure in continuous CSPs. In *Selected papers in the 2nd int. worksh. on global constrained optimization and constraints, COCOS, LNCS* (Vol. 3478, pp. 15–30).
24. Trombettoni, G., & Chabert, G., (2007). Constructive interval disjunction. In *Proc. of CP* (pp. 635–650).
25. Van Hentenryck, P., Michel, L., & Deville Y. (1997). *Numerica: A modeling language for global optimization*. Cambridge: MIT.
26. Wilczkowiak, M., Trombettoni, G., Jermann, C., Sturm, P., & Boyer, E. (2003). Scene modeling based on constraint system decomposition techniques. In *Proc. ICCV*.