When Interval Analysis Helps Inter-Block Backtracking

Bertrand Neveu, Gilles Chabert, and Gilles Trombettoni

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

Abstract. Inter-block backtracking (IBB) computes all the solutions of sparse systems of non-linear equations over the reals. This algorithm, introduced in 1998 by Bliek et al., 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 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 equation systems. Previous implementations used Ilog Solver and its IlcInterval library. The fact that this interval-based solver was more or less a black box implied several strong limitations.
The new results described in this paper come from the integration of IBB with the interval-based library developed by the second author. This new library allows IBB to become reliable (no solution is lost) while still gaining several orders of magnitude w.r.t. solving the whole system. We compare several variants of IBB on a sample of benchmarks, which allows us to better understand the behavior of IBB. The main conclusion is that the use of an interval Newton operator inside blocks has the most positive impact on the robustness and performance of IBB. This modifies the influence of other features, such as intelligent backtracking and filtering strategies.

Keywords: intervals, decomposition, solving sparse systems

1 Introduction

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

In several applications made of non linear constraints, systems are sufficiently sparse to be decomposed by equational or geometric techniques. CAD, scene reconstruction with geometric constraints [23], 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 [13, 10, 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 [14]. *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.

Contributions

The new results described in this paper come from the integration of `IBB` with our own interval-based library.

- This new library allows `IBB` to become reliable (no solution is lost) while still gaining several orders of magnitude w.r.t. solving the whole system.
- An extensive comparison on a sample of 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 interval filtering – IBF).

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 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 [23]).

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 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).

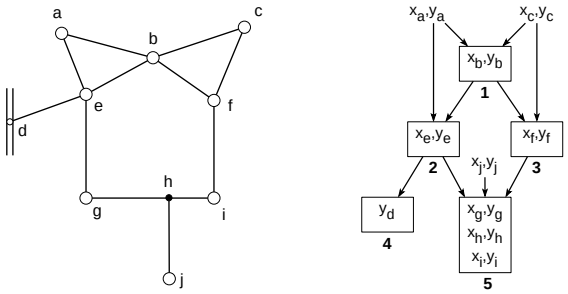

Fig. 1. Didactic problem and its DAG.

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 continues recursively in 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, one has to backtrack, i.e., the next solution of block $i-1$ is chosen, if any.

The reader should notice a significant difference between `IBB[BT]` and the chronological backtracking schema used in finite 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.

```
Algorithm IBB[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.
```

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. Block 2 must thus be solved twice, and with different equations, in case of backtracking, one for each value of x_b.

3.3 `BlockSolve` 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.

We present here a brief introduction of the most common 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 numeric CSPs.

Definition 1 *A numeric 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 can be represented by an n-dimensional parallelepiped called **box**. Because real numbers cannot be represented in computer architectures, the bounds of $\mathbf{x_i}$ are 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_1, where w_1 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 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_1 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 [19, 9, 17].
3. *Interval analysis/unicity test:* Such operators use the first and/or second derivatives of equations. 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.

Filtering/propagation

Propagation is performed by an AC3-like fix-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` [9] algorithms both consider one constraint at a time (like AC3) and reduce the bounds of the implied 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 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 sub-routine and a refutation principle (shaving; similar to the Singleton Arc Consistency [5] 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.

Interval analysis

We have implemented in our library two operators: the `Krawczyck` operator and the interval Newton (`I-Newton`) operator [22]. Both use an iterative numerical process, based on the first derivatives of equations, and extended to intervals. Without detailing these algorithms, 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 modifies 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 or equal to 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_1 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_1), which highlights the significant reduction obtained in the last iteration (see Section 7.2).

Interval techniques and block solving

Let us stress a characteristic of the equation systems corresponding to blocks when they are solved by interval-based techniques: the equations contain coefficients that are not scalar but (small) intervals. Indeed, the solutions obtained in a given block are atomic boxes and become parameters in 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 occasionnally 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 scalar 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 scalar parameters has the following meaning: if one takes *any* scalar 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 scalar values. Although this proposition has not been published (to our knowledge), this is straightforward to extend the "scalar" proof to systems in which the parameters are intervals.

Remark. In the old version using the `IlcInterval` library of Ilog Solver, the unicity test was closely associated to the `Box-consistency`. It made difficult to understand why mixing `Box` and `2B` was fruitful. An interesting consequence of the integration of our interval-based solver in `IBB` is that we now know that it was not due to the `Box-consistency` itself, but related to the unicity test that avoids bisection steps in the bottom of the search tree, and to the use of the *centered form* of the equations that produces additional pruning.

4 Variants of `IBB`

Since 1998, several variants of `IBB` have been tested [2, 14]. In particular, sophisticated versions have been designed to exploit the partial order between blocks. Indeed, `IBB[BT]` uses only the total order between blocks and forgets the actual dependencies between them. 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* [20, 2]. In 2003, we have also proposed a simpler variant `IBB[GBJ]` [14] based on the *Graph-based BackJumping* (GBJ) proposed by Dechter [6].

4.1 The recompute condition

In addition to intelligent backtracking schemes mentioned above, 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*. 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.

4.2 Inter-block filtering (IBF)

Inter-block filtering (in short *IBF*) can be incorporated into any variant of `IBB` using an interval-based solver. The principle of *IBF* is the following. Instead of limiting the filtering process to the current block i, we have extended the scope

of filtering to all the variables. More precisely, before solving a block i, one forms a subsystem extracted from the *friend blocks* F_i' of block i:

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].

IBF is integrated into `IBB` in the following way. When a bisection is applied 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 friend blocks of i.

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_1 and thus cannot be reduced further.

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.

The experiments confirm that it is always fruitful to perform a sophisticated filtering process inside blocks, whereas *IBF* (on the whole system) produces sometimes, but not always, additional gains in performance.

4.3 Mixing IBF and the recompute condition

Incorporating *IBF* into `IBB[BT]` is straightforward. This is not the case for the other variants of `IBB`. 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 cause 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

[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.

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 `IBB[BT+,IBF]` relatively to `IBB[BT,IBF]`.

4.4 Two implementations of `IBB`

A simple variant of `IBB`, called `BB`, is directly implemented in our interval-based solver. `BB` handles the entire system of equations as a numeric CSP, and uses a bisection heuristics based on blocks. The heuristics can only choose the next variable to be split inside the current block i. The specific variable inside the block i is chosen with a standard *round robin* strategy. Since a total solution is computed with a depth-first search in a standard interval-based solving, the recompute condition cannot be incorporated. Subsystems corresponding to blocks must be created to prune inside the blocks. Filtering on the whole system implements `IBF` in a simple way and produces the `BB[BT,IBF]` version. Otherwise we get the simple `BB[BT]` version.

It appears that `BB` is not robust against the **multiple solutions** problem that often occurs in practice. With interval solving, multiple solutions occur when several atomic boxes are close to each other: only one of them contains a solution and the others are not discarded by filtering. Even when the number of multiple solutions is small, `BB` explodes because of the multiplicative effect of the blocks (the multiple partial solutions are combined together). In order that this problem occurs more rarely, one can reduce the precision (i.e., enlarge w_1) 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).

Our second implementation, called `IBB`, does not explode because it takes the union of the multiple solutions (i.e., the hull of the solutions). `IBB` considers the different blocks separetely, so that all the solutions of a block can be computed before solving the next one. This allows `IBB` to merge multiple solutions together. This implementation is completely independent from the interval-based solver and allows more freedom in the creation of new variants (those with the prefix `IBB` in their name). Intelligent backtracking schemes and the recompute condition can be incorporated. However, as compared to `BB`, a slight overcost is sometimes caused by the explicit management of the friend blocks. Note that the previous implementations of `IBB` might lose some solutions because only one of the multiple partial solutions inside blocks was selected (i.e., no hull was performed) and might lead to a failure in the end when the selected atomic box did not contain a solution.

5 New contributions

The integration of our interval-based solver underlies three types of improvements of `IBB`. As previously mentioned, using a white box allows us to better understand what happens. Second, `IBB` is now reliable. The multiple solutions problem has been handled and an ancient **midpoint heuristics** is now abandoned. This heuristics replaced every parameter, i.e., input variable, of a block by a floating-point number in the "middle" of its interval. This heuristics was

necessary because the `IlcInterval` library previously used did not allow the use of interval coefficients. Our new solver accepts non scalar coefficients so that no solution is lost anymore, making thus `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!

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, can often reach the finest precision (which is of great interest due to the multiplicative effect of the blocks) and often certifies the solutions. Hence, the combinatorial explosion due to multiple solutions has been drastically limited. Moreover, the use of `I-Newton` alters the comparison between variants. 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 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.

6 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. `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.

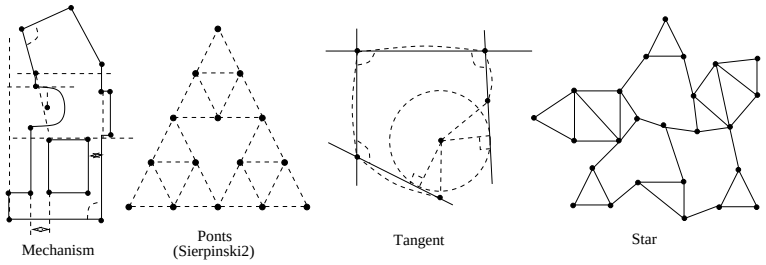

Fig. 2. 2D benchmarks: general view

The `DAGs` of blocks for the benchmarks have been obtained either with an equational method (abbrev. equ.) 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 is generally similar to assigning $]-\infty,+\infty[$ to every variable. 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 fractal *Sierpinski* at level 3, that is, 3 Sierpinski at level 2 (i.e., `Ponts`) put together. The time spent for the equational and geometric decompositions is always negligible, i.e., a few milliseconds for all the benchmarks.

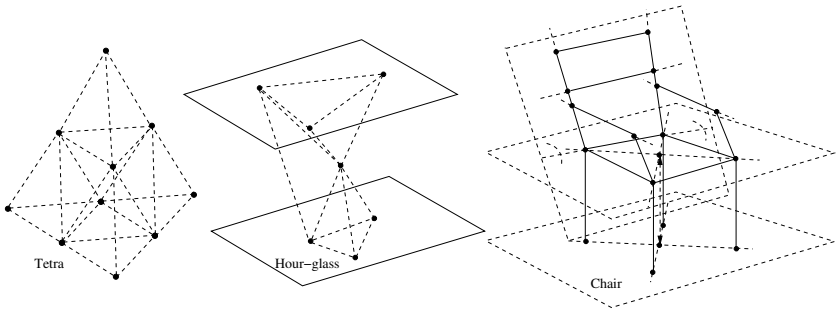

Fig. 3. 3D benchmarks: general view

GCSP	Dim.	Dec.	Size	Size of blocks	Dom.	#sols	w_1
`Mechanism`	2D	equ.	98	98 = 1x10, 2x4, 27x2, 26x1	10	448	5.10^{-6}
`Sierp3`		geo.	124	124 = 44x2, 36x1	1	198	10^{-8}
`PontsE`		equ.	30	30 = 1x14, 6x2, 4x1	100	128	10^{-8}
`PontsG`		geo.	38	38 = 13x2, 12x1	100	128	10^{-8}
`TangentE`		equ.	28	28 = 1x4, 10x2, 4x1	100	128	10^{-8}
`TangentG`		geo.	42	42 = 2x4, 11x2, 12x1	100	128	10^{-8}
`Star`		equ.	46	46 = 3x6, 3x4, 8x2	100	128	10^{-8}
`Chair`	3D	equ.	178	178 = 1x15,1x13,1x9,5x8,3x6,2x4,14x3,1x2,31x1	100	8	5.10^{-7}
`Tetra`		equ.	30	30 = 1x9, 4x3, 1x2, 7x1	100	256	10^{-8}
`Hourglass`		equ.	29	29 = 1x10, 1x4, 1x3, 10x1	100	8	10^{-8}

Table 1. Details on the benchmarks. 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_1).

7 Experiments

We have applied several variants of `IBB` on the benchmarks described above. All the tests have been conducted using the interval-based library implemented in `C++` by the second author [4]. 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 w_2 equal to 10^{-4} has been chosen: a constraint is not pushed in the propagation queue if the projection on its variables has reduced the corresponding intervals less than w_2. `I-Newton` is run when the largest interval in the current box is less than 10^{-2}. 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 *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.

7.1 Interest of system decomposition

The first results show the drastic improvement due to `IBB` as compared to four interval-based solvers tuned to obtain the best results on the benchmarks. All

the solvers use a round-robin splitting strategy. `Ilog Solver` [11] uses a filtering process mixing `2B-consistency` and `Box-consistency`. 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.

GCSP	Home	`RealPaver`	`Ilog Solver`	Best `IBB`	Home/(Best `IBB`)	Precision
`Mechanism`	85	256	> 4000	1.37	62	2.10^{-5}
`Sierp3`	1426	> 4000	> 4000	1.18	1208	4.10^{-11}
`Chair`	> 4000	> 4000	> 128 equations	0.5	> 8000	10^{-7}
`Tetra`	461	98^*	> 4000	0.92	501	2.10^{-14}
`PontsE`	10.3	9.5	103	0.97	10	7.10^{-14}
`PontsG`	5.6	15.7	294	0.31	18	10^{-13}
`Hourglass`	15.3	12.6^*	247	0.4	38	10^{-13}
`TangentE`	58	25.9^*	191	0.1	580	5.10^{-14}
`TangentG`	2710	2380	`XXX`	0.09	30000	4.10^{-14}
`Star`	2381	237^*	1451	0.08	30000	2.10^{-11}

Table 2. Interest of `IBB`. The columns in the left report the times (in seconds) spent by three interval-based solvers to handle the systems globally. *Home* corresponds to our own library. The column *Best* `IBB` reports the best time obtained by `IBB`. Gains of 1, 2, 3 or 4 orders of magnitude are highlighted by the column *Home/(Best* `IBB`*)*. The last column reports the size of the obtained atomic boxes. The obtained precision is often good (as compared to the specified parameter w_1 – see Table 1) thanks to the use of `I-Newton`. An entry `XXX` means that the solver is not able to isolate solutions.

`RealPaver` [7, 8] and our library use a `HC4+Newton` filtering and obtain in this case comparable performances. `RealPaver` uses `HC4+Newton+weak3B` on several benchmarks (entries with a *) and obtains better results. This highlights the interest of the *weak 3B-consistency* [7]. Our library uses a `HC4+Newton+3B` filtering on `Hourglass`.

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.

7.2 Main results

Tables 3, 4 and 5 report the main results we have obtained on several variants of `IBB`. Table 3 first shows that the different variants obtain similar CPU time results provided that `HC4+Newton` filtering is used. The conclusion is different otherwise, as shown in Table 4.

– `BT+` *vs* `BT`*:* `BT+` always reduces the number of bisections and the number of recomputed blocks. The reason why `BB[BT,IBF]` is better in time than `IBB[BT+,IBF]` on `Sierp3` and `PontsG` is the more costly management of friend blocks in the `IBB` implementation (see Section 4.4).

– `IBB` *implementation vs* `BB`*:* Table 4 clearly shows the combinatorial explosion of `BB` involved by the multiple solution problem. The use of `I-Newton` limits this problem, except for `Mechanism` (see Table 3, columns 4 and 5). However, it is important to explain that the problem needed also to be fixed by manually

GCSP	1 IBB[BT]	2 BB[BT]	3 IBB[BT+]	4 IBB[BT,IBF]	5 BB[BT,IBF]	6 IBB[BT+,IBF]
Mechanism	137 6186 1635	188 6186 1635	**132** 6101 1496	176 6164 1629	243 6808 1629	172 6142 1570
Sierp3	284 19455 21045	228 19455 21045	177 9537 11564	269 2874 3671	**118** 2874 3671	234 2136 3035
Chair	50 3814 344	51 3814 344	**18** 1176 97	107 3806 344	86 3806 344	76 2329 148
Tetra	100 3936 235	125 3936 235	**92** 3391 99	136 3942 235	137 3942 235	123 3397 99
PontsE	**98** 3091 131	**97** 3091 131	**97** 3046 86	121 3072 115	122 3072 115	120 3031 74
PontsG	117 8415 9283	96 8415 9283	83 5524 6825	50 1303 1011	**31** 1303 1011	51 1303 1011
Hourglass	40 995 19	41 995 19	40 995 15	43 994 19	45 994 19	42 994 19
TangentE	**11** 186 427	**10** 186 427	**10** 172 405	19 186 427	16 186 427	19 186 427
TangentG	11 402 411	13 402 411	**9** 83 238	20 402 411	17 402 411	20 402 396
Star	28 1420 457	25 1420 457	17 584 324	18 479 251	18 479 251	**8** 90 31

Table 3. Variants of IBB with HC4 + I-Newton filtering. 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.

selecting an adequate value for the parameter w_1. In particular, BB undergoes a combinatorial explosion on Chair and Mechanism when the precision is higher (i.e., when w_1 is smaller). On the contrary, the IBB implementation automatically adjusts w_1 in every block according to the width of the largest input variable (parameter) interval. This confirms that the IBB implementation is rarely worse in time than the BB one, is more robust and it can add the recompute condition.

– *Moderate interest of* IBF*:* Table 4 shows that IBF is not useful when only HC4 is used to prune inside blocks. As explained at the end of Section 5, when I-Newton is also called to prune inside blocks, IBF becomes sometimes useful: three instances benefit from the inter-block filtering (see Table 3). Moreover, IBF avoids using sophisticated backtracking schemas, as shown in the last table.

– *No Interest of intelligent backtracking:* Table 5 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], IBB[GPB]) is compensated by a gain in filtering with IBF. The number of backjumps is drastically reduced by the use of IBF ($6 \rightarrow 0$ on Star; $2974 \rightarrow 135$ on Sierp3). The times obtained with IBF are better than or equal to those obtained with intelligent backtracking schemas. Only a marginal gain is obtained by IBB[GBJ,IBF] w.r.t. IBB[BT+,IBF] for Sierp3.

GCSP	IBB[BT]	BB[BT]	IBB[BT+]	IBB[BT,IBF]	BB[BT,IBF]	IBB[BT+,IBF]	Precision
	281		277	361	890	354	
Mechanism	52870	XXX	52711	40696	40696	40650	7.10^{-4}
	1635		1496	1629	1629	1570	
	326		201	306		260	
Sierp3	116490	XXX	64538	15594	XXX	12228	7.10^{-6}
	21045		11564	3671		3035	
	50	10^4	17	110		82	
Chair	4408		1385	4316	XXX	2468	7.10^{-6}
	344		97	344		148	
	347		336	506		484	
Tetra	47406	XXX	42420	28140	XXX	24666	6.10^{-6}
	235		101	235		100	
	843		839	1379		1365	
PontsE	96522	XXX	95949	66033	XXX	65630	3.10^{-7}
	131		86	115		74	
	117		134	69		70	
PontsG	8415	XXX	51750	7910	XXX	7910	7.10^{-7}
	9283		9283	1011		1011	
	130	879	132	221	465	220	
Hourglass	12308		12308	11599	22007	11599	4.10^{-7}
	19		15	19	39	19	

Table 4. Variants of IBB with HC4 filtering. The last column highlights the rather bad precision obtained.

GCSP	1 IBB[BT]	2 IBB[BT+]	3 IBB[GBJ]	4 IBB[GPB]	5 BB[BT,IBF]	6 IBB[BT,IBF]	7 IBB[BT+,IBF]	8 IBB[GBJ,IBF]
	284	177	134		**118**	269	234	230
Sierp3	19455	9537	6357	> 64 blocks	2874	2874	2136	2088
	21045	11564	7690		3671	3671	3035	2867
			BJ=2974					BJ=135
	28	17	15	**8**	18	18	**8**	**8**
Star	1420	584	536	182	479	479	90	90
	457	324	277	64	251	251	31	31
			BJ=6					BJ=0

Table 5. No interest of intelligent backtracking.

Remarks

IBB can often certify solutions of a decomposed system. A straightforward induction ensures that a solution is certified iff all the corresponding partial solutions are certified in every block. Only solutions of Mechanism and Chair have not been certified.

8 Conclusion

IBB[BT+] is often the best tested version of IBB. It is fast, simple to implement and robust (IBB implementation). Its variant IBB[BT+,IBF] can advantageously replace a sophisticated backtracking schema in case some backjumps occur.

Anyway, the three tables above clearly show that the main impact on robustness and performance is due to the mix of local filtering and interval analysis operators inside blocks. To complement the analysis reported in Section 5, a clear indication is the good behavior of simple versions of IBB, i.e., IBB[BT] and BB[BT,IBF], when such filtering operators are used (see Table 3). Tables 4 and 5 show that the influence of IBF or intelligent backtracking is less significant.

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 scalar coefficients.

References

1. F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising Hull and Box Consistency. In *ICLP*, pages 230–244, 1999.
2. C. Bliek, B. Neveu, and G. Trombettoni. Using Graph Decomposition for Solving Continuous CSPs. In *Proc. CP'98, LNCS 1520*, pages 102–116, 1998.
3. W. Bouma, I. Fudos, C.M. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer Aided Design*, 27(6):487–501, 1995.
4. G. Chabert. *Contributions à la résolution de Contraintes sur intervalles ?* Phd thesis, Université de Nice–Sophia Antipolis, 2006. (to be defended).
5. R. Debruyne and C. Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. of IJCAI*, pages 412–417, 1997.
6. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.
7. L. Granvilliers. `RealPaver` *User's Manual, version 0.3.* University of Nantes, 2003. Available at `www.sciences.univ-nantes.fr/info/perso-/permanents/granvil/realpaver`.
8. L. Granvilliers. Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, , Accepted for publication.
9. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica : A Modeling Language for Global Optimization*. MIT Press, 1997.
10. C. Hoffmann, A. Lomonossov, and M. Sitharam. Finding solvable subsets of constraint graphs. In *Proc. Constraint Programming CP'97*, pages 463–477, 1997.
11. ILOG, Av. Galliéni, Gentilly. *Ilog Solver V. 5, Users' Reference Manual*, 2000.
12. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer-Verlag, 2001.
13. C. Jermann, B. Neveu, and G. Trombettoni. Algorithms for Identifying Rigid Subsystems in Geometric Constraint Systems. In *Proc. IJCAI*, pages 233–38, 2003.
14. C. Jermann, B. Neveu, and G. Trombettoni. Inter-Block Backtracking: Exploiting the Structure in Continuous CSPs. In *Proc. of* 2^{nd} *Int. Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS'03)*, 2003.
15. C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis. Decomposition of Geometric Constraint Systems: a Survey. *Int. Journal of Computational Geometry and Applications (IJCGA)*, 16, 2006.
16. R.S. Latham and A.E. Middleditch. Connectivity analysis: A tool for processing geometric constraints. *Computer Aided Design*, 28(11):917–928, 1996.
17. Y. Lebbah. *Contribution à la résolution de Contraintes par Consistance Forte*. Phd thesis, Université de Nantes, 1999.
18. Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis*, 42(5):2076–2097, 2005.
19. O. Lhomme. Consistency Tech. for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.
20. D.A. McAllester. Partial order backtracking. Research Note, Artificial Intelligence Laboratory, MIT, 1993. ftp://ftp.ai.mit.edu/people/dam/dynamic.ps.
21. Jean-Pierre Merlet. Optimal design for the micro robot. In *IEEE Int. Conf. on Robotics and Automation*, 2002.
22. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
23. M. Wilczkowiak, G. Trombettoni, C. Jermann, P. Sturm, and E. Boyer. Scene Modeling Based on Constraint System Decomposition Tech. In *Proc. ICCV*, 2003.