Finding solutions by finding inconsistencies

Ghiles Ziat, Marie Pelleau, Charlotte Truchet, Antoine Miné

Finding solutions by finding inconsistencies[⋆]

Ghiles Ziat[1], Marie Pelleau[2], Charlotte Truchet[3], and Antoine Miné[1]

[1] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France
[2] Université Côte d'Azur, CNRS, I3S, Nice, France
[3] TASC, LS2N UMR 6004, Univ. Nantes

Abstract. In continuous constraint programming, the solving process alternates propagation steps, which reduce the search space according to the constraints, and branching steps. In practice, the solvers spend a lot of computation time in propagation to separate feasible and infeasible parts of the search space. The constraint propagators cut the search space into two subspaces: the inconsistent one, which can be discarded, and the consistent one, which may contain solutions and where the search continues. The status of all this consistent subspace is thus indeterminate. In this article, we introduce a new step called *elimination*. It refines the analysis of the consistent subspace by dividing it into an indeterminate one, where the search must continue, and a *satisfied* one, where the constraints are always satisfied. The latter can be stored and removed from the search process. Elimination relies on the propagation of the negation of the constraints, and a new difference operator to efficiently compute the obtained set as an union of boxes, thus it uses the same representations and algorithms as those already existing in the solvers. Combined with propagation, elimination allows the solver to focus on the frontiers of the constraints, which is the core difficult part of the problem. We have implemented our method in the AbSolute solver, and present experimental results on classic benchmarks with good performances.

1 Introduction

Constraint solvers generally alternate two steps: propagation and exploration. The propagation step reduces the domains of the variables using the constraints. The exploration step adds hypotheses to divide the problem into several smaller sub-problems. In this article, we are interested in continuous constraint solving, where the variables have real values. In this case, the resolution of a problem usually consists in a paving of the solution space, which is not computer representable in general, using elements which are simple enough to manipulate (often floating-point boxes). This paving may correspond to an outer approximation or over-approximation of solutions, as in Ibex [6], or may correspond to an inner approximation or under-approximation as in [7].

[⋆] The work was supported, in part, by the project ANR-15-CE25-0002 Coverif from the French Agence Nationale de la Recherche, and in part by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

The efficiency of a solver depends on the choices made by the exploration process, these choices being often guided by heuristics. On discrete variables, such heuristics can for example try and provoke early failures (such as *fail-first* [9] or *dom /w deg* [4]).

On continuous variables, classic heuristics include: *largest first* [13], which consists in splitting the largest domain; *round robin*, where the domains are processed successively; or *maximal smear* [8], choosing the domain with the greatest slope based on the derivatives of the constraints. More recently, and closer to our work, *Mind The Gaps* [1] uses the idea from [8, 13] and uses partial consistencies to find interesting splitting points within the domain, according to the "gaps" in the search space: splitting the domains by taking into account such gaps reduces the search space.

In this paper, we focus on covering the entire solution space of continuous problems. We propose to add a new step, complementary to constraint propagation, in the solving process: the *elimination* step. This step divides the search space into two sub-spaces: one containing only solutions, and the other where the constraints are indeterminate — it may contain solutions as well as non-solutions. Our solving method alternates three steps: propagation, elimination, and exploration. It offers another way of reasoning on the constraints, since we are not only exploiting the constraints' consistencies (as does propagation) but also the constraint inconsistencies. With this improved reasoning, the interesting zones of the search space are better targeted: zones without solutions are discarded by propagation, and zones with only solutions are set aside by elimination into the solution space, which means in practice that they also are excluded from the search. The search effort can then focus on the indeterminate space — the part of the search space effectively requiring deeper exploration by the solver.

Our new step can be seen as a new contraction, in the same framework as the contractors described in [10] and used in Ibex [6] to perform a smarter exploration. We add an automatic propagation on the negation of the constraints, to identify subspaces containing only solutions. We thus reason on the negation of the constraints, hence we compute sets which are not boxes: to overcome this issue, we also add an operator on boxes to efficiently compute the difference of two boxes (or the complementary of one box in another) as a union of boxes. Thus, our method can be integrated into any solver without changing its domain representation nor modifying the propagators.

Our elimination phase relies on a notion of consistency to divide the search space and guide the search, similarly to *Mind The Gaps* [1] where consistency is also used to guide the search. But we go a step further by not only trying to identify the inconsistent parts of the search space (the "gaps"), but by using set complement to identify sub-spaces containing only solutions.

Our method is tailored to output an outer approximation as well as an inner approximation of the solution set: when the size of the indeterminate part is small enough and exploration stops, we can either include the indeterminate part with the definite solution space found by elimination steps to get an outer approximation, or return only the solution space found by elimination which is an inner approximation.

In fact, our solver can provide within the same process both an inner and an outer approximation, and due to the fact that the computed boxes better fit the constraints' shapes, this comes at no cost according to our experiments. We have tested our method on both the Coconut[16] and the MinLPLib[5] benchmarks. Our results show that first, our method computes both the inner and outer approximation with no time overhead, and second, it produces fewer boxes as an output, which makes the computed solution much more tractable.

This paper is organized as follows. Section 2 presents formally classic continuous constraint solving, on which our work is based. Section 3 introduces our new solving step: elimination. Section 4 presents experiments with our new solving method. Finally, Sect. 5 concludes and discusses future work.

2 Preliminaries

This section recalls basic notions of continuous Constraint Programming (CP). For a more detailed presentation, we refer the reader to [14, Chap. 16].

2.1 Constraint Satisfaction Problems

We consider a Constraint Satisfaction Problem (CSP) defined by: a set of n variables $\mathcal{X} = \{x_1, \dots, x_n\}$; the domain of each variable $\mathcal{D} = \{d_1, \dots, d_n\}$, i.e., $x_k \in d_k, \forall k \in [1, n]$; and a set of m constraints $\mathcal{C} = (C_1, \dots, C_m)$. A possible assignment of the variables is a tuple in $D = d_1 \times \cdots \times d_n$. A solution of the CSP is an element of D satisfying all the constraints in $\mathcal{C}$. We denote as $\mathcal{S}$ the set of all solutions, i.e., $\mathcal{S} = \{(s_1, \dots, s_n) \in D \mid \forall i \in \{1, \dots, m\}, C_i(s_1, \dots, s_n)\}$. We also denote as $\mathcal{S}_C$ the solution set for the constraint C alone: $\mathcal{S}_C = \{(s_1, \dots, s_n) \in D \mid C(s_1, \dots, s_n)\}$.

In the CP framework, variables can either be discrete or continuous. In this article, we focus on continuous, real-valued variables. Domains of variables are intervals of $\mathcal{R}$. We also assume that the bounds are (finite) floating point numbers, to be computer-representable. They can be either excluded or included. Let $\mathcal{F}$ be the set of finite floating point numbers. For $a, b \in \mathcal{F}$, we define a real-interval as the conjunction of two half-spaces $\{x \in \mathcal{R} \mid a \lhd x \lhd b\}$ where $\lhd \in \{<, \leq\}$, and let $\mathcal{I}$ be the set of all such intervals.

A Cartesian product of intervals is called a box. We note $\mathcal{B} = \mathcal{I}^n$ the set of boxes of dimension n. Note that our definition of interval encompasses intervals with excluded end-points which will be useful later.

For continuous CSPs, with domains in $\mathcal{I}$, the exact solution set $\mathcal{S} \subseteq \mathcal{R}^n$ is generally not computer-representable. Constraint solvers usually return a collection of boxes with floating-point bounds containing the solutions, the union of these being an over-approximation of $\mathcal{S}$.

2.2 Consistency

The notion of *local consistency* is central in CP. We recall the definition of Hull-consistency [3], one of the classic local consistencies for continuous constraints.

Definition 1 (Hull-Consistency). *Let $x_1, \dots, x_n$ be variables over continuous domains represented by intervals $d_1, \dots, d_n \in \mathcal{I}$, and C a constraint. The domains are said to be Hull-consistent for C if and only if $D = d_1 \times \cdots \times d_n$ is the smallest floating-point box containing the solutions for C in D.*

Intuitively, no bound of a consistent box D can be tightened without losing a solution of C. Given a constraint C over domains $d_1, \dots, d_n$, an algorithm that computes locally consistent domains $d'_1, \dots, d'_n$ that contain the same solution set as C in $d_1 \times \cdots \times d_n$ is called a *propagator* for C. Naturally, $\forall k \in [1, n], d'_k \subseteq d_k$. Given a constraint C and domains $d_1, \dots, d_n$, we will write $H_C(d_1, \dots, d_n)$ the corresponding Hull-consistent domains and $\rho_C : \mathcal{B} \to \mathcal{B}$ a propagator for C. While we only refer to the Hull-consistency in this work, our method is based upon the propagator notion and holds for any kind of consistency.

The domains which are locally consistent for all constraints are the largest common fixpoints of all the constraint propagators [2, 15]. In practice, propagators often compute over-approximations of the locally consistent domains. In the following, we will use the standard algorithm HC4 [3], which propagates continuous constraints, relying on the syntax of the constraints and interval arithmetic [11], although our method could be combined with other propagators. HC4 generally does not reach Hull consistency, in particular in case of multiple occurrences of the variables in the constraints.

Local consistency computations can be seen as deductions, performed on domains by analyzing the constraints. If the propagators return the empty set, the domains are inconsistent and the problem has no solution. Otherwise, non-empty local consistent domains are computed. This is often not sufficient to accurately approximate the solution set. In that case, choices are made on the variable values. For continuous constraints, typically a domain d is chosen and split into two (or more) parts, which are in turn narrowed by the propagators. The solver alternates propagation and split phases a given precision is reached, *i.e* all the boxes which are still considered are smaller than a given parameter. Of course, as soon as a box is proven to contain only solutions, it can be removed from the search space and added to the solution set. Upon termination, the collection of boxes returned covers the solution set $\mathcal{S}$, under some hypotheses on the propagators and splits [2].

A solving method is said to be *complete* if it returns an over-approximation of the solution set (no solution is missed). It is said to be *sound* if it returns an under-approximation of the solution set (only solutions are returned). For problems with real variables, the solving method cannot be both complete and sound in general asbeing sound (returning only solution), requires the result to under-approximate the solution space, and being complete (returning all the solutions) requires the result to over-approximate the solution space. In practice, solving methods are often complete and not sound.

Algorithm 1 Solving without / with elimination (in pink)

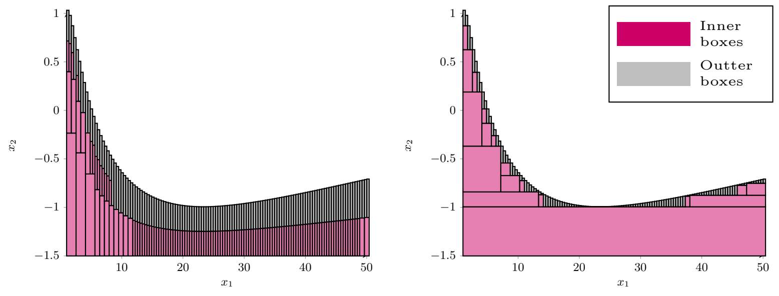
```
1: function SOLVE($\mathcal{D}$, $\mathcal{C}$, $r$, elim)    ▷ $\mathcal{D}$: domains, $\mathcal{C}$: constraints, $r$: real, set elim to
2:                                        false for classic solving, true for elimination
3:     sols ← $\emptyset$                                                  ▷ sound solutions
4:     undet ← $\emptyset$                                         ▷ indeterminate solutions
5:     explore ← $\emptyset$                                               ▷ boxes to explore
6:     $e$ =init($\mathcal{D}$)                                               ▷ initialization
7:     push $e$ in explore
8:     while explore $\neq \emptyset$ do
9:         $e$ ← pop(explore)
10:        $e$ ← filter($e$, $\mathcal{C}$)
11:        if $e \neq \emptyset$ then
12:            if satisfies(e, $\mathcal{C}$) then
13:                sols ← sols $\cup$ $e$
14:            else
15:                if $\tau(e) \leq r$ then
16:                    undet ← undet $\cup$ $e$
17:                else
18:                    if !elim then
19:                        push $\oplus(e)$ in explore              ▷ Classic solving process
20:                    else
21:                        $(S, E)$ = elimination($e$, $\mathcal{C}$)     ▷ Solving with elimination
22:                        sols ← sols $\cup$ $S$
23:                        push $\oplus(E)$ in explore
```

2.3 Solving method

In this article, we rely on the general abstract solving process described in [12], instantiated with the interval domain. The solver thus operates on boxes, as defined above. Algorithm 1 gives the pseudo-code of the abstract solving method, where $\tau \in \mathcal{B} \rightarrow \mathbb{R}$ is the precision measure and $\oplus \in \mathcal{B} \rightarrow \wp(\mathcal{B})$ is the split operator. In this section we have the `elim` parameter set to false, thus we do not consider the part highlighted in pink. By alternating propagation and exploration, Algorithm 1 builds a disjunction of boxes that covers the solution space. It uses three auxiliary functions: `init` $\in \mathcal{D} \rightarrow \mathcal{B}$, `filter` $\in \mathcal{B} \rightarrow \mathcal{B}$, and `satisfies` $\in \mathcal{B} \times \mathcal{C} \rightarrow \{\text{true}, \text{false}\}$. Firstly, `init` creates a box from the initial domains of the problem. Then, `filter` corresponds to the propagation loop: it applies the propagator for each constraint in turn. Finally, `satisfies` checks whether a box satisfies all the constraints, that is, if it contains only solutions. This function corresponds to a contractor as defined in [6].

This solving method works as follows: at each step, the current box is tightened using the propagators on the constraints (function `filter`). After propagation, if the tightened box is not empty, three cases are possible:

– If the box contains only solutions (function `satisfies`), then it is directly added to the set of solutions `sols`.



(a) Default solving (b) Solving vith elimination

Fig. 1: (a): 127 inner boxes, 128 outer boxes. Inner boxes represent 59% of the coverage area. Computation time 0.015s. (b): 18 inner boxes, 128 outer boxes. Inner boxes represent 92% of the coverage area. Computation time: 0.008s

– Otherwise, if the box is small enough with respect to a parameter r ($\tau(e) \leq r$), then it is added to the set of indeterminate solutions `undet` — i.e., the box, which may contain both solutions and non-solutions, is considered small enough to be left out of the search.
– Finally, if the size of the box is larger than r and may contain solutions, as `elim` is set to false, then it is divided using a split operator $\oplus$ and the process is repeated on the resulting boxes.

Figure 1(a) shows the result obtained with Algorithm 1 with `elim` set to false, for a problem with two variables $x_1 \in [1, 50]$ and $x_2 \in [-1.5, 1]$ constrained by $cos(ln(x_1)) > x_2$. Note that this solving method can either produce an under-approximation of the solution set by considering only the inner elements, or an over-approximation by considering all the resulting elements. Figure 1(b) shows that by making different splitting choices, we could avoid computations and reach the given precision with less iterations. We achieve that using Elimination, with wich we obtain fewer, but larger inner boxes. We introduce this new step in the next section and explain how it pushes the reasoning based on the constraints one step further in order to avoid superfluous splitting steps.

3 Elimination

The propagation step reduces the search space by removing non-consistent subspaces. Elimination aims at reducing the search space by focusing on the frontiers of the problem. This is done in three steps: computing the elimination for each constraint, combining the result with the domains with a new difference operator, and finally integrate this mechanism in the solving process.

3.1 Elimination for one constraint

We introduce here the concept of elimination for a single constraint. It relies on the constraint propagator to over-approximate the set of instantiations that *can*

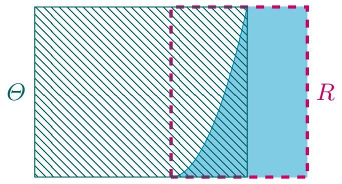


Fig. 2: Given the constraint $y \leq x^3$, in blue, the box R over-approximates the solutions and the hatched box Θ over-approximates the inconsistencies.

not be solutions. We will refer to these instantiations as inconsistent instantiations. By elimination, the rest of the search space can only contain solutions.

In the remainder of the subsection, given a constraint C and a box $D = d_1 \times \cdots \times d_n$, we will write D_C the set of instantiations of D that satisfy C and $\overline{D_C}$ and, the — complementary — set of inconsistent instantiations w.r.t C. Thus, we have $\overline{D_C} = D \setminus D_C$. As D_C, $\overline{D_C}$ can be uncomputable, so we compute an over-approximation. For a single constraint, this can be achieved simply by reusing the propagation, over the negation of the constraint.

Definition 2. *Let $x_1, \ldots, x_n$ be variables in domains $d_1, \ldots, d_n$, and C a constraint on $x_1, \ldots, x_n$. We define a function $\theta_C : \mathcal{B} \to \mathcal{B}$ such that $\theta_C(d_1, \ldots, d_n) = \rho_{\neg C}(d_1, \ldots, d_n)$.*

Combining this function with propagation, we partition D relatively to the satisfiability of C. Let $S_C = \rho_C(d_1, \ldots, d_n)$ and $\overline{S_C} = \theta_C(d_1, \ldots, d_n)$ be respectively the over-approximation of D_c and $\overline{D_C}$, we differentiate three kinds of instantiations:

- the ones that belong to $\overline{S_C}$ and not to S_C, which are inconsistent,
- the ones that belong to S_C, and not to $\overline{S_C}$, which are consistent,
- the remaining ones that belong to both S_c and $\overline{S_C}$, which are indeterminate.

Figure 2 shows an example of this partitioning. For the constraint $y \leq x^3$ (filled with blue), the box $\mathcal{S}_C$ (dashed), computed through propagation, over-approximates the solutions and the box Θ (hatched in green), computed by applying propagation over the negation of the constraint, over-approximates the inconsistencies. We can see that the complement of Θ under-approximates the set of solutions, while the complement of R under-approximates the set of inconsistencies. The intersection $\Theta \cap R$ can contain both solutions and inconsistencies.

Once this partitioning is done, the inconsistent part can be discarded (as usual) and the consistent one can be directly added to the set `sols` of solutions. What remains is the indeterminate space in which the solving process continues. This principle is then generalized to the case of several constraints: the consistent part is the intersection of all the consistent parts associated to each constraint. Symmetrically, the inconsistent part is the union of all the inconsistent parts associated to each constraint. What remains is the indeterminate part.

Remark In practice, in the case of continuous constraints, elimination can rely on the original propagation algorithms of the considered constraint, since we can easily compute the negation of a constraint (based on predicates $<, =, \leq$). It would also be valid for discrete constraints *provided that* the same property holds. Indeed, primitive constraints could be dealt with elimination, but handling global constraints would require to specifically define their negations and introduce dedicated propagators.

The indeterminate space is defined as an intersection of boxes, which results in a box. Hence, the solving process continues within a box, as in a classic propagation-based solver, except that the box is possibly smaller as we intersect the result of propagation with the result of elimination. However, $S_C \setminus \overline{S_C}$ is not necessarily a box. Computing this set difference requires taking the complement of a box relative to another box. In the following section, we define a set difference operator over boxes. It computes the difference as a set of boxes, that can be directly added to `sols`.

3.2 Difference operator

Given two boxes B_1 and B_2, their difference $B_1 \setminus B_2$ is not necessarily a box. However, we can express it as a collection of boxes that covers $B_1 \setminus B_2$. To guarantee a non-redundancy property over the result, this cover should be a partition. This would prevent boxes from overlapping and have instantiations covered by several boxes. However, a cover is sufficient to have a sound and complete resolution method, and is easier to build as we will see in the current section. Our difference operator should satisfy the following properties:

Definition 3 (Difference operator). *A difference operator* $\ominus : \mathcal{B} \times \mathcal{B} \to \wp(\mathcal{B})$ *is a binary operator such that* $\forall B_1, B_2 \in \mathcal{B}$*:*
(1) $|B_1 \ominus B_2|$ *is finite;*
(2) $\forall b \in (B_1 \ominus B_2) \Rightarrow b \cap B_2 = \emptyset$*;*
(3) $B_1 = (B_1 \cap B_2) \cup \bigcup \{\, b \in B_1 \ominus B_2 \,\}$.

The first condition ensures that the solving method produces a finite set of boxes. The second one ensures that the operator eliminates from the box B_1 the values inside the box B_2. Finally, the third condition guarantees that the difference of B_1 and B_2, union B_2, covers the initial box B_1. The second condition is related to soundness and the third one to completeness.

Our difference operator on boxes works with constraints. A box can be defined as a conjunction of constraints $B = \bigwedge_{i=1,\ldots,p} c_i$, where each constraint $c_i = \pm x_i \triangleleft a_i$, with $\triangleleft \in \{<, \leq\}$, gives a lower or an upper bound — not necessarily included — on x_i.

Note that it is mandatory to be able to express both strict and large inequalities. Otherwise, a problem would arise as the negation of $\pm x_i > a_i$ would not be exactly representable, and we would have no way to ensure property Def. 3.2. As the difference operator is used to compute $\mathcal{S}$, an under-approximation of the set of solutions, adding to $\mathcal{S}$ the closure of boxes which should actually be open, could add to it points that are not solutions to the problem, and thus break the soundness criterion.

Each c_i defines a half-space, and the difference between a box and a half-space is still a box. A first step is thus to compute the difference between two boxes, by considering each half space independently, as shown on Fig. 3(b).

Definition 4 (Difference for boxes). *Let B_1 and B_2 be two boxes, with B_2 represented as the set of constraints C_2. The difference of B_1 and B_2 is:*

$$B_1 \ominus B_2 \triangleq \{B_1 \cap (\neg c) \mid c \in C_2\} \tag{1}$$

This naive method can result in widely overlapping boxes in the output. Nevertheless, it is an acceptable difference operator as it satisfies Def. 3:

(1) $B_1 \ominus B_2$ returns a set that, associates a box to each constraint in B_2. The number of constraints in B_2 is finite, hence this set is finite (Def. 3.(1)).
(2) By definition of the intersection, the condition Def. 3.(2) is satisfied as each box in the result is included in B_1.
(3) Finally, Def. 3.(3) is also satisfied: $B_1 \ominus B_2$ can be rewritten as $B_1 \cap \overline{B_2}$. No solution is lost as B_1 is entirely covered by B_2 and $B_1 \ominus B_2$.

Figure 3 shows an example of the application of the difference operator on two boxes. Figure 3(a) gives the initial boxes B_1 and B_2, with B_2 represented by the constraints $\{c_1, \ldots, c_4\}$. Figure 3(b) shows the result of the naive difference operator. Here, $B_1 \setminus B_2$ is covered by three elements, one per constraint of C_2, after removing the constraints that, intersected with B_1, yield the empty set (c_4 in this case). Overlapping boxes in the output appear in a darker shades. This overlapping implies that some instantiations may be covered by more than one box: the result is redundant.

We now propose an improved difference operator in order to obtain non-overlapping boxes when building a partition of $B_1 \setminus B_2$.

Definition 5 (Non-redundant difference for boxes). *Let B_1 and B_2 be two boxes and B_2 is represented by the set of constraints $C_2 = \{c_1, \ldots, c_p\}$. The difference of B_1 and B_2 is defined as:*

$$B_1 \ominus B_2 \triangleq \left\{ B_1 \cap (\neg c_i) \cap \bigcap_{j<i} c_j \mid i \in \{1, \ldots, p\} \right\} \tag{2}$$

For similar reasons to the naive difference operator, Def. 3.(1)–(3) is also satisfied for the non-redundant difference operator. Additionally, we strengthen the property that $B_1 \ominus B_2$ is a cover for $B_1 \setminus B_2$ by making this cover a partition, i.e, the elements of $B_1 \ominus B_2$ are pairwise disjoints: we ensure that, for any pair of boxes $b_i, b_j \in B_1 \ominus B_2$ such that $i \neq j$, we have $b_i \cap b_j = \emptyset$.

Proposition 1. *$B_1 \ominus B_2$ is a partition of $B_1 \setminus B_2$.*

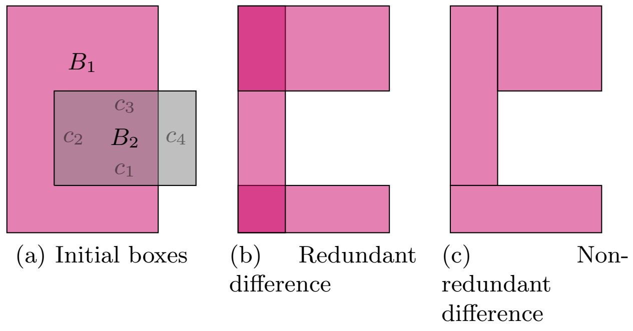


Fig. 3: Comparing naive and non-redundant difference operator: $B_1 \ominus B_2$.

Proof. If $|B_1 \ominus B_2| = 1$ then, trivially, $B_1 \ominus B_2$ is a partition of $B_1 \setminus B_2$. If $|B_1 \ominus B_2| > 1$, we have to prove that the elements of $B_1 \ominus B_2$ are pairwise disjoints. Let $C_2 = \{c_1, ..., c_p\}$ be the constraints of B_2, and b_i, b_j be respectively the i-th and the j-th value of $B_1 \ominus B_2$ according to (2), with $i, j \in 1..p$ and $i \neq j$. Then, b_i is constrained by $\neg c_i$. Assuming w.l.o.g. that $i < j$, then b_j is constrained by c_i, and $b_i \cap b_j = \emptyset$. We also have to prove that $B_1 = B_1 \setminus B_2 \cup B_2$, or equivalently, $\cup_i b_i = B_1 \setminus B_2$: let $x \in \cup_i b_i$ be an instanciation of B_1. By definition of B_2, there is at least a constraint $c_i \in C_2$ such that x does not satisfy. Let i_0 be the smallest such i, then $x \in b_{i_0}$. Thus, the whole of $B_1 \setminus B_2$ is covered by the boxes b_i.

Fig. 3(c) shows the result of the non-redundant difference operator and shows that there are no overlapping darker zones (shown in a darker shade). Here, $B_1 \setminus B_2$ is now partitioned into three elements, one per constraint of C_2 (once again ignoring c_4 which leads to an empty box).

3.3 New solving step

Computing $\widetilde{S} = \theta_C(d_1, \ldots, d_n) \cap \rho_C(d_1, \ldots, d_n)$ by employing both propagation and elimination reduces the search space, because it allows the solver to quickly identify parts of the solutions. In fact, when the propagation of ρ_C is done, we propose an elimination step θ_C before splitting. Rather than performing arbitrary splits anywhere on a box, the elimination identifies parts of the box containing only solutions, and allows the solver to perform splits on the part of the search space that can not be discriminated as containing only solutions, nor as containing no solution. More precisely, elimination makes the split happen exactly at the frontier of the constraint.

Algorithm 2 gives the pseudo-code associated with the new elimination step. This algorithm processes elements that do not satisfy at least one constraint. The function `complement` computes $e_{non-cons}$, an over-approximation of the inconsistencies. Then, the difference operator is used to find the boxes containing only solutions. Finally, solving continues in the indeterminate search space $e \cap e_{non-cons}$ (instead of e).

Algorithm 2 Elimination function

```
1: function ELIMINATION($e, \mathcal{C}$)                ▷ $e$: box, $\mathcal{C}$: constraints
2:     $e_{non-cons} \leftarrow$ complement($e, \mathcal{C}$)
3:     $e_{cons} \leftarrow e \ominus e_{non-cons}$
4:     $S \leftarrow \emptyset$
5:     for $e_i \in e_{cons}$ do
6:         $S \leftarrow S \cup e_i$
       return ($S, \oplus(e \cap e_{non-cons})$)
```

Figure 1(b) shows the results obtained with our propagation/elimination/split loop on the CSP given previously, and gives for the same precision, much more satisfactory results: we require less elements to cover more space and in a comparable amount of time, showing that this technique deduces more relevant frontier than using a simple propagation/split loop.

In the following section, we analyze the performance of our solving method.

4 Experiments

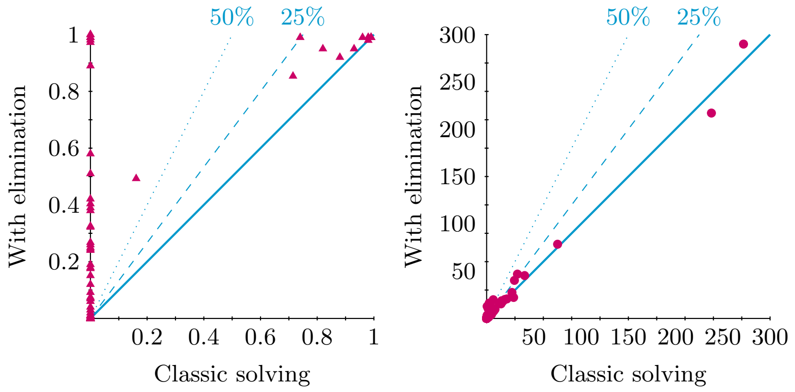
We have implemented our technique for boxes in the open-source solver AbSolute[4]. This solver is based on the method presented in [12], where we integrated our elimination step. We rely on the abstract domain representation in AbSolute, which is based on constraints, to efficiently implement the constraint negation necessary for the elimination step. The unified constraint representation makes it possible to have a lightweight and generic difference operator.

4.1 Protocol

We tested our method on problems with continuous variables from the MinLPLib and the Coconut[5] benchmarks. For minimization problems, we first transform them into satisfaction problems, which can be handled by the solver. This transformation consists in adding an objective variable to the problem that will act as the value to minimize. Default bounds for unconstrained variables are set to -10^7 for the lower bound and 10^7 for the upper bound as our method requires the domains of the variables to be bounded. All of the runs are made with a time limit set to 300 seconds and no memory limit. Precision was fixed to 10^{-3} (i.e., the size limit where exploration stops), and branching depth was limited by 50. Note that limiting depth does not break soundness nor completeness as our algorithm can be tailored to be produce either a complete or a sound output after the same run: its output can be splitted into two sets: the boxes that contain only solutions and the undetermined ones. At any point of the resolution, the union

[4] `https://github.com/mpelleau/AbSolute`

[5] All informations about the problems can be found at `http://www.gamsworld.org/minlp/minlplib/minlpstat.htm` and `http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html`



(a) Comparison of ratio (b) Comparison of time

Fig. 4: Comparison between the classic solving method and our method. On the left, comparison of the ratio, a mark above the bisector (in plain blue) means that our method is better than the classic solving. On the right, comparison of the computation time, a mark above the bisector (in plain blue) means that our method is slower than the classic solving.

of these two sets yield a complete solution, while taking into account only the first gives a sound solution. Thus, stopping the search at a given depth, makes the resolution faster, yet less precise, but does not break neither soundness nor completeness. The solver was run on a Dell server with two 12-core Intel Xeon E5-2650 CPU at 2.20GHz, although only one core was used, and 128GB RAM.

We have tested the solving with the elimination step against the default solving method of the AbSolute solver over all of the problems that the solver's functionalities (types, constraint, arithmetic functions) are able to cover, that is 197 problems.

4.2 Description

Figure 4 summarizes the results obtained with our method compared to the classic solving. Figure 4(a) compares the ratio δ of inner volume of the cover. It corresponds to $V_i/(V_i + V_e)$ where V_i and V_e are respectively the inner and outer volume. This ratio is a quality measure of the solving method: the closer this ratio is to one, the bigger is the part of the coverage that will only contain solutions. In this figure, a mark above the bisector means that our method is better than the classic one. We can see that on most of the instances, our method finds a coverage with a much smaller indeterminate space.

Figure 4(b) compares the computation time of our method to the classic one. In this figure, each mark above the bisector means that our method is slower than the classic one. As can be seen on this figure, our method is slightly slower, the elimination performing additional computation during one iteration. However, solving all the 197 problems took 1157 seconds with our method against 1032 seconds with the classic solving method.

		with elimination				without elimination			
problem	$\|\mathcal{X}\|, \|\mathcal{C}\|$	#I	#E	δ	t	#I	#E	δ	t
Coconut problems									
abs1	1,2	2047	3072	0.99	**0.04**	4092	4096	0.99	0.06
aljazzaf	2,3	2309	19405	**0.58**	0.89	0	14319	0	**0.54**
allinitu	1,5	318	5066	**0.07**	3.26	0	5066	0	**2.50**
b	4,4	2	88	**0.39**	0.07	0	88	0.00	0.07
booth	1,2	90	45	**0.12**	**0.11**	0	45	0	0.13
bqp	1,1	4	1	0.99	0.01	8	1	0.99	0.01
chi	1,2	1.88e6	3.48e6	0.99	45.30	3.08e6	3.63e6	0.99	**40.20**
ex1411	2,5	1.78e6	2.59e6	0.98	**217.08**	1.95e6	3.74e6	0.98	237.89
ex1413	4,3	4884	34893	**0.25**	**0.52**	0	32698	0	0.78
ex_newton	2,5	638	950	**0.95**	**0.45**	729	892	0.93	0.57
griewank	1,2	19972	31868	**0.99**	**1.44**	29645	35105	0.98	2.30
h76	3,4	24	174	**0.04**	0.05	0	82	0	0.05
hs23	6,2	825	2132	**0.99**	**0.43**	1315	1801	0.98	0.58
kear11	8,8	0	844	0	0.05	0	844	0	0.05
ladders	13,7	4	93	**0.01**	**0.90**	0	215	0.00	1.07
mickey	2,5	4315	12709	0.99	**2.40**	8372	9858	0.99	2.73
nonlin1	2,3	1550	1978	**0.95**	**0.49**	2059	1772	0.82	0.69
nonlin2	3,2	4238	10560	**0.92**	**0.39**	8643	10692	0.88	0.42
zy2	3,3	6260	28147	**0.99**	1.00	13179	22499	0.74	**0.85**
MinLPLib problems									
csched1a	23,29	0	8192	0	6.44	0	8192	0	**4.85**
deb10	130,183	0	0	0	0.01	0	0	0	0.01
dosemin2d	119,166	0	0	0	0.181	0	0	0	**0.177**
ex1222	4,4	8	60927	**0.01**	1.39	0	61787	0	**0.97**
ex1223a	10,8	746	27097	**0.01**	**20.60**	0	48283	0	21.40
ex1223b	10,8	820	44084	**0.01**	40.22	0	500510	0	**29.41**
gbd	5,5	576	31829	**0.19**	**1.27**	0	22927	0	0.93
prob03	2,3	0	5.81e6	0	10.81	0	5.81e6	0	**6.77**
qapw	256,451	0	0	0	**1.20**	0	0	0	1.23
st_e13	4,3	378	3102	**0.02**	0.05	0	18	0	**0.50**
st_miqp2	4,5	1352	38104	**0.38**	2.29	0	4564	0	**0.31**
st_miqp3	2,3	27	1117	**0.24**	0.03	0	1051	0	**0.02**
st_miqp5	14,8	187	2080	**0.01**	4.71	0	6324	0	**2.38**
st_test1	2,6	1559	2.44e6	**0.03**	18.70	0	2.31e6	0	**15.97**
st_test5	12,11	22	29520	**0.01**	7.82	0	11167	0.00	**7.55**
synthes1	7,7	97	33747	**0.01**	**4.18**	0	1285	0	5.16
tls2	25,38	0	18030	0	27.46	0	18030	0	**26.81**
windfac	14,15	0	19561	0	6.66	0	19561	0	**6.51**

Table 1: Comparing solving with and without elimination step

For reasons of space, Table 1 highlights only some of the results representative of the behavior of our method. Those are described with respect to solving times (in seconds), cardinality of the partition and volume covered. We performed experiments on the whole benchmarks (Coconut and MinLPLib), but we do not show here the problems which time out for both methods.

The first two columns provide information about the problem: name, number of variables $|\mathcal{X}|$, and number of constraints $|\mathcal{C}|$. The rest of the table provides information on each solving methods: the number of inner (columns $\#I$) and outer (columns $\#E$) boxes.

4.3 Analysis

These runs highlight one very crucial feature of our method: it is able to quickly find boxes that contain only solutions of problems where the default solving method fails to do so (problems *aljazzaf*, *allintu*, *ex1222*, *gbd*, ...): on the whole benchmark, for almost 30% of the problems (58 out of 197), solving with the elimination step exhibited at least one solution while the default solving method did not succeed to do so. This comes with no time less in average: on the whole benchmark, solving with elimination was slightly slower than without (1157 seconds against 1032 seconds). In fact, 39% of the problems (39 out of 197) were solved faster with the elimination than without (problems *ex1411*, *mickey*, *ex1223a*, *synthes1*...). This illustrates the fact that results of the solver are more precise: elimination avoids unnecessary splits, better identifies the constraints' frontiers, and compute within the same process inner and outer approximations for no (or little) overhead. A deeper analysis of the results shows that the default solving method spends time splitting variables with large ranges, while elimination focuses on the shape of the constraints to locate areas than can be directly removed from the search space and added to the solution set.

Another conclusion of the analysis of this benchmark is about the solution coverage. The experiments show that the coverage of the solution space is significantly more accurate with the elimination step. On all of the runs, our method always finds a greater or equal inner volume than the one found by the default method. Moreover, it also reduces the number of elements involved in the partition in the same time, which means that the inner approximation is achieved with fewer, bigger elements. This is shown by examples *chi* and *mickey* where both methods achieve a 0.99 ratio of inner volume, only with elimination, we need half the elements required by the default solving method to do so. On the whole benchmark, on average, we need 40 times fewer elements to cover the same inner volume with elimination. This property may become very handy as it allows a better re-usability of the results since we need to treat fewer elements to cover the solution space. The δ columns indicates the part of the returned elements that corresponds to an inner approximation, i.e. contains only solutions. This ratio is always greater with the elimination step. On the whole benchmark, the average ratio is 0.49 of inner volume for the elimination while it is 0.27 without. This confirms that the elimination step allows the solving process to target more efficiently the parts of the search space that contain only solutions.

These good results confirm the intuition that cutting an element according to the constraints it does not satisfy can be more interesting than cutting it arbitrarily regardless of the constraints. Since solvers are often used as a precomputation for other programs, reducing the size of their output (i.e., reducing the number of boxes required to represent a solution at a given precision) can be an important feature. Also, note that, by quickly identifying solutions and removing them from the search space, the elimination step makes it possible to carry out fewer propagation and exploration steps.

5 Conclusion

In classic continuous constraint solvers, propagation is used to remove from the search space values that can not be solutions. We presented in this paper a new method to, symmetrically, eliminate from the search space values that can only be solutions. We have incorporated the elimination mechanism to improve the results in terms of a qualitative and quantitative criterion, also without a too large time overhead. This technique, which delays a splitting heuristic that can be inaccurate, makes it possible to take better advantage of the constraints of a problem, by reusing and adapting the same tools as propagation, combined with a difference operator we have introduced. Finally, it should be emphasized that, although it is implemented in a specific solver using abstract domains, this technique can perfectly be integrated into a more classic solver and combined with any type of propagator.

We believe that this resolution technique can be useful in many cases. For problems or zones of non-consistent instantiations forming "holes" in the solution space, or more generally, when it is non-convex, it can avoid several cutting steps by directly targeting the most relevant boundaries. This property may be particularly interesting in the context of inner-approximation applications, as shown by the experiments, or counter-example exhibition (feasibility proving) when it comes to find at least one solution as our method outperforms the default solving method in that competence.

Further research includes the development of elimination beyond boxes, for instance on polyhedra which can also be defined as a conjunction of constraints, making it possible to add a difference operator. It would also be interesting to measure the performance of this technique with other consistency and splitting heuristics.

References

1. Heikel Batnini, Claude Michel, and Michel Rueher. Mind the gaps: A new splitting strategy for consistency techniques. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2005.

2. Frédéric Benhamou. Heterogeneous constraint solvings. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, pages 62–76, 1996.
3. Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revisiting hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming*, pages 230–244, 1999.
4. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI'2004)*, pages 146–150. IOS Press, 2004.
5. Michael R. Bussieck, Arne Stolbjerg Drud, and Alexander Meeraus. Minlplib - A collection of test models for mixed-integer nonlinear programming. *INFORMS Journal on Computing*, 15(1):114–119, 2003.
6. Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173:1079–1100, 2009.
7. Hélène Collavizza, François Delobel, and Michel Rueher. Extending consistent domains of numeric CSP. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 406–413, 1999.
8. Eldon Hansen. *Global optimization using interval analysis.* Marcel Dekker, 1992.
9. Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial intelligence (IJCAI'79)*, pages 356–364. Morgan Kaufmann Publishers Inc., 1979.
10. Luc Jaulin and Eric Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993.
11. Ramon Edgar Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs N. J., 1966.
12. Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, 2013.
13. Dietmar Ratz. Box-splitting strategies for the interval Gauss-Seidel step in a global optimization method. *Computing*, 53:337–354, 1994.
14. Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming.* Elsevier Science Inc., New York, NY, USA, 2006.
15. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.
16. Oleg Shcherbina, Arnold Neumaier, Djamila Sam-Haroud, Xuan-Ha Vu, and Tuan-Viet Nguyen. Benchmarking global optimization and constraint satisfaction codes. In *Global Optimization and Constraint Satisfaction, First International Workshop Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Valbonne-Sophia Antipolis, France, October 2-4, 2002, Revised Selected Papers*, pages 211–222, 2002.