Inter-Block Backtracking: Exploiting the Structure in Continuous CSPs

Bertrand Neveu[1], Christophe Jermann[2], and Gilles Trombettoni[1]

[1] COPRIN Project, CERMICS-I3S-INRIA, 2004 route des lucioles,
06902 Sophia.Antipolis cedex, B.P. 93, France
{neveu, trombe}@sophia.inria.fr
[2] Laboratoire IRIN, Université de Nantes
2, rue de la Houssinière, B.P. 92208, 44322 Nantes cedex 3, France
Christophe.Jermann@irin.univ-nantes.fr

Abstract. This paper details a technique, called inter-block backtracking (IBB), which improves interval solving of decomposed systems with non-linear equations over the reals.
This technique, 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. All solutions are obtained by combining the solutions computed in the different blocks. The approach seems particularly suitable for improving interval solving techniques.
In this paper, we analyze into the details the different variants of IBB which differ in their backtracking and filtering strategies. We also introduce IBB-GBJ, a new variant based on Dechter's graph-based backjumping.
An extensive comparison on a sample of eight CSPs allows us to better understand the behavior of IBB. It shows that the variants IBB-BT+ and IBB-GBJ are good compromises between simplicity and performance. Moreover, it clearly shows that limiting the scope of the filtering to the blocks is very useful. For all the tested instances, IBB gains several orders of magnitude as compared to a global solving.

Keywords: intervals, decomposition, backtracking, solving sparse systems

1 Introduction

Only a few techniques can be used to compute all the solutions to a system of continuous non-linear constraints. Symbolic techniques, such as the Groebner bases [4] and Ritt-Wu methods [19] are often very time-consuming and are limited to algebraic constraints. The continuation method, also known as the homotopy technique [12, 7], may give very satisfactory results. However, finding a solution over the reals (and not the complex numbers) is not straightforward. Moreover, using it within a constraint solving tool is difficult. Indeed, the continuation method must start from an initial system "close" to the one to be solved. This renders the automatization difficult, especially for non algebraic systems.

Interval techniques are promising alternatives. They obtain good results in several fields, including robust control [10] and robotics [16]. 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 [18, 17], molecular biology and robotics represent such promising application fields. Different techniques can be used to decompose such systems into $k \times k$ *blocks*. Equational decompositions work on the graph made of variables and equations [2, 1]. When equations model geometric constraints, geometric decompositions generally produce smaller blocks [11, 9].

An original approach, introduced in 1998 [2], and called in the present paper *Inter-Block Backtracking* (`IBB`), can be used after this decomposition phase. Following the partial order between blocks given by the decomposition, a solving process can be applied within the blocks, tackling thus systems of reduced size. `IBB` combines the obtained partial solutions to construct the solutions of the problem.

Although `IBB` could be used with other types of solvers, we have integrated interval techniques which are general-purpose and more and more efficient. The first paper [2] presented first versions of `IBB` which included several backtracking schemas, along with an equational decomposition technique. Since then, several variants of `IBB` have been developed which had never been detailed before ([11] focussed on the geometric decomposition techniques based on flow machinery.)

Contributions

This paper details the solving phases performed by `IBB` with interval techniques. It brings several contributions:

- Numerous experiments have been performed on existing and new benchmarks of bigger size (between 30 and 178 equations). This leads to a more fair comparison between variants. Also, this confirms that `IBB` can gain several orders of magnitude in computing time as compared to interval techniques applied to the whole system. Finally, it allows us to better understand subtleties when integrating interval techniques into `IBB`.
- A new version of `IBB` is presented, based on the well-known `GBJ` by Dechter [6]. The experiments show that `IBB-GBJ` is a good compromise between previous versions.
- An inter-block interval filtering can be added to `IBB`. Its impact on performance is experimentally analyzed.

Contents

Section 2 gives some hypotheses about the problems that can be tackled. Section 3 recalls the principles behind `IBB` and interval solving. Section 4 details `IBB-GBJ` and the inter-block interval propagation strategy. Section 5 reports experiments performed on a sample of eight benchmarks. A discussion is given in Section 6 on how to correct a heuristics, used inside `IBB`, that might lead to a loss of solutions.

2 Assumptions

IBB works on a decomposed system of equations over the reals. Any type of equation can be tackled a priori, algebraic or not. Our benchmarks contain linear and quadratic equations. IBB is used for finding **all** the solutions of a constraint system. It could be modified for global optimization (selecting the solution minimizing a given criterion) by replacing the inter-block backtracking by a classical branch and bound. Nothing has been done in this direction so far.

We assume that the systems have a finite set of solutions, that is, the variety of the solutions is 0-dimensional. This condition also holds on every sub-system (block), which allows IBB to combine together a finite set of partial solutions.

Because the conditions above are also respected for our benchmarks and because one equation can generally fix one of its variables, the system is *square*, that is, it contains as many equations as variables to be assigned; the blocks are square as well.

No more hypotheses must hold on the decomposition technique. However, since we use a structural decomposition, the system must include no redundant constraint, that is, no dependent equations. Inequalities or additional equations must be added during the solving phase in the block corresponding to their variables (as "soft" constraints in Numerica [8]), but this integration is out of the scope of this article.

For handling redundant equations, decompositions based on symbolic techniques can be envisaged [5]. These algorithms take into account the *coefficients* of the constraints, and not only the structural dependencies between variables and constraints.

Remark

In practice, the problems which 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, read on a sketch, given by a preliminary process (e.g., in scene reconstruction [18]), or may come from the modeling (e.g., in robotics, the degrees of freedom are chosen during the design of the robot and serve to pilot it).

3 Background

First, this section briefly presents interval solving. The simplest version of IBB is then introduced on an example.

3.1 Interval techniques

Continuous CSP

A continuous CSP $P = (V, C, I)$ contains a set of constraints C and a set of n variables V. Every variable $v_i \in V$ can take a real value in the interval $d_i \in I$; the bounds of d_i are floating-point numbers. Solving P consists in assigning variables in V to values such that all the constraints in C are satisfied.

A n-set of intervals can be represented by an n-dimensional parallelepiped called **box**. Reals cannot be represented in computer architectures, so that a solving process reduces the initial box and stops when a very small box has been obtained. Such a box is called an **atomic box** in this paper. In theory, an interval could have a width of one float at the end. In practice, the process is interrupted when all the intervals contain w_1 floats[1]. It is important to highlight that an atomic box does not necessarily contain a solution. Indeed, the process is semi-deterministic: evaluating an equation with interval arithmetic can prove that the equation has no solution (when the left and right boxes do not intersect), but cannot assert that there exists a solution in the intersection of left and right boxes.

The interval solver used in `IBB`
We use `IlogSolver` version 5.0 and its `IlcInterval` library. `IlcInterval` implements most of the features of the language `Numerica` [8]. These libraries use several principles developed in interval analysis and in constraint programming. The interval solving process used with `IBB` can be summarized as follows:

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 (on constraints handled individually) or a more global one (`3B`) is used to reduce the current box. If the current box becomes empty, the corresponding branch (with no solution) in the search tree is cut [14, 8].
3. *Unicity test:* It is performed on the whole system of equations. It takes into account the current box B and the first and/or second derivatives of equations. When it succeeds, it finds a box B' that contains a unique solution. Also, a specific local numeric algorithm, starting from the center of B', can converge to the solution. Thus, this test generally avoids further bisection steps on B.

The three steps are iteratively performed. The process stops when an atomic box of size less than w_1 is obtained, or when the unicity test is verified on the current box.

Propagation is performed by an AC3-like fix-point algorithm. Four types of filtering reduce the bounds of intervals (no hole is created in the current box). The `box-consistency` [8] comes from `IlcInterval`; the `2B-consistency` works in `IlogSolver`. Although algorithmically different, they both consider one constraint at a time for reducing the bounds of the implied variables (like AC3), and can be used together. The `3B-consistency` [14] uses the `2B-consistency` as sub-routine and a refutation principle (shaving) to reduce the bounds of every variable iteratively. The `bound-consistency` follows the same principle, but uses

[1] w_1 is a user-defined parameter. In most implementations, w_1 is a width and not a number of floats.

the `box-consistency` as sub-routine. A parameter w_2 is specified for the `bound` or the `3B`: a bound of a variable is not updated if the reduction is less than w_2. The w_1 parameter is also used to avoid a huge number of propagations in case of slow convergence of `2B` or `Box`: a reduction is performed when the portion to be removed is greater than w_1.

The unicity test is implemented in `IlcInterval`. Unfortunatly, due to the implementation, it can be performed only with `Box` or `Bound`, and also cannot be called with `2B` or `3B` alone. This sometimes prevents us from finely analyzing the behavior of the solving.

3.2 `IBB-BT`

`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**, will be replaced by values during the solving of the block. The other variables are called **output variables**. A **square** block has as many equations as output variables. There exists an arc from a block i to a block j iff an equation in j involves at least one variable solved in i. The block i is called parent of j. The `DAG` implies a partial order in the solving performed by `IBB`.

Example

To illustrate the principle of `IBB`, we will take the 2D mechanical configuration example introduced in [2] (see Fig. 1). Various points (white circles) are

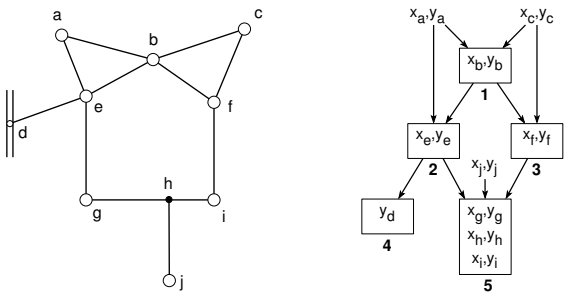

Fig. 1. Didactic problem and its DAG.

connected with rigid rods (lines). Rods impose a distance constraint between two points. Point h (black circle) differs from the others in that it is attached to the rod $\langle g, i\rangle$. 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 technique produces the `DAG` shown in Fig. 1-right.

Illustration of IBB

Respecting the order of the DAG, IBB follows one of the induced total orders, e.g., block 1, then 2, 3, 4, and 5. It first calls the interval-based solver on block 1 and obtains a first solution for x_b (the block has two solutions). Once we have this solution, we can substitute x_b by its value in the equations of subsequent blocks: 2 and 3. Then we process block 3, 4 and 5 in a similar fashion.

When a block has no solution, one has to backtrack. A chronological backtracking goes back to the previous block. IBB computes a different solution for that block and restarts to solve the blocks downstream. However, due to the chronological backtracking of this IBB-BT version, the partial order induced by the DAG is not taken into account. Indeed, in the example above, 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 CBJ and Dynamic backtracking schemas cannot be used to take into account the structure given by the DAG. An intelligent backtracking, IBB-GPB, was introduced, based on the partial order backtracking [15, 2]. The main difficulty in implementing IBB-GPB is to maintain a set of nogoods. Moreover, any modification of IBB-GPB, for adding a feature or heuristics, such as the inter-block filtering, demands a great attention.

We present in this paper a simpler variant based on the graph-based backJumping (in short GBJ) by Dechter [6], and we compare it with IBB-GPB and IBB-BT.

Remarks

The reader should notice a significant difference between IBB and the backtracking schema used in finite CSPs. The domains of variables in a CSP are static, whereas the equation system in a block evolves and so does the corresponding set of solutions. Indeed, when a new solution has been selected in a parent, the corresponding variables are replaced by new values. Hence, the current block contains a new system of equations because the equations have different coefficients.

Due to interval techniques, one does not obtain a solution made of a set of scalars, but an atomic box. Thus, replacing variables from the parent blocks by constants amounts in introducing small constant intervals of width w_1 in the current block to be solved. However, the solver we use does not allow us to define constant intervals. Therefore we need to resort with a **midpoint heuristics** that replaces a constant interval by a (scalar) floating-point number comprised in it (in the "middle"). This heuristics has several significant implications on solving that are discussed in Section 6.1.

4 Use of the DAG structure and inter-block filtering

The structure of the DAG can be taken into account in two ways:

– *top-down:* a *recompute condition* can sometimes avoid to compute again solutions in a block;
– *bottom-up:* when a block has no solution, one can backtrack (or backjump) to a parent block, and not necessarily to the previous block.

The following two subsections present these improvements. The third one details the inter-block filtering which can be added to all the backtracking schemas. This leads to several variants of `IBB` which are fully tested on our benchmarks.

4.1 The recompute condition

This condition can be tested in all the `IBB` variants, even in `IBB-BT`. Testing the recompute condition is not costly and leads to significant gains in performance.

The **recompute condition** states that it is useless to compute a solution in a block if the parent variables have not changed. In that case, `IBB` can reuse the solutions computed the last time the block has been handled. Let us illustrate when it can occur on the didactic example solved by `IBB-BT`.

Suppose that a first solution has been computed in block 3, and that all the solutions computed in block 4 have led to no solution. `IBB-BT` then backtracks on block 3 and the second position of point f is computed. When `IBB` 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 previously computed can be reused at this step. It is easy to avoid this useless computation by using the `DAG`: when `IBB` goes down to block 4, one checks that the parent variables x_e and y_e have not changed, so that the stored solutions can be reused.

4.2 `IBB-GBJ`

Six arrays are used in `IBB-GBJ`:

– `solutions[i, j]` yields the j^{th} solution of block i.
– `#sols[i]` yields the number of solutions in block i.
– `sol_index[i]` yields the index of the current solution in block i.
– `blocks_back[i]` yields the set of blocks that may be the causes of failure of block i. The more recently visited block among them (i.e., the one with the highest number) is selected in case of backtracking.
– `parents[i]` yields the set of parent blocks of block i.
– `assignment[v]` yields the current value assigned to variable v.
– `save_parents[i]` yields the values of the variables in the parent blocks of i the last time i has been solved. This array is only used when the recompute condition is called.

`IBB-GBJ` can find all the solutions to a continuous `CSP`. Based on the `DAG`, the blocks are first ordered in a total order and numbered from 1 to `#blocks`. After an initialization phase, the `while` loop corresponds to the search for solutions, i being the current block. The process ends when $i = 0$, which means that all the solutions below have been found.

```
Algorithm IBB_GBJ (#blocks, solutions, parents, save_parents, assignment)

  for i = 1 to #blocks  do
     blocks_back[i] = parents[i]
     sol_index[i] = 0
     #sols[i] = 0
  end_for

  i = 1
  while (i >= 1) do

      if (Parents_changed? (i, parents, save_parents, assignment)) then
           update_save_parents (i, parents, save_parents, assignment)
           sol_index[i] = 0
           #sols[i] = 0
      end_if

      if (sol_index[i] >= #sols[i])    and
           not (next_solution(i, solutions, #sols)))
      then
          i = backjumping (i, blocks_back, sol_index)
      else  /* solutions [i, sol_index[i] ] are assigned to block i */
          assign_block (i, solutions, sol_index, assignment)
          sol_index[i] = sol_index[i] + 1
          if (i == #blocks)  then  /* total solution found */
             store_total_solution (solutions, sol_index, i)
             blocks_back[#blocks - 1] = {1...#blocks-1}
          else
             i = i + 1
          end_if
       end_if
   end_while
```

The function `next_solution` calls the solver to compute the next solution in the block i. If a solution has been found, the returned boolean is *true*, and the arrays `solutions` and `#sols` are updated. Otherwise, the function returns *false*.

The body corresponding to the first `else` contains actions to be performed when a solution of a block is selected. The procedure `assign_block` modifies the array `assignment` such that the values of the solution found are assigned to the variables of block i. When a total solution is found, `blocks_back[#blocks]` is updated with all the previous blocks to ensure completeness [6]. The recompute condition is checked by the function `Parents_changed?`[2].

[2] A simple way to discard this improvement is to force `Parents_changed?` to always return *true*.

When a block has no solution, a standard function `backjumping` returns a new level j where it is possible to backtrack without losing any solution. It is important to add in the causes of failure of block j (i.e., `blocks_back[j]`) those of block i. Indeed, those blocks are a possible cause of failure for the current value in block i.

```
function backjumping (i, in-out blocks_back, in-out sol_index)

 if blocks-back[i] then
     j = more_recent (blocks_back[i])
     blocks_back[j] = blocks_back[j] U blocks_back[i] \ {j}
 else
     j = 0
 end_if

 for k = j+1 to i do
    blocks_back[k] = parents[k]
    sol_index[k] = 0
 end_for

 return j
```

Favoring the current value

The main drawback of algorithms based on backjumping is that the work performed by the blocks between i and j is lost. When those blocks are handled again, one selects first the current value of a variable, instead of traversing the domain from the beginning. Since the domains are dynamic with `IBB` (the solutions of a block change when new input values are given to it), this improvement can be performed only when the recompute condition allows `IBB` to reuse the previous solutions.

This heuristics has been added to `IBB-GBJ`[3]. However, probably due to the remark above, the gains in performance obtained by the heuristics are small and are not detailed in the description of experiments (see Section 5).

4.3 Inter-block filtering

Contrary to the features above related to backjumping, *inter-block filtering* (in short *ibf*) is specific to interval techniques. *ibf* can thus be incorporated into any variant of `IBB` using an interval-based solver.

In finite CSP instances, it has generally been observed that, during the solving, performing filtering on all the remaining problem is fruitful. Therefore we

[3] The algorithm must manage another index in addition to `sol_index`.

decided to embedd an inter-block filtering in IBB: instead of limiting the filtering process (based on 2B, 3B, Box or Bound in our tool) to the current block, we have extended the scope of filtering to all the variables.

More precisely, before solving a block i, one forms a subsystem of variables and equations extracted from the following blocks:

1. take the set $B = \{i...\#blocks\}$ containing the blocks not yet "instantiated",
2. keep in B only the blocks connected to i in the DAG[4].

Then, the bisection is applied only on block i while the filtering process can be run on all the variables of blocks in B.

To illustrate *ibf*, let us consider the DAG of the didactic example. When block 1 is solved, all the blocks are considered by *ibf* since they are all connected to block 1. Thus, any interval reduction in block 1 can imply a reduction in any variable of the system. When block 2 is solved, a reduction can have an influence on blocks 3, 4, 5 for the same reasons. (Notice that block 3 is not downstream to block 2.) When block 3 is solved, a reduction can have an influence on blocks 5 only. Indeed, after having removed blocks 1 and 2, block 3 and 4 do not belong to the same connected component. In fact, no propagation can reach block 4 since the parent variables of block 5 which are in block 2 have an interval of width at most w_1 and thus cannot be still reduced.

Remark

One must pay attention to the way *ibf* is incorporated in IBB-GBJ. Indeed, the reductions induced by the previous blocks must be regarded as possible causes of failure. This modification is not detailed and we just illustrate the point on the DAG of the didactic example. If no solution is found in block 3, IBB with *ibf* must go back to block 2 and not to block 1. Indeed, when block 2 had been solved, a reduction could have propagated on block 3 (through 5).

5 Experiments

Exhaustive experiments have been performed on 8 benchmarks made of geometric constraints. They compare different variants of IBB and interval solving applied to the whole system (called *global solving* below).

5.1 Benchmarks

Some of them are artificial problems, mostly made of quadratic distance constraints. Mechanism and Tangent have been found in [13] and [3]. Chair is a realistic assembly made of 178 equations from a large variety of geometric constraints: distances, angles, incidence, parallelisms, orthogonalities, etc.

[4] The orientation of the DAG is forgotten at this step, that is, the arcs of the DAG are transformed in non-directed edges, so that the filtering can apply on "brother" blocks.

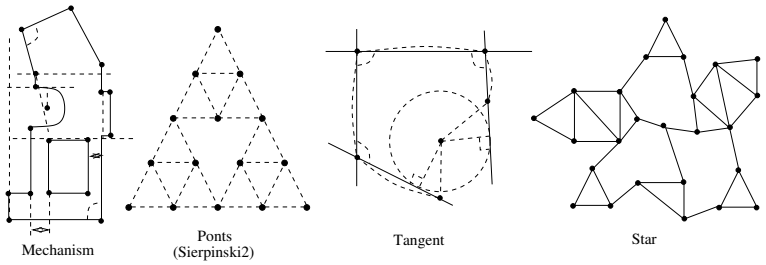

Fig. 2. 2D benchmarks: general view

Dim	GCSP	Dec.	Size	Size Dec.	Ti.	Small	Med.	Large
2D	Mechanism	equ.	98	98 = 1x10, 2x4, 27x2, 26x1	1	8	48	448
	Ponts	equ.	30	30 = 1x14, 6x2, 4x1	1	15	96	128
	Sierpinski3	geo.	84	124 = 44x2, 36x1	1	8	96	138
	Tangent	geo.	28	42 = 2x4, 11x2, 12x1	4	16	32	64
	Star	equ.	46	46 = 3x6, 3x4, 8x2	1	4	8	8
3D	Chair	equ.	178	178 = 1x15,1x13,1x9,5x8,3x6,2x4,14x3,1x2,31x1	6	6	18	36
	Hourglass	geo.	29	39 = 2x4, 3x3, 2x2, 18x1	1	1	2	8
	Tetra	equ.	30	30 = 1x9, 4x3, 1x2, 7x1	1	16	68	256

Table 1. Details on the benchmarks: type of decomposition method. (Dec.); number of equations (Size); Size of blocks (Size Dec.)- NxK means N blocks of size K - # of solutions with the four types of domains selected: tiny (width = 0.1), small (1), medium (10), large (100).

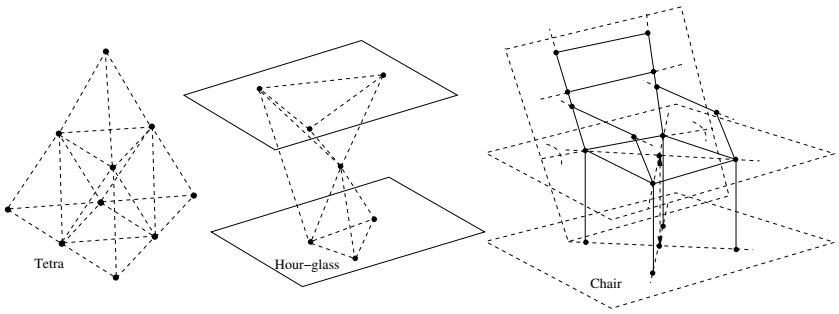

Fig. 3. 3D benchmarks: general view

The domains have been selected around a given solution and lead to radically different search spaces. Note that a problem defined with large domains is generally similar to assign $]-\infty,+\infty[$ to every variable.

`Sierpinski3` is the fractal `Sierpinski` at level 3, that is, 3 `Sierpinski2` put together. The corresponding equation system would have about 2^{40} solutions, so that the initial domains are limited to a width 0.1 (tiny), 0.8 (small), 0.9 (medium), 1 (large).

5.2 Choice of filtering

With the aim of not handicapping the global solving, we select the best filtering algorithms by performing tests on two benchmarks of medium size. Several widths have been tried for w_1 and w_2 (see Table 2).

	w_2	w_1	2B/3B	Box/Bound	2B+Box/3B+Bound
Ponts	0	1e-6	*sing.*	264	**29**
		1e-8	*sing.*	292	**32**
		1e-10	*sing.*	278	**32**
	1e-2	1e-6	116	2078	309
		1e-8	2712	2642	1303
		1e-10	13565	2652	5570
	1e-4	1e-6	84	>54000	523
		1e-8	4413	>54000	5274
Tangent	0	1e-6	*sing.*	547	81
		1e-8	*sing.*	553	82
		1e-10	*sing.*	562	86
	1e-2	1e-6	**26**	265	91
		1e-8	**35**	270	94
		1e-10	**60**	266	93
	1e-4	1e-6	51	2516	369
		1e-8	68	2535	393

Table 2. Comparison of different partial consistencies. The best results appear in boldfaced. A 0 in column w_2 means that the lines 1 and 4 report results obtained by `2B`, `Box`, or `2B+Box`. Otherwise, when w_2 is 1e-2 or 1e-4, the corresponding lines report results obtained by `3B`, `bound`, or `3B+bound`. Cells containing *sing.* (singularity) indicate that multiple solutions are obtained and lead to a combinatorial explosion (see Section 6).

Clearly, `2B+Box` and `3B` outperfom the other combinations. All the following tests have been performed with these two filtering techniques.

5.3 Main tests

The main conclusions about the tests reported in Table 3 are the following:

- `IBB` always outperforms the global solving, which highlights the interest of exploiting the structure. One, two or three orders of magnitude can be gained in performance. Even with tiny domains, the gains can be significant (see `Sierpinski3`)[5].
- The inter-block filtering is always counter-productive and sometimes very bad (see `Tangent`). Several lines with `3B` have been added to show that the loss of time of inter-block filtering is reduced with `3B`.

[5] The global solving compares advantageously with `IBB` on the `Star` benchmark with tiny domains. This is due to a greater precision required for `IBB` to make it complete (see Section 6). With the same precision, the global solving spends 75 s to find the solutions.

– The exploitation of the `DAG` structure by the recompute condition is very useful.

		Tiny		Small		Medium		Large	
		¬IBF	IBF	¬IBF	IBF	¬IBF	IBF	¬IBF	IBF
Chair	Global	XXS		XXS		XXS		XXS	
	BT	3.3	XXS	3.2	XXS	9.4	XXS	12.4	XXS
	BT+	2.4	XXS	2.3	XXS	4.5	XXS	4.7	XXS
	GBJ	2.4	XXS	2.3	XXS	4.5	XXS	4.7	XXS
Mechanism	Global	XXS		XXS		XXS		XXS	
	BT	0.17	14.1	0.6	15.0	2.8	18.7	13.3	32.8
	BT+	0.11	14.1	0.4	13.6	2.6	17.2	13.1	30.6
	GBJ	0.10	14.1	0.4	13.5	2.6	17.3	13.1	30.4
	GPB	0.10	14.2	0.4	13.3	2.7	17.4	13.1	30.5
	3B(GBJ)	0.23	0.68	1.7	2.3	9.7	11	83	88
Ponts	Global	0.73		32		82		110	
	BT	0.16	0.63	2.38	4.2	6.5	10.6	9.1	14.6
	BT+	0.16	0.63	2.36	4.2	6.1	10.2	8.8	14.7
	GBJ	0.17	0.58	2.35	4.1	6.0	10.4	8.7	14.4
	GPB	0.22	0.61	2.37	4.1	6.3	10.4	8.7	14.4
	3B(GBJ)	0.3	0.6	12	15	25	31	49	59
Hour-glass	Global	0.12		1.89		1.47		22.77	
	BT+	0.03	0.88	0.03	1.64	0.06	1.00	0.06	1.21
	GBJ	0.04	0.75	0.03	1.60	0.02	0.83	0.06	1.19
	GPB	0.05	0.73	0.03	1.61	0.05	0.88	0.05	1.15
	3B(GBJ)	0.03	0.3	0.05	0.6	0.05	0.2	0.1	0.4
Sierpinski3	Global	3.1		>54000		>54000		>54000	
	3B(BT)	0.1	1.32	12.3	160	96	788	136	1094
	3B(BT+)	0.1	1.32	12.7	160	67	703	93	928
	3B(GBJ)	0.1	1.32	12	166	61	682	85	916
Tangent	Global	0.5		35		39		46	
	BT+	0.05	1.26	0.11	1.89	0.13	7.63	0.20	8.15
	BJ	0.07	1.17	0.11	1.89	0.14	7.69	0.19	8.00
	GPB	0.07	1.19	0.10	1.93	0.11	7.69	0.22	8.04
	3B(GBJ)	0.2	0.7	0.2	1.3	0.2	1.3	0.3	1.7
Tetra	Global	2.15		92		197		406	
	BT+	0.14	0.74	1.08	4.00	2.37	7.01	4.73	13.56
	GBJ	0.14	0.67	1.10	3.87	2.30	6.80	4.74	13.20
	GPB	0.16	0.65	1.11	3.90	2.29	6.71	4.72	13.19
Star	Global	8.7		2908		2068		1987	
	BT	9.96	70	40.2	137	81.4	241	80.3	240
	BT+	9.96	70	29.5	99.6	78.1	102	78	102
	GBJ	9.96	70	29.1	99.6	77.9	102	77.9	102
	GPB	9.96	70	29.4	99.6	49.3	102	49	102

Table 3. Results of experiments. `BT+` is `IBB-BT` with the recompute condition. For every algorithm and every domain size, times are given either without inter-block filtering (¬IBF) or with IBF. All the times are obtained in seconds on a `PentiumIII 935 Mhz` with a Linux operating system. The reported times are obtained with `2B+Box` which is often better than `3B`. The lines `3B(GBJ)` report times with `IBB-GBJ` and `3B` when it is competitive.

Remark

Entries in Table 3 containing XXS correspond to a failure in the solving process due to `IlogSolver` when a maximum size is exceeded.

To refine our conclusions, Table 4 reports statistics made on the number of backjumps performed by `IBB-GBJ`. Note that no backjumps have been observed with the other four benchmarks.

These experiments highlight a significant result. Most of the backjumps disappear with the use of inter-block filtering, which reminds similar results observed

in finite CSPs. However, the price paid by inter-block filtering for removing these backjumps does not bring in good returns. `Sierpinski3`-Large highlights the trend: 2114 on 2118 backjumps are eliminated by inter-block filtering, but the algorithm is 10 times slower than `IBB-GBJ`!

	IB filtering	Tiny	Small	Medium	Large
Ponts	no	0	0	1	0
	yes	0	0	0	0
Mechanism	no	3	4	0	0
	yes	0	0	0	0
Star	no	0	2	6	6
	yes	0	0	0	0
Sierpinski3	no	0	12	829	2118
	yes	0	0	5	4

Table 4. Number of backjumps with and without inter-block filtering

6 Discussion

Two difficulties come from the use of interval techniques with `IBB`. They are detailed below.

6.1 Midpoint heuristics

This heuristics (see Section 3.2) is not satisfactory because solutions might be lost, making the whole process incomplete[6]. When the midpoint heuristics had been introduced [2], our examples were small, and the case had not occurred. Since then, it has occurred with `Star`, `Sierpinski3` and `Chair`. The problem has been fixed with `Star` and `Sierpinski3` by increasing the precision (i.e., decreasing w_1). Ad-hoc modifications of the equation system must have been made to fix the problem on `Chair`.

The clean solution consists in introducing constant intervals in equations instead of the midpoints (which is not currently possible with `IlogSolver`). We think that the overcost in time would be negligible with the filtering/bisection solving schema. On the contrary, unicity tests may become inefficient because computing the inverse of a jacobian matrix including intervals (instead of scalars) can lead to large overestimates of the intervals.

6.2 Dealing with multiple solutions

Another limit of interval techniques is worsened with `IBB`. Multiple solutions occur when several atomic boxes are close to each other: only one contains a solution and the others are not discarded by filtering. Even when the number

[6] Its correctness can be ensured by a final and quasi-immediate filtering process performed on the whole equation system, where the domains form an atomic box.

of multiple solutions is small, the multiplicative effect due to IBB (the partial solutions are combined together) may render the problem intractable.

An ad-hoc solution consists in improving the precision (i.e., reducing w_1), which fixes some cases. Mixing several filtering techniques, such as 2B+Box, also reduces the phenomenon (The *sing.* entries in Table 2 with 2B come from this phenomenon.) We have implemented a first way to detect multiple solutions. In this case, we select only one of them. This has solved the problem in most cases. A few pathologic cases remain due to an interaction with the midpoint heuristics. Taking the union of the multiple solutions should be more robust.

7 Conclusion

This paper has detailed the generic inter-block backtracking framework to solve decomposed continuous CSPs. We have implemented three backtracking schemas (chronological BT, GBJ, partial order backtracking). Every backtracking schema can incorporate a recompute condition that avoids sometimes a useless call to the solver. Every schema can also use an inter-block filtering.

Series of exhaustive tests have been performed on a sample of benchmarks of acceptable size and made of non-linear equations. First, all the variants of IBB can gain several orders of magnitude as compared to solving the constraint system globally. Second, exploiting the structure of the DAG with the recompute condition is very useful whereas a more sophisticated exploitation (backjumping) only improves slightly the performance of IBB. However, it might lead to important gains while never producing an overhead. This leads us to propose the IBB-GBJ version presented in this paper.

Another clear result of this paper is that inter-block filtering is counter-productive. This highlights that a global filtering which does not take the structure into account makes a lot of useless work.

The next step of our work is to deal with small constant intervals to discard the midpoint heuristics and make our implementation more robust.

References

1. S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Compugraphic*, 1993.
2. Christian Bliek, Bertrand Neveu, and Gilles Trombettoni. Using graph decomposition for solving continuous csps. In *Principles and Practice of Constraint Programming, CP'98*, volume 1520 of *LNCS*, pages 102–116. Springer, 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. B. Buchberger. An algebraic method in ideal theory. In *Multidimensional System Theory*, pages 184–232. Reidel Publishing Co., 1985.
5. B. Mourrain D. Bondyfalat and T. Papadopoulo. An application of automatic theorem proving in computer vision. In *2nd International Workshop on Automated Deduction in Geometry, Springer-Verlag*, 1999.
6. Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

7. C. Durand. *Symbolic and Numerical Techniques For Constraint Solving.* PhD thesis, Purdue University, 1998.
8. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica : A Modeling Language for Global Optimization.* MIT Press, 1997.
9. Christoph Hoffmann, Andrew Lomonossov, and Meera Sitharam. Finding solvable subsets of constraint graphs. In *Proc. Constraint Programming CP'97*, pages 463–477, 1997.
10. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis.* Springer-Verlag, 2001.
11. Christophe Jermann, Gilles Trombettoni, Bertrand Neveu, and Michel Rueher. A constraint programming approach for solving rigid geometric systems. In *Principles and Practice of Constraint Programming, CP 2000*, volume 1894 of *LNCS*, pages 233–248, 2000.
12. E. Lahaye. Une méthode de résolution d'une catégorie d'équations transcendantes. *Compte-rendu des Séances de L'Académie des Sciences*, 198:1840–1842, 1934.
13. R.S. Latham and A.E. Middleditch. Connectivity analysis: A tool for processing geometric constraints. *Computer Aided Design*, 28(11):917–928, 1996.
14. O. Lhomme. Consistency techniques for numeric csps. In *IJCAI*, pages 232–238, 1993.
15. D.A. McAllester. Partial order backtracking. Research Note, Artificial Intelligence Laboratory, MIT, 1993. ftp://ftp.ai.mit.edu/people/dam/dynamic.ps.
16. Jean-Pierre Merlet. Optimal design for the micro robot. In *in IEEE Int. Conf. on Robotics and Automation*, 2002.
17. Gilles Trombettoni and Marta Wilczkowiak. Scene Reconstruction based on Constraints: Details on the Equation System Decomposition. In *Proc. International Conference on Constraint Programming, CP'03*, volume LNCS 2833, pages 956–961, 2003.
18. Marta Wilczkowiak, Gilles Trombettoni, Christophe Jermann, Peter Sturm, and Edmond Boyer. Scene Modeling Based on Constraint System Decomposition Techniques. In *Proc. International Conference on Computer Vision, ICCV'03*, 2003.
19. W. Wu. Basic principles of mechanical theorem proving in elementary geometries. *J. Automated Reasoning*, 2:221–254, 1986.