Sweeping with Continuous Domains

Gilles Chabert and Nicolas Beldiceanu

École des Mines de Nantes LINA CNRS UMR 6241,
4, rue Alfred Kastler 44300 Nantes, France
(gilles.chabert|nicolas.beldiceanu)@mines-nantes.fr

Abstract. The `geost` constraint has been proposed to model and solve discrete placement problems involving multi-dimensional boxes (packing in space and time). The filtering technique is based on a sweeping algorithm that requires the ability for each constraint to compute a forbidden box around a given fixed point and within a surrounding area. Several cases have been studied so far, including integer linear inequalities. Motivated by the placement of objects with curved shapes, this paper shows how to implement this service for continuous constraints with arbitrary mathematical expressions. The approach relies on symbolic processing and defines a new interval arithmetic.

1 Introduction

Sweeping [4] is a generic technique for filtering with constraints, like propagation for instance. This technique has been fruitfully applied in the context of placement problems (rectangle packing, container loading, warehouse management). In particular, the filtering algorithm of the `geost` constraint [5, 8, 6], today implemented in different systems like `Choco` [1], `JaCop` [10] or `SICStus` [9], is based on a sweeping loop.

Propagation (e.g., `AC3` [11]) requires a *propagator* for each constraint, that is, an operator that removes inconsistent values. Symmetrically, we will see that sweeping requires an *inflater* for each constraint, that is, an operator that returns a set of unfeasible tuples.

Propagators can be built simply by checking consistency for each value. Likewise, inflaters can be built by enumerating values inside the cross product of domains and checking for inconsistency.

However, such brute force methods, apart from being inefficient, are not possible with continuous domains (that are not countable). In previous publications, inflaters have been proposed for several important classes of constraints: rectangle inclusion and non-overlapping [5], linear equations, distance equations [2] as well as constraints derived from business rules [8]. The method in [8] is generic as it addresses a class of first-order formulae with linear constraints by way of predicates. However, all these methods are restricted to discrete domains and only propose ad-hoc solutions for (a few) nonlinear constraints.

The main contribution of this paper is a generic inflater that works for *any* constraint on continuous domains, as long as the constraint has a mathematical

expression made of usual operators ($+,\times,-,/$) and functions (sqr, sqrt, sin, etc.). The complexity of this inflater is linear in the length of the constraint expression and optimal (i.e., the corresponding box cannot be extended any more in any dimension) if variables have no multiple occurrences.

Hence, thanks to this new algorithm, the applicability of `geost` can be extended to a significantly larger class of placement problems: nonlinear inequalities can indeed model, e.g., the non-overlapping of objects with curved shapes (balls, cylinders, etc.) in two or three dimensions.

The paper is organized as follows. In Section 2, the sweeping algorithm is recalled, but in a slightly revised form that makes it independent of the nature of domains (discrete or continuous). In Section 3, a generic inflater is described.

2 Sweeping

Since this paper is dedicated to continuous domains, we will need notations to represent and handle intervals of reals. Let us state them now.

Notations. Vectors will be represented by bold face letters. Intervals and boxes (cross products of intervals) will be surrounded by brackets, e.g., $[x]$ and $[\mathbf{x}]$. The symbol $\mathbb{IR}$ represents the set of all intervals. If $[x]$ is an interval, $\underline{x}$ and $\overline{x}$ will stand for the lower and upper bound respectively of $[x]$. If $[\mathbf{x}]$ is a box, the i^{th} component of $[\mathbf{x}]$ is an interval $[x]_i$ (or $[x_i]$) and the bounds of $[x]_i$ will be denoted by $\underline{x}_i$ and $\overline{x}_i$.

2.1 From Propagation to Sweeping

Consider first a classical propagation of constraints. We are comparing here inference methods (not particular algorithms), but to fix idea one may think about `AC3`. The principle of propagation is the same for continuous domains as for discrete ones (except that only bounds are usually reduced for practical reasons). In the situation depicted in Figure 1.(a), each constraint is able, separately, to filter the domain of a variable. This results in "strips" removed from the domain $[\mathbf{x}] = [x]_1 \times [x]_2$. For instance, when c_1 reduces the left bound of x_1 to $\tilde{x}_1$, all the tuples in the hatched strip are proven to be inconsistent.

In the situation of Figure 1.(b), propagation is blocked because all the bounds ($\underline{x}_1$, $\overline{x}_1$, $\underline{x}_2$ and $\overline{x}_2$) are consistent (have supports) w.r.t. each constraint. However, let us now assume that, by some means, one can build *forbidden boxes* in the cross domain $[x] \times [y]$, according to the following definition:

Definition 1 (Forbidden box). *Given a constraint c over $\mathbb{R}^n$, a box $[\mathbf{x}] \subset \mathbb{R}^n$ is called* forbidden *w.r.t. c if no vector in $[\mathbf{x}]$ satisfies c.*

A first forbidden box $[\mathbf{f}]$, w.r.t. c_1, can be placed in the lower left corner of $[\mathbf{x}]$ (see Figure 1.(b)). A second forbidden box $[\mathbf{f}']$ can then be built above the first one, but by considering this time c_2. By considering again c_1 a third box $[\mathbf{f}'']$ can be

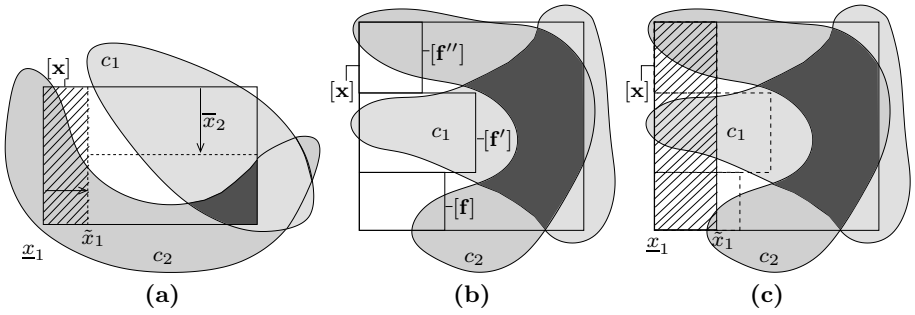

Fig. 1: **Comparing propagation and sweeping**. The outer rectangle represents the initial box $[\mathbf{x}]$. Feasible points of constraints are represented in light gray. The set of solutions in $[\mathbf{x}]$ is in dark gray. **(a).** Propagation can be applied: the left bound of x_1 is reduced by the constraint c_1 (hatched strip). The upper bound of x_2 is then reduced by c_2. Note that, at this stage, the fixpoint is not reached yet and pruning could be carried on. **(b).** In this situation, nothing happens with propagation: the bounds of $[\mathbf{x}]$ are consistent w.r.t. c_1 and c_2 (taken separately). However a forbidden box can be built w.r.t. each constraint. **(c)** The union of the forbidden boxes allows to prune x_1.

placed again on top of $[\mathbf{f}']$. The union of these boxes covers a strip $[\underline{x}_1, \tilde{x}_1] \times [x]_2$ (see Figure 1.(c)) that can be removed, just as in classical propagation. Note that $[\underline{x}_1, \tilde{x}_1]$ is $[f]_1 \cap [f']_1 \cap [f'']_1$, that is, $\tilde{x}_1$ is the smallest upper bound of the x_1's, among the forbidden boxes.

There is a duality between propagation and sweeping. Propagation requires each *individual* constraint to provide a *global* filtering (a full strip) while sweeping aggregates the result of *several* constraints, each only providing a *partial* filtering (a forbidden box). The advantage of propagation is clear: a global filtering is an information that can be directly shared by other constraints. But global filtering is clearly difficult to obtained, in general, from a single constraint (especially when related variables have large domains). This is typically observed in placement problems as an empirical fact, even when some objects are initially fixed.

On the contrary, a forbidden box is easier to obtain but does not help the other constraints directly. Computations of forbidden boxes are independent from one constraint to the other. The key idea behind sweeping is to guide these computations so that piling up forbidden boxes maximizes the chance to perform a global filtering quickly. This relies on the concept of *working area*.

2.2 The Working Area

In this section, we shall first consider 2 and then $n > 2$ variables. For the sake of clarity and w.l.o.g., pruning is always assumed to be performed on the left bound of the first variable, x_1. The other cases are obtained symmetrically, by reordering variables and reversing some inequalities.

A strip like $[\underline{x}_1, \tilde{x}_1] \times [x]_2$ in Figure 1.(c) is formed when the union of the projections of forbidden boxes over x_2 entirely cover $[x]_2$. We say, in this case, that x_2 is *saturated*. The property each $x_1 \in [\underline{x}_1, \tilde{x}_1]$ satisfies is then $\forall x_2 \in [x]_2$, (x_1, x_2) is not a solution, which indeed means that $[\underline{x}_1, \tilde{x}_1]$ is inconsistent. Forbidden boxes have to be built within a limited space, bounded below and above by *anchors*:

1. A forbidden box has to be placed on a precise location. For instance, while x_2 is not saturated, the next forbidden box must be placed on the lower left corner above the last one. This location (the lower left corner) defines a *left anchor* for x_1 and x_2.
2. As said above, $\tilde{x}_1$ is the lowest upper bound of the x_1's among the forbidden boxes used to saturate x_2. This bound can be initialized to $\overline{x}_1$ and updated incrementally along the process, until saturation of x_2 is achieved. Then, the bound is reset to $\overline{x}_1$, for the next strip. This bound defines a *right anchor* for x_1. The right anchor for x_2 can simply be set to $\overline{x}_2$ since x_2 is always the first dimension to saturate (in our considered situation).

We call *working area* and denote by $[\mathbf{X}]$ the box delimited by the anchors. It is clear that for a forbidden box $[\mathbf{f}]$, all the part that exceeds the working area is superfluous. Consider for instance Figure 1.(b), once $[\mathbf{f}]$ is calculated. The right anchor for x_1 becomes $\overline{f}_1$. Then, all the points $\mathbf{x}$ of $[\mathbf{f}']$ with $x_1 > \overline{f}_1$ are useless and, indeed, they won't belong to the final strip. All this extends to n variables thanks to a n-dimensional working area, as shown in Figure 2 for $n = 3$.

For all j, $1 \leq j \leq n$, sweeping over x_j (potentially up to saturation) is recursively based on the saturation of x_{j+1}. When x_{j+1} is saturated, the $(n-j)$-dimensional face $[x]_{j+1} \times \cdots \times [x]_n$ is covered by the projection of forbidden boxes. It is then proven that $[X]_1 \times \cdots \times [X]_j$ is inconsistent. Geometrically, the following box:

$$\Big([X]_1 \times \cdots \times [X]_j\Big) \times \Big([x]_{j+1} \times \cdots \times [x]_n\Big) \tag{1}$$

forms a "bar", that generalizes the "strip" in 2D above. When $j = 1$, the interval $[X]_1$ can be definitely pruned. The working area is initialized to the whole box $[\mathbf{x}]$ and maintained in two steps each time a new box is calculated. First step: for all j, $\overline{X}_j$ is updated in order to be the minimum of $\overline{f}_j$ among the forbidden boxes $[\mathbf{f}]$ obtained since the last saturation of x_{j+1}. At this point, inconsistency of (1) holds. Second step: if x_{j+1} is saturated, the area is extended to all the points necessary to saturate x_j: $[X]_j \leftarrow [\overline{X}_j, \overline{x}_j]$ and $[X]_i \leftarrow [x]_i$ for all $i > j$.

Hence, boxes are calculated in order to saturate x_n, x_{n-1}, ... downto x_1, each time from bottom to up. These choices are clearly arbitrary (except for x_1 that must be the last saturated variable if one is to prune this variable) and we could proceed in other ways. This means that tuples have to be ordered. The natural order we chose is the lexicographic one.

2.3 Inflaters

Let us now focus on the way forbidden boxes are built. For reasons that will be soon apparent, we consider that forbidden boxes are the result of a new kind

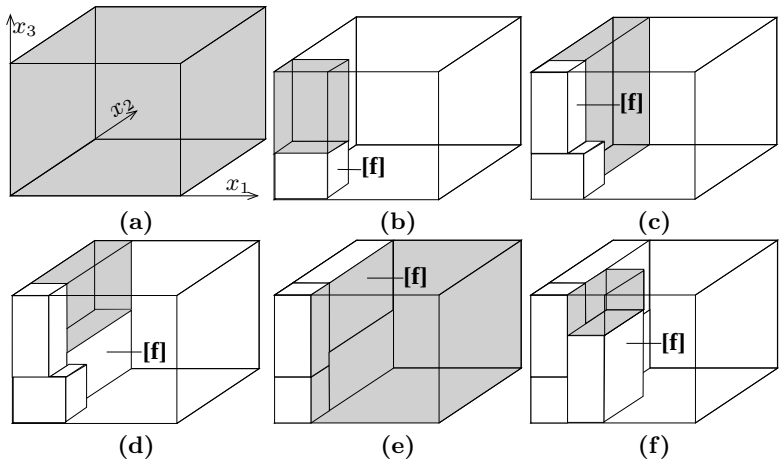

Fig. 2: **Evolution of the working area.** The forbidden boxes are in white and the working area in transparent (light gray). **(a)** At the beginning, $[\mathbf{X}] \leftarrow [\mathbf{x}]$. (**b**-1) $\overline{X}_1 \leftarrow \overline{f}_1$, $\overline{X}_2 \leftarrow \overline{f}_2$, $\overline{X}_3 \leftarrow \overline{f}_3$. (**b**-2) Since x_3 is the last dimension: $[X]_3 \leftarrow [\overline{X}_3, \overline{x}_3]$. (**c**-1) $\overline{X}_1 \leftarrow \overline{f}_1$ ($\overline{X}_2$ and $\overline{X}_3$ are not impacted: $\overline{f}_2 = \overline{X}_2$ and $\overline{f}_3 = \overline{X}_3$). (**c**-2) x_3 is saturated so $[X]_2 \leftarrow [\overline{X}_2, \overline{x}_2]$ and $[X]_3 \leftarrow [x]_3$. (**d**-1) only X_3 is impacted: $\overline{X}_3 \leftarrow \overline{f}_3$. (**d**-2) $[X]_3 \leftarrow [\overline{X}_3, \overline{x}_3]$. (**e**-1) Nothing is impacted. (**e**-2) Since x_2 is saturated, $[X]_1 \leftarrow [\overline{X}_1, \overline{x}_1]$, $[X]_2 \leftarrow [x]_2$ and $[X]_3 \leftarrow [x]_3$. (**f**-1) $\overline{X}_1 \leftarrow \overline{f}_1$, $\overline{X}_2 \leftarrow \overline{f}_2$, $\overline{X}_3 \leftarrow \overline{f}_3$. (**f**-2) $[X]_3 \leftarrow [\overline{X}_3, \overline{x}_3]$.

of operator, called *inflater*. The basic idea behind an *inflater* is to take a single forbidden point $\tilde{\mathbf{x}}$ and to build a forbidden (and as large as possible) box around $\tilde{\mathbf{x}}$ inside a working area. However, it is often possible to inflate in different (and incomparable) ways as Figure 3 suggests.

Boxes that are not well-balanced slow down the sweeping process. To help the intuition, imagine that a degenerated interval (reduced to a single point) appears on a dimension. The working area is then flattened on this dimension until the subsequent dimensions are all saturated. All this work is done for no gain at all.

With a very basic geometric argument, we can say that the most extended a box is in a given direction, the less it is in another. As a consequence, taking into account the working area at the source of the inflation is important. This explains why, in the next definition, an inflater takes an initial forbidden point $\tilde{\mathbf{x}}$ **and** a working area $[\mathbf{X}]$ (a bad alternative would be to calculate a forbidden box unboundedly and intersect the result with the working area).

Definition 2 (Inflater). *Let $\mathcal{C}$ be a constraint over a set of n variables, i.e., a subset of $\mathbb{R}^n$. An inflater I of $\mathcal{C}$ is an operator from $\mathbb{R}^n \times \mathbb{IR}^n \to \mathbb{IR}^n$ such that:*

$$\forall [\mathbf{X}] \in \mathbb{R}^n, \forall \tilde{\mathbf{x}} \in [\mathbf{X}], \quad \begin{cases} (i) \quad I(\tilde{\mathbf{x}}, [\mathbf{X}]) \subseteq [\mathbf{X}] \\ (ii) \quad I(\tilde{\mathbf{x}}, [\mathbf{X}]) \neq \emptyset \Longrightarrow \tilde{\mathbf{x}} \in I(\tilde{\mathbf{x}}, [\mathbf{X}]) \\ (iii) \; I(\tilde{\mathbf{x}}, [\mathbf{X}]) \cap \mathcal{C} = \emptyset \end{cases}$$

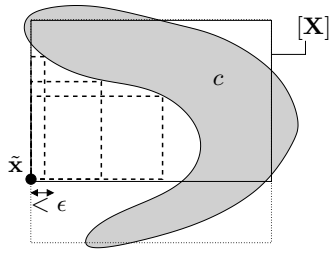

Fig. 3: **Non-unicity and quality of inflation.** Three different forbidden boxes (inside the same area $[\mathbf{X}]$ and w.r.t. the same constraint c) are represented with dashed contour. They are all optimal, i.e., they cannot be inflated more in any direction. The projection of a forbidden box can be wide on a dimension and very narrow on another one.

(i) means that the inflation is bounded by the working area. (ii) means that the anchor $\tilde{\mathbf{x}}$ belongs to the inflated box. (iii) means that the inflated box is forbidden.

Note that the inflater can return an empty set. This property ensures that the definition is consistent with the case where $\tilde{\mathbf{x}}$ is feasible w.r.t. a constraint.

2.4 Algorithm

We can give now our lexicographic sweeping algorithm for pruning the lower bound of a variable. This corresponds to a revision of the algorithm *pruneMin* given in [5] page 9. The algorithm for pruning the upper bound (*pruneMax*) is easy to obtain from the same algorithm by swapping bounds and inequality signs conveniently, it will be omitted here.

The procedure *try_inflate* tries to build a forbidden box containing $\tilde{\mathbf{x}}$ and inside $[\mathbf{X}]$, by considering each constraint of $\mathcal{C}$ in turn. The purpose of Section 3 is precisely to show how to inflate w.r.t. a given constraint. If the best forbidden box found $[\mathbf{f}]$ is significantly large, the procedure sets $[\mathbf{X}]$ to $[\mathbf{f}]$ and returns **true** (and **false** otherwise).

The changes we brought to the original discrete version are:

1. A working area is maintained instead of a *sweep point* and a *jump vector*, in the terminology of the papers cited above.
2. This working area is transmitted to the inflaters for the calculation of more well-balanced forbidden boxes (as justified in §2.3).
3. Since we are in a continuous setting, the lower left corner of $[\mathbf{X}]$ is not the lowest feasible vector but the greatest unfeasible one reached by the inflaters.
4. Forbidden boxes whose size is below a certain floor are discarded.

2.5 Tradeoff between Sweeping and Propagation

In presence of n variables, sweeping means dealing with a worst-case number of forbidden boxes that grows exponentially with n. Therefore, a tradeoff between

Algorithm 1: *pruneMin*$(\mathcal{C}, [\mathbf{x}], d)$

Input: a set of constraints $\mathcal{C}$, a k-dimensional box $[\mathbf{x}]$
Output: a subbox of $[\mathbf{x}]$ with the lower bound of the d^{th} coordinate pruned w.r.t. $\mathcal{C}$

```
1  b ← true;                                          // true while a solution may exist in [x]
2  success ← false;                                   // true if filtering occurs
3  [X] ← [x] ;                                        // init the working area
4  while b and try_inflate(C, X_, [X]) do             // see comment below
5      b ← false
6      for j = k − 1 downto 0 do                      // for each saturated dimension
7          j' ← (j + d) mod k
8          if X̄_{j'} < x̄_{j'} then                    // (j'+1) is the last saturated dimension
9              if j = 0 then success ← true;          // "global" filtering has occurred
10             [X]_{j'} ← [X̄_{j'}, x̄_{j'}];           // start a new "strip"
11             b ← true;                              // something is left inside [x]
12             break;                                 // exit the "for" loop (x_{j'} is not saturated)
13         [X]_{j'} ← [x]_{j'};                       // reset to the whole domain (saturated dimension)
14 if b then
15     [x'] ← [x] ;                                   // load the initial box
16     [x']_d ← [X_d, x̄_d];                           // apply the global filtering
17 else
18     [x'] ← ∅;                                      // raise unfeasibility
19 return [x']
```

sweeping and propagation is usually made. The number k of variables involved in a sweeping loop is typically limited to 2 or 3. The pruning obtained by sweeping is then propagated to the other groups of variables, in a classical way.

Of course, there must be some correlation between variables of a same group. Otherwise, sweeping has less effect: if we consider the extreme case of two independent variables, sweeping with these two variables amounts to filter with constraints separately, i.e., propagation.

We must therefore choose groups of k variables that are "correlated". But finding such mappings in a general setting is a problem on its own (see, e.g., [3]). This is the main reason why sweeping is well adapted to geometrical problems: we have in these problems a direct correlation between the variables that correspond to the coordinates of the same object. In other words, the dimensions of the sweeping loop match the geometrical dimensions.

3 A Generic Inflater for Arithmetical Constraints

Our inflater manipulates constraints in symbolic form. Inflation is made by an induction over the syntactical tree of the constraint expression. Symbolic processing is well known through modern computer algebra systems (like Maple or

Mathematica). But it has also led to significant results in constraint programming, for building propagators and calculating enclosures of derivatives, see [7]. Our algorithm needs to symbolically inverse elementary functions, as done by `HC4Revise` [7] for filtering a constraint. However, the inverted functions are evaluated with a new interval arithmetic (cf. Algorithms 2 and 3).

The syntactical tree of a mathematical expression is a composition of operators $\circ \in \{+, -, \times, /\}$ and elementary functions $\phi \in \{\text{sqr}, \text{sqrt}, \cos, \ldots\}$. A constraint can therefore be decomposed into a tree-structured network of *elementary* constraints $z = x \circ y$ and $y = \phi(x)$. For instance, consider the following distance constraint of arity 5:

$$(c) \quad x_5 = (x_1 - x_2)^2 + (x_3 - x_4)^2.$$

The network of elementary constraints equivalent to c is given in Figure 4.

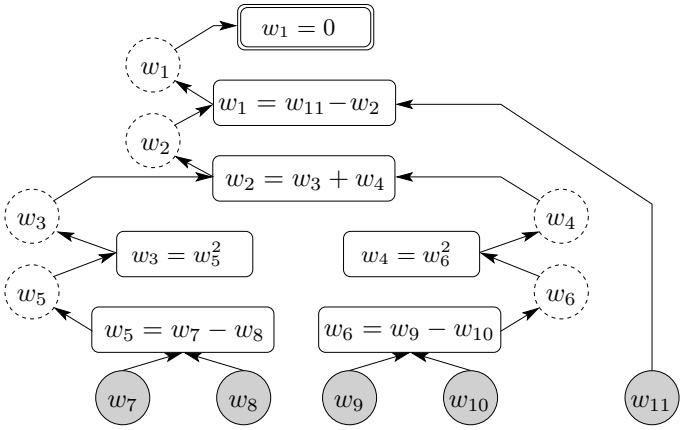

Fig. 4: **Decomposition of the constraint** (c) **into elementary constraints.** Variables $w_7, \ldots, w_{11}$ correspond to the real variables $x_1, \ldots, x_5$. The variables $w_1, \ldots, w_6$, are dummy variables, created on-the-fly upon activation of the constraint.

Let us call $dec(c)$ the CSP resulting from the decomposition of c, and k the number of dummy variables. Let us denote by $w_1, \ldots, w_{k+n}$ the variables of $dec(c)$ sorted increasingly with respect to the partial order induced by the tree, i.e., w_1 is the variable of the root constraint and $i < j$ if w_j is a variable that appears below w_i in the same branch. Note that Figure 4 respects this convention. Let us also call $output(j)$ the index i of w such that w_i is the output variable of the elementary constraint involving w_j as input (e.g., $output(9) = 6$, $output(6) = 4$ and $output(4) = 2$). Finally, we will consider the functions $f_1(x), \ldots, f_k(x)$ such that $w_i = f_i(x)$ is the constraint corresponding to the subtree below w_i. E.g., $f_5(x) = x_1 - x_2$, $f_4(x) = (x_3 - x_4)^2$.

The basic idea is to calculate a $(k+n)$-dimensional box

$$(-\infty,+\infty)^k \times [w_{k+1}] \times \cdots \times [w_{k+n}]$$

that is forbidden w.r.t. $dec(c)$. The forbidden box $[\mathbf{x}]$ is then simply obtained by projection on the n last dimensions:

$$[\mathbf{x}] \leftarrow [w_{k+1}] \times \cdots [w_{k+n}].$$

Indeed, if some $x \in [\mathbf{x}]$ would be a solution w.r.t. c, there would be an instantiation of the w_i's with $w_{k+1} = x_1, \ldots, w_{k+n} = x_n$ that would satisfy $dec(c)$. Such instantiation would necessarily belong to $[\mathbf{f}]$, giving a contraction.

Our algorithm in split into two phase. The *forward* phase propagates $\tilde{\mathbf{x}}$ (the initial forbidden point) and $[\mathbf{X}]$, that is, calculates a point $\tilde{w}_i$ and an interval $[W_i]$ for each variable of $dec(c)$. The *backward* phase yields the desired forbidden box.

3.1 Forward phase

In this phase, a vector $\tilde{\mathbf{w}}$ is calculated from $\tilde{\mathbf{x}}$. Similarly, an area $[\mathbf{W}]$ is calculated from $[\mathbf{X}]$. The vector $\tilde{\mathbf{w}}$ (resp., the box $[\mathbf{W}]$) are built by instantiating leaves to $\tilde{\mathbf{x}}$ (resp. fixing their domains to $[\mathbf{X}]$) and propagating with the elementary constraints up to the root. The root constraint $w_1 = 0$ is ignored in this phase.

Base Case. For all $i > k$, set $\tilde{w}_i \leftarrow \tilde{x}_{i-k}$ and $[W_i] \leftarrow [X_{i-k}]$.

Induction. If $i < k$, w_i is the output of an elementary constraint. If the constraint is $w_i = \phi(w_j)$, we set:

$$\tilde{w}_i \leftarrow \phi(\tilde{w}_j).$$

We also set $[W_i]$ to the image of $[W_j]$ by ϕ using standard interval arithmetic:

$$[W_i] \leftarrow \phi([W_j]).$$

If the constraint is $w_i = w_{j_1} \circ w_{j_2}$, we set in a similarly way:

$$\begin{array}{ll} (i) & \tilde{w}_i \leftarrow \tilde{w}_{j_1} \circ \tilde{w}_{j_2}, \\ (ii) & [W_i] \leftarrow [W_{j_1}] \circ [W_{j_2}], \end{array} \tag{2}$$

At the end of the forward phase, $\tilde{\mathbf{w}}$ and $[\mathbf{W}]$ have the following properties:

$$\begin{array}{ll} (i) & \tilde{\mathbf{w}} \text{ is unfeasible w.r.t. } dec(c), \\ (ii) & \forall i,\ 1 \leq i \leq k+n, \quad \tilde{w}_i \in [W_i], \\ (iii) & \forall i < k,\ f_i([\mathbf{X}]) \subseteq [W_i]. \end{array} \tag{3}$$

(i) comes from the fact that $\tilde{\mathbf{x}}$ is unfeasible w.r.t. c, by using the same argument by contradiction as above. (ii) and (iii) are obtained by a straightforward induction (remember that $\tilde{\mathbf{x}} \in [\mathbf{X}]$).

If it turns that $0 \notin [W_1]$, then, by (3.iii), the whole box $[\mathbf{X}]$ is unfeasible. In this case, the backward phase is skipped and the area $[\mathbf{X}]$ can be directly returned.

We need now the concept of *inner inflaters* for the backward phase. We introduce it first.

3.2 Inner Inflaters

Consider first an elementary function ϕ. Intuitively, assume we have an interval $[y]$ that is "forbidden", that is, values in $[y]$ are not allowed in some "context" (all this will be restated on a more rigorous footing in §3.3). The purpose of an inner inflater is to calculate an interval $[x]$ (as large as possible) such that $\phi([x]) \subseteq [y]$. Then, $[x]$ is "forbidden" since the image of all x in $[x]$ is a forbidden point of $[y]$.

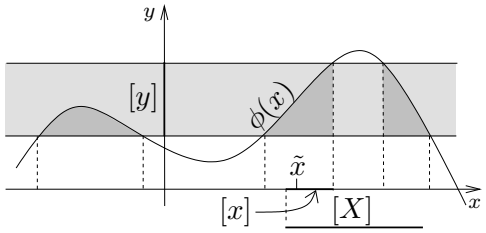

Fig. 5: **Inner inflation for a function ϕ.** The initial forbidden point is $\tilde{x}$ and the working area $[X]$. The result of the inflater is the interval $[x]$.

However, there are often different possible intervals $[x]$, as illustrated in Figure 5. For instance, if $y = x^2$ and $[y] = [4, 9]$, then $[x]$ can be set to $[-3, -2]$ or $[2, 3]$. We will require the interval to contain a specific point $\tilde{\mathbf{x}}$ (a precondition being $\phi(\tilde{x}) \in [y]$). With this additional parameter, we can select a (maximal) interval containing $\tilde{x}$ and whose image is included in $[y]$. We will also require the result to be limited by an "area" $[\mathbf{X}]$ (again, a precondition being $\tilde{x} \in [\mathbf{X}]$).

The exact same idea carries over binary operators. All is summarized in the following definition:

Definition 3 (Inner Inflater). *Let ϕ be an elementary function. An inner inflater of ϕ is an operator $[\phi]^{\diamondsuit}$ that takes as input a point $\tilde{x}$ and two intervals $[X]$ and $[y]$ such that $\tilde{x} \in [X]$ and $\phi(\tilde{x}) \in [y]$. Then, the result*

$$[x] := [\phi]^{\diamondsuit}(\tilde{x}, [X], [y]) \text{ satisfies } \begin{cases} [x] \subseteq [X] \\ x \in [x] \\ \phi([x]) \subseteq [y] \end{cases}$$

Similarly, if $\circ$ is an operator, an inner inflater of $\circ$ is an operator $[\circ]^{\diamondsuit}$ that takes as input a couple $(\tilde{x}, \tilde{y})$ and three intervals $[X], [Y]$ and $[z]$ such that $(\tilde{x}, \tilde{y}) \in$

$[X]\times[Y]$ *and* $\tilde{x}\circ\tilde{y}\in[z]$. *Then the result*

$$[x]\times[y] := [\circ]^{\Diamond}(\tilde{x},\tilde{y},[X],[Y],[z]) \text{ satisfies } \begin{cases} [x]\times[y]\subseteq[X]\times[Y] \\ (\tilde{x},\tilde{y})\in[x]\times[y] \\ [x]\circ[y]\subseteq[z] \end{cases}$$

Examples of algorithms for inner inflaters will be given in §3.5.

3.3 Backward phase

The backward phase inflates $\tilde{\mathbf{w}}$ to a forbidden box $[\mathbf{f}]$ of the following form:

$$[\mathbf{f}] := (-\infty,+\infty)^k \times [w_{k+1}]\times\cdots\times[w_{k+n}]. \tag{4}$$

that also satisfies

$$(\tilde{w}_{k+1},\ldots,\tilde{w}_{k+n})\in[w_{k+1}]\times\cdots\times[w_{k+n}]\subseteq[W_{k+1}]\times\cdots[W_{k+n}]. \tag{5}$$

To prove the correctness, we shall consider boxes $[\mathbf{f}^{(1)}],\ldots,[\mathbf{f}^{(k+1)}]$ (of course, these boxes are not actually *built*). The invariant satisfied by the i^{th} box is:

$$\begin{cases} [\mathbf{f}^{(i)}] \text{ is forbidden} & (i) \\ j<i \Longrightarrow [f^{(i)}]_j = (-\infty,+\infty) & (ii) \\ (j>i \wedge \mathit{output}(j)\geq i) \Longrightarrow [f^{(i)}]_j = (-\infty,+\infty) & (iii) \\ (j\geq i \wedge \mathit{output}(j)<i) \Longrightarrow [f^{(i)}]_j \subseteq [W]_j & (iv) \\ \forall j,\ \tilde{w}_j\in[f^{(i)}]_j & (v) \end{cases} \tag{6}$$

We proceed this time from the root down to the leaves of the tree.

Base Case The base case is the root constraint $w_1=0$, $w_1\leq 0$ or $w_1\geq 0$. Consider $w_1=0$ (the other cases are derived straightforwardly). By (3.ii) $\tilde{w}_1\in[W_1]$. We have to build $[w_1]\subseteq[W_1]$ such that $\tilde{w_1}\in[w_1]$ and $[w_1]\cap\{0\}=\emptyset$:

If $\tilde{w_1}>0$, we just set $[w_1]$ to $[W_1]\cap(0,+\infty)$. In practice, the open bound 0 can be replaced by any floating point number greater than 0 and smaller than $\tilde{w_1}$. Similarly, if $\tilde{w_1}<0$, $[w_1]$ is set to $[W_1]\cap(-\infty,0)$.

At this point, the box $[\mathbf{f}^{(1)}] := [w_1]\times(-\infty,+\infty)^{k+n-1}$ satisfies (6), as one can check directly.

Now, the intuition behind the recursion is to move the "bubble" (the interval $[w_1]$) downto the leaves by swapping successively the bubble with infinite intervals.

Induction Assume now by induction that for $i>0$, a box $[\mathbf{f}^{(i)}]$ satisfies (6).

If $i=k+1$, we are done (w_i and the subsequent variables are the leaves) and the desired box $[\mathbf{f}]$ is simply $[\mathbf{f}^{(k+1)}]$. The conditions (4) and (5) required for $[\mathbf{f}]$ are fulfilled via (6.ii), (6.iv) and (6.v).

If $i \leq k$, w_i is the output variable of an elementary constraint. To deal with the most general situation, we shall consider that the constraint is $w_i = w_{j_1} \circ w_{j_2}$ (and necessarily, $j_1 > i$, $j_2 > i$). The case of a binary constraint $w_i = \phi(w_{j_1})$ is obtained easily by canceling terms with j_2 in what follows.

Since $(\tilde{w}_{j_1}, \tilde{w}_{j_2}) \in [W_{j_1}] \times [W_{j_2}]$ by (3.ii) and since $\tilde{w}_{j_1} \circ \tilde{w}_{j_2} = \tilde{w}_i$ by (2.i) with $\tilde{w}_i \in [w_i]$ by (6.v), we can apply the inner inflater of the constraint as follows:

$$[w_{j_1}] \times [w_{j_2}] \leftarrow [\circ]^{\diamondsuit}(\tilde{w}_{j_1}, \tilde{w}_{j_2}, [W_{j_1}], [W_{j_2}], [w_i]).$$

Assume now that all the other components w_k ($k \notin \{i, j_1, j_2\}$) are instantiated to any values inside their respective domains $[f^{(i)}]_k$. By (6.i) and (6.iii) we have:

$$\forall w_i \in [w_i],\ \forall w_{j_1} \in (-\infty, +\infty),\ \forall w_{j_2} \in (-\infty, +\infty) \quad w \text{ is unfeasible.}$$

In particular,

$$\forall w_i \in [w_i],\ \forall w_{j_1} \in [w_{j_1}],\ \forall w_{j_2} \in [w_{j_2}] \quad w \text{ is unfeasible.}$$

Now, for all $(w_{j_1}, w_{j_2}) \in [w_{j_1}] \times [w_{j_2}]$, either $w_i \in [w_i]$ and w is unfeasible by virtue of the previous relation, either $w_i \notin [w_i]$ and $w_i \neq w_{j_1} \circ w_{j_2}$ (by virtue of the inner inflation) which also means that w is unfeasible. Finally:

$$\forall w_i \in (-\infty, +\infty),\ \forall w_{j_1} \in [w_{j_1}],\ \forall w_{j_2} \in [w_{j_2}] \quad w \text{ is unfeasible.} \tag{7}$$

We have "shifted the bubble" from i to j_1 and j_2. Build $[\mathbf{f}^{(i+1)}]$ as follows:

$$\begin{array}{ll} \forall k \notin \{i, j\},\ [f^{(i+1)}]_k \leftarrow [f^{(i)}]_k, & [f^{(i+1)}]_i \leftarrow (-\infty, +\infty), \\ [f^{(i+1)}]_{j_1} \leftarrow [w_{j_1}], & [f^{(i+1)}]_{j_2} \leftarrow [w_{j_2}]. \end{array}$$

Since the choice for the other components was arbitrary in (7), we have that $[\mathbf{f}^{(i+1)}]$ is forbidden. The other properties of (6) simply follow from the way $[\mathbf{f}^{(i+1)}]$ was built. □

The complexity of both phases is linear in the size of the tree (i.e., the length of the constraint expression) as long as inner inflaters take constant time. Furthermore, if inner inflaters are optimal, it can be easily proven by induction that the forbidden box is optimal in the n directions represented by the leaves of the tree. This means that the inflation w.r.t. the original constraint c is optimal if variables have no multiple occurrences. Otherwise, the best we can do is to take the union of the $[w_i]$'s corresponding to the same variable.

Requiring inner inflaters to be optimal and in constant time is easily fulfilled, as it will be shown below.

3.4 Remark on the Area of Dummy Variables

One may wonder why we need to restrict inflation of the w_i's with $i \leq k$ to an area $[W_i]$ since these variables disappear at the end of the process. Notice first that extending w_i out of the bounds of $[W_i]$ is useless because of (3.iii). Such values for w_i only have support on x outside of $[\mathbf{X}]$. Now, by limiting in this way the inflation for a dummy variable, we can improve the inflation for another one (this will be illustrated in Figure 6) resulting, in fine, to a wider inflated box.

3.5 Algorithms for Inner Inflaters

In this section, we give two examples of inner inflaters: one for the square function and one for the addition. In order to give turn-key algorithms, we shall consider roundoff issues. All the operations are now potentially inaccurate and can be rounded upward or downward. The two variants are marked by the symbols $^{\triangle}$ and $^{\triangledown}$ respectively. For instance, $x \hat{+} y$ is the addition rounded upward; $\sqrt[\triangledown]{y}$ is the square root of y rounded downward.

In the forward phase above, the vector $\tilde{\mathbf{w}}$ was calculated from $\tilde{\mathbf{x}}$ by applying elementary functions and operators recursively. Because of rounding errors, all these operations are inaccurate and must be replaced by interval counterparts. Therefore, all the $\tilde{w}_i$'s are now intervals $[\tilde{w}_i]$. And, for the leaves, we set $[\tilde{w}_{k+i}]$ to the degenerated interval $[\tilde{x}_i, \tilde{x}_i]$. Note that rounding may cause $[\tilde{w}_1]$ to contain 0 although the initial box $[\tilde{\mathbf{x}}]$ is forbidden w.r.t. c. In this case, the backward phase can also be skipped since no inflation at all can be expected.

Square root

Algorithm 2: $[sqr]^{\diamondsuit}([\tilde{x}], [X], [y])$

Input: three intervals $[\tilde{x}]$, $[y]$ and $[X]$ with $[\tilde{x}] \subseteq [X]$ and $[\tilde{x}]^2 \subseteq [y]$
Output: a maximal interval $[x]$ such that $[\tilde{x}] \subseteq [x] \subseteq [X]$ and $[x]^2 \subseteq [y]$

```
1 u ← max{ ▽√ȳ, 0};
2 if y > 0 then
3 |  l ← △√y;
4 |  if l < u then
5 |  |  if x̃ > 0 then return ([X] ∩ [l, u]) ∪ [x̃];
6 |  |  else return ([X] ∩ [−u, −l]) ∪ [x̃];
7 |  else return [x̃];
8 else return ([X] ∩ [−u, u]) ∪ [x̃];
```

line 1. u represents the maximal upper bound of $[x]$. To guarantee $[x]^2 \subseteq [y]$, we must have $\overline{x}^2 \leq \overline{y}$, i.e., $\overline{x} \leq \sqrt{\overline{y}}$. Therefore, $\sqrt{\overline{y}}$ has to be rounded downward. But, at least in theory, this may lead to a negative number, whence the max.
line 2. If $\underline{y} > 0$, there are two disjoint intervals whose image is $[y]$ (e.g., if $[y] = [4, 9]$, $[-3, -2]^2 = [2, 3]^2 = [y]$). We must identify the one that contains $\tilde{x}$.
line 3. l represents the lowest positive bound for $[x]$ which is $\sqrt{\underline{y}}$ rounded upward.
lines 4 & 7. Because of rounding, l is not necessarily smaller than u. If $l > u$, inflation has failed and the best we can do is to return $[\tilde{x}]$.
lines 5 & 6. If $\tilde{x} > 0$, the nonnegative interval is selected and intersected with the area $[X]$. Still because of rounding, $[\tilde{x}]$ may not be included in the resulting interval. It has to be merged. The case $\tilde{x} < 0$ is symmetric.
line 8. If $l = 0$, the largest interval whose image is included in $[y]$ is $[-u, u]$. The latter must be intersected with the area $[X]$ and merged with $[\tilde{x}]$ as before.

Addition

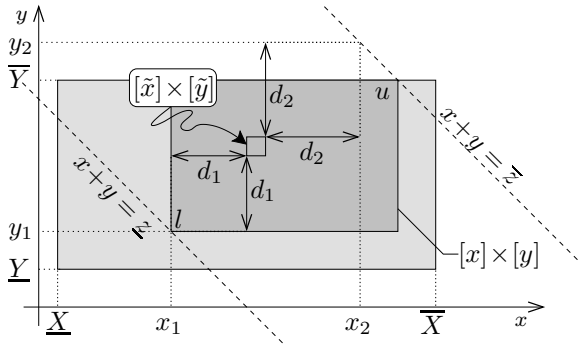

Fig. 6: **Inner inflation for addition.** Isolines $x+y=\underline{z}$ and $x+y=\overline{z}$ are drawn with dashes. All couple (x,y) between by these two lines satisfy $x+y\in[z]$. Therefore, the box $[\tilde{x}]\times[\tilde{y}]$ must be inflated inside this area. The half distances d_1 and d_2 gives a perfectly balanced box (a square). However, the surrounding $[X]\times[Y]$ may limit inflation in one direction, giving more freedom in the other one. This appears here with $\overline{Y}$ being less than y_2. The resulting box $[x]\times[y]$ is therefore a rectangle elongated on the x axis.

Algorithm 3: $[+]^{\diamondsuit}(\tilde{x},\tilde{y},[X],[Y],[z])$

Input: 5 intervals $[\tilde{x}],[\tilde{y}],[X],[Y],[z]$ such that $[\tilde{x}]\times[\tilde{y}]\subseteq[X]\times[Y]\ \wedge\ [\tilde{x}]+[\tilde{y}]\subseteq[z]$
Output: a box $[x]\times[y]$ such that $[\tilde{x}]\times[\tilde{y}]\subseteq[x]\times[y]\subseteq[X]\times[Y]$ and $[x]+[y]\subseteq[z]$

1 $d_1\leftarrow(\underline{\tilde{x}}\,\underline{\dot{+}}^{\nabla}\,\underline{\tilde{y}}\,\underline{\dot{-}}^{\nabla}\,\underline{z})\,\underline{/}^{\nabla}2$; $d_2\leftarrow(\overline{z}\,\overline{\dot{-}}^{\nabla}\,\overline{\tilde{x}}\,\overline{\dot{-}}^{\nabla}\,\overline{\tilde{y}})\,\overline{/}^{\nabla}2$;
2 $x_1\leftarrow\underline{\tilde{x}}\,\dot{+}^{\triangle}\,d_1$; $x_2\leftarrow\overline{\tilde{x}}\,\overline{\dot{+}}^{\nabla}\,d_2$;
3 $y_1\leftarrow\underline{\tilde{y}}\,\dot{+}^{\triangle}\,d_1$; $y_2\leftarrow\overline{\tilde{y}}\,\overline{\dot{+}}^{\nabla}\,d_2$;
4 **if** $\underline{Y}>y_1$ **then** $l\leftarrow(\max\{\underline{X},\min\{\overline{\tilde{x}},\underline{z}\,\dot{-}^{\triangle}\,\underline{Y}\}\},\underline{Y})$;
5 **else if** $\underline{X}>x_1$ **then** $l\leftarrow(\underline{X},\max\{\underline{Y},\min\{\overline{\tilde{y}},\underline{z}\,\dot{-}^{\triangle}\,\underline{X}\}\})$;
6 **else** $l\leftarrow(x_1,y_1)$;
7 **if** $\overline{Y}<y_2$ **then** $u\leftarrow(\min\{\overline{X},\max\{\overline{\tilde{x}},\overline{z}\,\overline{\dot{-}}^{\nabla}\,\overline{Y}\}\},\overline{Y})$;
8 **else if** $\overline{X}<x_2$ **then** $u\leftarrow(\overline{X},\min\{\overline{Y},\max\{\overline{\tilde{y}},\overline{z}\,\overline{\dot{-}}^{\nabla}\,\overline{X}\}\})$;
9 **else** $u\leftarrow(x_2,y_2)$;
10 **return** $[l_1,u_1]\times[l_2,u_2]$;

The variables initialized in Lines 1-3 are all represented in Figure 6. Rounding is made in order to guarantee $x_1+y_1\geq\underline{z}$ and $x_2+y_2\leq\overline{z}$. All the other lines work symmetrically. They compute two vectors: l and u, the lower and upper corners of $[x]\times[y]$. The case depicted for u in Figure 6 correspond to Line 7. The upper bound for x is $\overline{Y}-\overline{z}$, except if this value exceeds $\overline{X}$ (whence the min).

Furthermore, downward rounding may lose the upper bound of $[\tilde{x}]$ for the same reason evoked for $[\text{sqr}]^{\Diamond}$. This bound has to be kept anyway, whence the max.

4 Discussion

In this paper, we have restated sweeping in a continuous setting and given a generic *inflater*. This inflater builds automatically forbidden boxes w.r.t. a constraint whose expression is a composition of standard mathematical operators. This makes `geost` fully generic and now applicable for packing curved objects. The inflater is fast and optimal if no variable is duplicated in the constraint, which includes situations of particular interest such as distance constraints.

This is the first theoretical contribution for adapting sweeping (hence `geost`) to continuous domains. Of course, benchmarking has to be made now to check that our revised global constraint `geost` still outperforms generic propagation techniques in presence of curved objects. One practically useful extension of this work would also be to adapt the proposed inflater to discrete domains.

References

1. Choco: An open source Java CP library. documentation. http://choco.emn.fr/.
2. M. Agren, N. Beldiceanu, M. Carlsson, M. Sbihi, C. Tuchet, and S. Zampelli. 6 Ways of Integreting Symmetries Within Non-Overlapping Constraints. In *CPAIOR'09*, pages 11–25, 2009.
3. I. Araya, G. Trombettoni, and B. Neveu. Filtering Numerical CSPs Using Well-Constrained Subsystems. In *CP'09*, pages 158–172, 2009.
4. N. Beldiceanu and M. Carlsson. Sweep as a Generic Pruning Technique Applied to the Non-Overlapping Rectangles Constraints. In *CP'01*, pages 377–391, 2001.
5. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects. In *CP'07*, pages 180–194. Springer, 2007.
6. N. Beldiceanu, M. Carlsson, and S. Thiel. Sweep Synchronisation as a Global Propagation Mechanism. *Comp. & Operations Research*, 33(10):2835–2851, 2006.
7. F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising Hull and Box Consistency. In *ICLP*, pages 230–244, 1999.
8. M. Carlsson, N. Beldiceanu, and J. Martin. A Geometric Constraint over k-Dimensional Objects and Shapes Subject to Business Rules. In *CP*, pages 220–234, 2008.
9. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium*, volume 1292, pages 191–206, 1997.
10. K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, 2003.
11. A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.