Constraints (2010) 15:213–237
DOI 10.1007/s10601-009-9084-1

A branch and bound algorithm for numerical Max-CSP

Jean-Marie Normand · Alexandre Goldsztejn · Marc Christie · Frédéric Benhamou

Published online: 11 November 2009

Abstract The Constraint Satisfaction Problem (CSP) framework allows users to define problems in a declarative way, quite independently from the solving process. However, when the problem is over-constrained, the answer "no solution" is generally unsatisfactory. A Max-CSP $\mathcal{P}_m = \langle V, \mathbf{D}, C \rangle$ is a triple defining a constraint problem whose solutions maximize the number of satisfied constraints. In this paper, we focus on numerical CSPs, which are defined on real variables represented as floating point intervals and which constraints are numerical relations defined in intension. Solving such a problem (*i.e.*, exactly characterizing its solution set) is generally undecidable and thus consists in providing approximations. We propose a Branch and Bound algorithm that provides under and over approximations of a solution set that maximize the maximum number $m_{\mathcal{P}}$ of satisfied constraints. The technique is applied on three numeric applications and provides promising results.

Keywords Numerical Max-CSP · Constraint programming · Numerical constraints

J.-M. Normand (✉) · A. Goldsztejn · M. Christie · F. Benhamou
LINA, University of Nantes, Nantes, France
e-mail: Jean-Marie.Normand@univ-nantes.fr

F. Benhamou
e-mail: Frederic.Benhamou@univ-nantes.fr

J.-M. Normand
EVENT Lab, Universitat de Barcelona, Barcelona, Spain

A. Goldsztejn
CNRS, University of Nantes, Nantes, France
e-mail: Alexandre.Goldsztejn@univ-nantes.fr

M. Christie
INRIA/IRISA, Rennes Cedex, France
e-mail: Marc.Christie@univ-nantes.fr

1 Introduction

CSP provides a powerful and efficient framework for modeling and solving problems that are well defined. However, in the modeling of real-life applications, under or over-constrained systems often occur. In embodiment design, for example, when searching for different concepts, engineers often need to tune both the constraints and the domains. In such cases, the classical CSP framework is not well adapted (a *no solution* answer is certainly not satisfactory) and there is a need for specific frameworks such as Max-CSP. Yet in a large number of cases, the nature of the model (over-, well or under-constrained) is *a priori* not known and thus the choice of the solving technique is critical. Furthermore, when tackling over-constrained problems in the Max-CSP framework, most techniques focus on discrete domains.

In this paper, we propose a branch and bound algorithm that approximate the solutions of a numerical Max-CSP. The algorithm computes both an under (often called inner in continuous domains) and an over (often called outer) approximation of the solution set of a Max-CSP along with the sets of constraints that maximize constraint satisfaction. These approximations ensure that the inner approximation contains only solutions of the Max-CSP while the outer approximation guarantees that no solution is lost during the solving process. As a consequence, since we aim at computing the sets of inner and outer approximations of a numerical Max-CSP, it has to be composed of constraints that give rise to continuum of solutions, namely inequalities.

This paper is organized as follows: Section 2 motivates the research and presents a numerical Max-CSP application. Section 3 presents the definitions associated to the Max-CSP framework. A brief introduction to Interval analysis and its related concepts is drawn in Section 4, while our branch and bound algorithm is described in Section 5. Section 6 presents the work related to Max-CSP in both continuous and discrete domains while Section 7 is dedicated to experimental results showing the relevance of our approach.

2 A motivating example

We motivate the usefulness of the numerical Max-CSP approach on a problem of virtual camera placement (VCP), which consists in positioning a camera in a 2D or a 3D environment w.r.t. constraints, namely cinematographic properties, defined on the objects of a virtual scene. For example, one wishes to see a character from a cinematographic point of view (*e.g.*, profile, front, back, *etc.*) or from a typical shot distance (close-up, long shot, *etc.*).

Within a classical CSP framework, these cinematographic properties are transformed into numerical constraints in order to find an appropriate camera position and orientation. One of the counterparts of the expressiveness offered by the constraint programming framework lies in the fact that it is easy for the user to specify an over-constrained problem. In such cases, classical solvers will output a *no solution* statement and does not provide any information neither on why the problem was over-constrained nor on how it could be softened. The Max-CSP framework will, on the other hand, explore the search space in order to compute the set of configurations that satisfy as many constraints as possible. This provides a classification of the solutions according to satisfied constraints and enables the user to select a particular solution, or to remodel the problem.

For the purpose of illustration, we present a small over-constrained problem. The scene consists of three characters defined by their positions, and by an orientation vector (that denotes the front of the character). VCP frameworks generally provide the *angle* property that specifies the orientation the character should have on the screen (*i.e.*, profile, *etc.*), and the *shot* property that specifies the way a character is shot on the screen. The first properties obviously constrains both the location and the orientation of the camera, whereas the second constrains the distance to the character.

In Fig. 1, a sample scene is displayed with three characters A, B and C, each of which must be viewed from the front and with a long shot. The regions corresponding to each property are represented (wedges for the *angle* property and circles for the *shot distance*). No region of the space fully satisfies all the constraints since the intersection of the wedges and the circles is empty, meaning that the original problem is over-constrained. Conversely, the Max-CSP can be solved and its solutions are represented as dark areas. The solving process associated to this simple example is provided in Section 7.2.

For clarity sake, we need to emphasize the fact that Fig. 1 represents a simplified version of the properties generally offered in VCP frameworks. Indeed, the shot constraint is represented here by a circle which does not take into account the actual shot properties. A shot property aims at constraining the size of an object on the screen. Classical cinematographic shot distances are taken into account by those properties, it is thus possible to constrain the size of an object (or a character) by using for example close-up, medium close-up (also called *plan américain*), or long shot distances. In order to achieve those kinds of shot, the camera has to be situated within a certain distance to the object involved. The correct modeling of such constraints

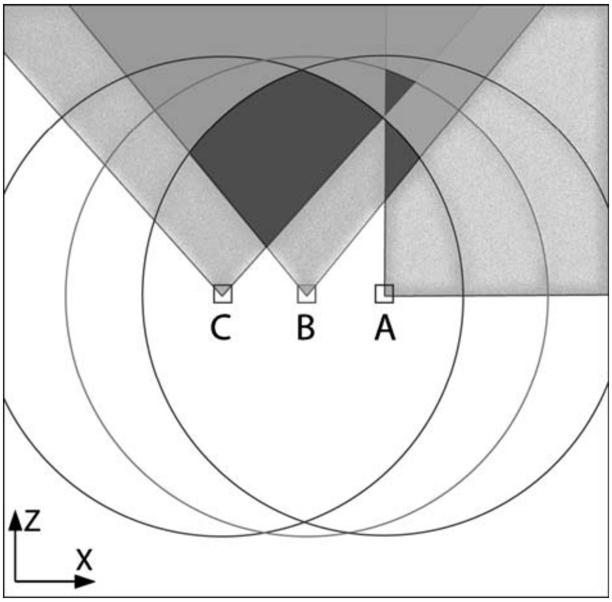

Fig. 1 Top-view of a 3D scene representing the camera's position search space for a VCP problem. Requested orientations of objects are represented by dotted wedges areas centered on each object of interest, *shot* distances are represented as *circles*. *Light gray areas* represent intersection of constraints while *dark gray areas* represent the max-solution regions of the over-constrained problem, *i.e.*, where the maximum number of constraints are satisfied

should thus be done by placing a 2D ring (for a top-view scheme such as Fig. 1 or by a 3D torus in 3D environments) around the object. For additional information on VCP frameworks and the modeling of the corresponding properties into a constraint satisfaction problem, the reader is invited to refer to [7].

3 The Max-CSP framework

The current section is dedicated to the Max-CSP framework and its related concepts. According to [11, 28], a Max-CSP $\mathcal{P}$ is a triplet $\langle V, \mathbf{D}, C\rangle$ where $V = \{x_1, \ldots, x_n\}$, $D = \{D_1, \ldots, D_n\}$ and $C = \{c_1, \ldots, c_m\}$ are respectively a set of variables, a set of domains assigned to each variable and a set of constraints. For clarity sake, we use vectorial notations where vectors are denoted by boldface characters: the vector $\mathbf{x} = (x_1, \ldots, x_n)$ represents the variables and $\mathbf{D} = D_1 \times \ldots \times D_n$ is interpreted as the Cartesian product of the variables domains. To define solutions of a Max-CSP, a function sat_c is first defined as follows, for each constraint $c \in \mathcal{C}$ and for each n-dimensional $\mathbf{x}$ in $\mathbf{D}$:

$$sat_c(\mathbf{x}) = \begin{cases} 1 \text{ if } c(\mathbf{x}) \text{ is true} \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

Once the satisfaction function sat_c is defined for a constraint c, we can introduce a satisfaction function, $sat_{\mathcal{C}}$, defined over a set of constraints $\mathcal{C}$ and $\forall \mathbf{x} \in \mathbf{D}$ as follows:

$$sat_{\mathcal{C}}(\mathbf{x}) = \sum_{c \in \mathcal{C}} sat_c(\mathbf{x}) \tag{2}$$

Obviously, we have $sat_{\mathcal{C}}(\mathbf{x}) \leq \#C$ since at most all the constraints of $\mathcal{C}$ are satisfied.

In the sequel, the maximum number of satisfied constraints of a Max-CSP $\mathcal{P}$ is denoted by $m_{\mathcal{P}}$ and formally defined as follows:

$$m_{\mathcal{P}} := \max_{\mathbf{x} \in \mathbf{D}} \ sat_{\mathcal{C}}(\mathbf{x}). \tag{3}$$

In the context of Max-CSP solving, one is usually more interested in the maximum number $m_{\mathcal{P}}$ of satisfied constraints than in the variables assignments that actually satisfy the $m_{\mathcal{P}}$ constraints. However, for our purpose, the set of these Max-CSP solutions is of central importance, since we want to know the assignments. We call it the *solution set* of the Max-CSP, and denote it by MaxSol($\mathcal{P}$). Formally, the following set of solution vectors (or variables assignments in the discrete case):

$$\text{MaxSol}(\mathcal{P}) = \left\{ \mathbf{x} \in \mathbf{D} : \sum_{c \in \mathcal{C}} sat_c(\mathbf{x}) = m_{\mathcal{P}} \right\}. \tag{4}$$

4 Interval analysis for numerical CSPs

Numerical CSPs present the additional difficulty of dealing with continuous domains, not exactly representable in practice. As a consequence, it is impossible to enumerate all the values that can take variables in their domains. Interval analysis (*cf.* [15, 26]) is a key framework in this context: soundness and completeness properties of real values belonging to some given intervals can be ensured by using correctly rounded computations over floating point numbers. We now present basic concepts allowing us to ensure that an interval vector contains no solution (resp. only solutions) of an inequality constraint.

4.1 Intervals, interval vectors and interval arithmetic

An interval $[x]$ with floating-point bounds (floating points are numbers representable in practice on a computer, *cf.* [14]) is called a *floating-point interval*. It is of the form: $[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$ where $\underline{x}, \overline{x} \in \mathbb{F}$. Given a floating-point value x, we denote by $x^{\oplus}$ (resp. $x^{\ominus}$) the smallest floating point value strictly greater than x (respectively, the greatest floating point value smaller than x).

An interval vector (also called a box) $[\mathbf{x}] = ([x_1], \ldots, [x_n])$ represents the Cartesian product of the n intervals $[x_i]$. The set of interval vectors with representable bounds is denoted by $\mathbb{IF}^n$.

Operations and functions over reals are also replaced by an *interval extension* enjoying the *containment property* (see [24]).

Definition 1 (Interval extension [26]) Given a real function $f\colon \mathbb{R}^n \to \mathbb{R}$, an *interval extension* $[f]\colon \mathbb{IF}^n \to \mathbb{IF}$ of f is an interval function verifying

$$[f]([\mathbf{x}]) \supseteq \{f(\mathbf{x}) : \mathbf{x} \in [\mathbf{x}]\}. \tag{5}$$

Interval extensions of elementary functions can be computed formally thanks to their simplicity, *e.g.*, for $f(x, y) = x + y$ we have $[f]([x], [y]) = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. To obtain an efficient framework, these elementary interval extensions are used to define an interval arithmetic.

For example, the interval extension of $f(x, y) = x + y$ is used to define the interval addition: $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. Therefrom, an arithmetical expression can be evaluated for interval arguments. The fundamental theorem of interval analysis [24] states that the interval evaluation of an expression gives rise to an interval extension of the corresponding real function. For example:

$$[x] + \exp([x] + [y]) \supseteq \{x + \exp(x + y) : x \in [x], y \in [y]\}. \tag{6}$$

4.2 Interval contractors

Discarding all inconsistent values from a box composed of variables domains in a CSP is an intractable problem when the constraints are real ones. There exist many techniques to obtain a contraction, like 2B consistency (also called hull consistency) [4, 20], box consistency [3] or interval Newton operator [26]. Those methods rely on outer contracting operators that discard values according to a given consistency. These methods are instances of the local consistency framework in artificial intelligence [21].

Definition 2 (Interval contracting operator) Let c be an n-ary constraint, a contractor for c is a function $\text{Contract}_c : \mathbb{IF}^n \longrightarrow \mathbb{IF}^n$ which satisfies

$$\mathbf{x} \in [\mathbf{x}] \wedge c(\mathbf{x}) \implies \mathbf{x} \in \text{Contract}_c([\mathbf{x}]). \tag{7}$$

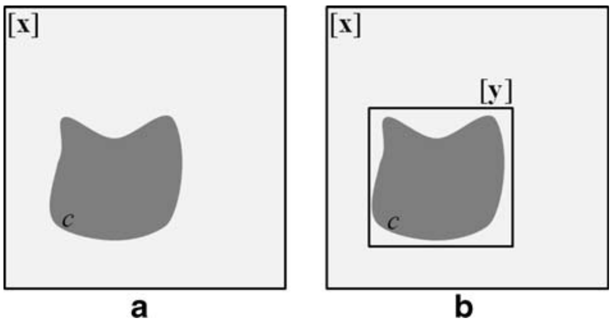

Fig. 2 Diagrams **a** shows the constraint c (satisfied in *dark gray regions*) in an original box $[x]$ that represents the search space. Diagram **b** shows how applying an interval contracting operator over the constraint c and the box $[x]$ gives rise to a new box $[y]$ in which as many inconsistent values of $[x]$ as possible have been removed

Figure 2 illustrates the application of an interval contracting operator in a given search space $[x]$ over a constraint c. The contractor aims at discarding from the original box $[x]$ as many inconsistent values as possible according to a given consistency. This process gives rise to a new box $[y]$ that encompasses the constraint c as tightly as possible.

Experiments of Section 7 are performed with two different contractors dedicated to inequality constraints $f(\mathbf{x}) \leq 0$. The first is called the *evaluation contractor* and is defined as follows: Given an interval extension $[f]$ of a real function f

$$\mathrm{Contract}_{f(\mathbf{x})\leq 0}(\mathbf{x}) = \begin{cases} \emptyset & \text{if } [f]([\mathbf{x}]) > 0 \\ [\mathbf{x}] & \text{otherwise.} \end{cases} \tag{8}$$

The *evaluation contractor* is thus very simple while evaluating a box $[x]$, either every value in $[x]$ are consistent and the box is kept, otherwise, $[x]$ is rejected and a null box is returned.

The second, called the *hull contractor*, is based on the hull consistency, *cf.* [4, 20] for details.[1] Given an interval extension $[f]$ of a real function f, the *hull contractor* is applied as follows:

$$\mathrm{Contract}_{f(\mathbf{x})\leq 0}(\mathbf{x}) = [\mathbf{y}] \tag{9}$$

where $[\mathbf{y}]$ is the smallest box encompassing the solutions of the real function f, in practice, only a polynomial approximation of the hull consistency is enforced using HC3 or HC4 like algorithms (*cf.* [1, 20] for more details).

Interval contractors can be used to partition a box into smaller boxes where the constraint can be decided. To this end, we introduce the *inflated set difference* in the following subsection.

4.3 Inflated set difference

When an interval contractor is applied in a search space on a constraint, it outputs a tighter box which encompasses the constraint more precisely. As for our branch and bound algorithm for numerical Max-CSP, we want to apply in sequence such

[1] A contractor based on box consistency [4] could also be used but the optimality between hull and box consistency is problem dependent.

a procedure on each constraint $c \in \mathcal{C}$ of the problem in order to get a set of boxes in which the constraints are satisfied. In order to compute the regions of the search space that maximize constraint satisfaction for a numerical Max-CSP $\mathcal{P}$, we want to compute the *inflated set difference* between every box obtained by inner and outer contractions over a constraint c.

This is achieved by introducing the operator $\mathsf{idiff}([\mathbf{x}], [\mathbf{y}])$ which computes a set of boxes $\{[\mathbf{d}_1], \ldots, [\mathbf{d}_k]\}$ included in $[\mathbf{x}]$ such that $[\mathbf{d}_i] \cap [\mathbf{y}] = \emptyset$ and $[\mathbf{d}_1] \cup \cdots \cup [\mathbf{d}_k] \cup [\mathbf{y}]^{\oplus} = [\mathbf{x}]$, where $[\mathbf{y}]^{\oplus}$ is the smallest box that strictly contains $[\mathbf{y}]$. The inflated set difference is illustrated in two dimensions for clarity sake, in Fig. 3.

When combining the results obtained by interval contraction over a constraint and the inflated set difference operator, we can compute a set of boxes that does not contain any solution of the constraint involved in the outer contraction process. Indeed, we can observe thanks to diagrams (a) and (c) of Fig. 3 that the computation of:

$$\mathsf{idiff}([\mathbf{x}], \mathrm{Contract}_c([\mathbf{x}]))$$

gives rise to a set of boxes $\otimes$ that do not contain any potential solution to the constraint c.

Moreover, following [2, 9] this technique can also be used to compute boxes that contain only solutions, applying the contractor to the negation of the constraint: $\mathsf{idiff}([\mathbf{x}], \mathrm{Contract}_{\neg c}([\mathbf{x}]))$ outputs a set of boxes that contain only solutions. Indeed, applying an outer contractor operator on the negation of a constraint encompasses the region of the search space that cannot satisfy the constraint. Thus, the set difference between the search space $[\mathbf{x}]$ and the box $[\mathbf{y}]$ (that does not contain any solution to the constraint c) gives rise to the set of boxes $\otimes$ that contains only solutions to the constraint c, (*cf.* diagrams (b) and (c) of Fig. 3).

Let us illustrate this process over a simple example. Consider the CSP $\mathcal{P} = \langle x, [-1, 1], x \leq 0\rangle$, being composed of only one variable x that takes its values in the floating-point interval $[-1, 1]$ and that involves a single inequality: $x \leq 0$.

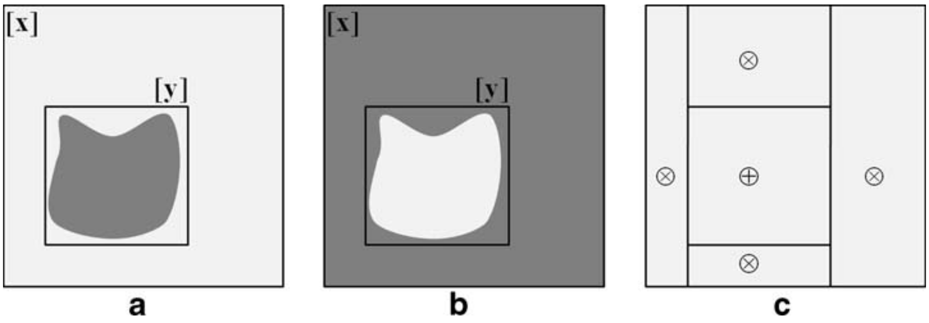

Fig. 3 Diagrams **a** and **b** show the constraint c (satisfied in *dark gray regions* and unsatisfied in *light gray regions*), the initial SU-box $\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$ and the contracted box $[\mathbf{y}]$ for two different situations: **a** outer contraction and **b** inner contraction for constraint c. Diagram **c**: Five new SU-boxes are inferred from the contraction of $[\mathbf{x}]$ to $[\mathbf{y}]$. The contraction does not provide any information on the SU-box $\oplus = \langle[\mathbf{x}] \cap [\mathbf{y}], \mathcal{S}, \mathcal{U}\rangle$, hence its sets of constraints remain unchanged. The SU-boxes $\otimes = \langle[\mathbf{x}'], \mathcal{S}', \mathcal{U}'\rangle$ are proved to contain no solution of c (if outer pruning was applied) or only solutions (if inner pruning was applied), their sets of constraint can thus be updated: $\mathcal{U}' = \mathcal{U} \backslash \{c\}$, and, $\mathcal{S}' = \mathcal{S}$ for outer pruning, while $\mathcal{S}' = \mathcal{S} \cup \{c\}$ and $\mathcal{U}' = \mathcal{U}$ for inner pruning

The first process is to apply an outer contracting operator on the constraint in order to remove as many inconsistent values as possible from the original search space ($[-1, 1]$). Then, we observe that:

$$\text{Contract}_{x\leq 0}([-1, 1]) = [-1, 0]$$

meaning that the interval $[-1, 0]$ contains all the solutions of the constraint $x \leq 0$ within the domain $[-1, 1]$.

What we want to do now is to compute the interval that does not contain any solution to the CSP $\mathcal{P}$. This is achieved by computing a set difference (denoted by the operator $\backslash$) between the original search space $[-1, 1]$ and the result of the outer contraction $[-1, 0]$:

$$[-1, 1]\backslash[-1, 0] =]0, 1].$$

This set difference computation outputs an semi-open interval $]0, 1]$ that is not easily dealt with in the context of interval analysis.

In order to bypass this limitation, we introduce the operator idiff that computes the inflated set difference between two intervals. This operator is thus able to compute a set difference without using any open intervals. Indeed, if we compute the inflated set difference on the intervals used in our previous example, the result is the following:

$$\text{idiff}([-1, 1], [-1, 0]) = [0^{\oplus}, 1].$$

where $0^{\oplus}$ is the smallest representable number greater than 0. As a consequence, the constraint $x \leq 0$ is proved to be false inside $[0^{\oplus}, 1]$. Nonetheless, in order to provide a full covering of the initial domains of the search space, the interval $[-1, 0^{\oplus}]$ is kept for further processing instead of $[-1, 0]$.

To sum up, the outer contraction has proved that the interval $[-1, 0]$ contains all the solutions (but not only solutions) to the CSP $\mathcal{P}$ because the outer contracting step has removed as many inconsistent values as possible. As a consequence and thanks to the inflated set difference operator idiff, we also proved that the interval $[0^{\oplus}, 1]$ does not contain any solution of $\mathcal{P}$.

Finally, the domain $[-1, 0^{\oplus}]$ remains to be processed, and we look for solutions therein by computing $\text{Contract}_{x>0}([-1, 0^{\oplus}]) = [0, 0^{\oplus}]$. This time, the outer contracting operator is applied to the negation of the only constraint of $\mathcal{P}$. We use again $\text{idiff}([-1, 0^{\oplus}], [0, 0^{\oplus}]) = [-1, 0^{\ominus}]$ (where $0^{\ominus}$ is the largest representable number lower than 0) to infer that $[-1, 0^{\ominus}]$ contains only solutions. Finally, the constraint cannot be decided for the values of the interval $[0^{\ominus}, 0^{\oplus}]$, it is called a *boundary box*.

5 The algorithm

The main idea of the algorithm is to attach to each box, computed by a tree search, the set of constraints that were proved to be satisfied for all vectors of this box ($\mathcal{S}$) and the set of constraints that have not been decided yet, *i.e.*, unknown constraints ($\mathcal{U}$). The pseudo-code of our branch and bound algorithm for numerical Max-CSP is presented in Algorithm 2 and detailed in the last subsection. Before presenting the concepts and procedures required by our algorithm, we draw a global sketch of the algorithm.

First, for each constraint c of the problem, the algorithm computes an outer contraction of the current box $[\mathbf{x}]$ (*i.e.*, the current search space). This step removes as much inconsistent values as possible for constraint c in $[\mathbf{x}]$. In order to ensure the exhaustive exploration of the search space, and thus to enforce the *completeness* property of the algorithm, a set difference operator is applied in a second step between the original search space $[\mathbf{x}]$ and the result of the outer contraction $[\mathbf{y}]$, giving rise to a set of boxes $\{[\mathbf{z}_i]\}$. From this set, only the boxes that can satisfy more constraints than the current lower bound ($\underline{m}$) of the algorithm are kept for further study.

In a third step, an inner contraction is applied for the constraint c in the original search space $[\mathbf{x}]$. This is achieved by applying a classical outer contraction procedure on the negation $\neg c$ of the constraint. This step outputs a box $[\mathbf{w}]$ that removes all inconsistent values for $\neg c$, as shown in Fig. 3. As a consequence the set difference between boxes $[\mathbf{x}]$ and $[\mathbf{w}]$ (*i.e.*, $[\mathbf{x}]\backslash[\mathbf{w}]$) only contain values consistent with constraint c.

However, since the original search space $[\mathbf{x}]$ has already given rise to the set of boxes $\{[\mathbf{z}_i]\}$ in the outer contraction step, in order to speed up the solving process, we do not compute the set difference between $[\mathbf{x}]$ and $[\mathbf{w}]$ but between $\{[\mathbf{z}_i]\}$ and $[\mathbf{w}]$, as shown in Fig. 4. This gives rise to a new set of boxes $\{[\mathbf{u}_i]\}$ that allows us to update the lower bound of the algorithm as well as to further reject uninteresting boxes. Finally, we iterate on the previous steps until no more new boxes are created or all the constraints have been decided. In order to converge, bisection steps might be applied in order to obtain smaller boxes where more constraints can be decided.

The remainder of this section is constructed as follows: first we propose the definition of a SU-box, which represents a triplet consisting of a box $[\mathbf{x}]$ and the two sets of constraints, $\mathcal{S}$ and $\mathcal{U}$, described above as the sets of satisfied and unknown constraints for $[\mathbf{x}]$. Then, we present the specification and usage of outer and inner contractions in the following three subsections. Finally, as stated above, we describe the core algorithm in the last subsection.

5.1 SU-boxes

The main idea of our algorithm is to associate to a box $[\mathbf{x}]$ the sets of constraints $\mathcal{S}$ and $\mathcal{U}$ which respectively contain the constraints proved to be satisfied for all vectors

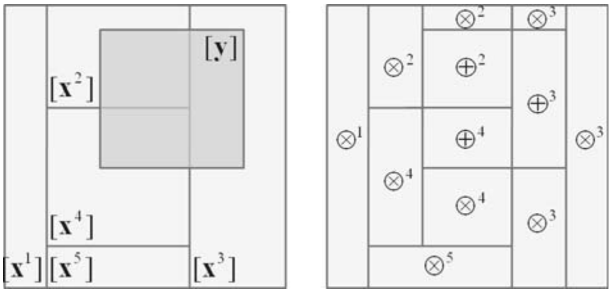

Fig. 4 *Left hand side* illustrates the $[\mathbf{x}^k]$ boxes obtained from Fig. 3. Moreover, the box $[\mathbf{y}]$ represents the contraction (inner or outer) of a box. *Right hand side*: The process illustrated by Fig. 3 is repeated for each box $[\mathbf{x}^k]$, leading to twelve new SU-boxes, among which $\otimes^k$ have their sets of constraints updated

in $[\mathbf{x}]$ and the constraints which remain to be treated. Such a box, altogether with the satisfaction information of each constraint is called a *SU-box*[2] and is denoted by $\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$. The following definition naturally links a SU-box with a Max-CSP $\mathcal{P}$:

Definition 3 Let $\mathcal{P}$ be a CSP whose set of constraints is $\mathcal{C}$. Then, a SU-box $\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$ is consistent with $\mathcal{P}$ if and only if the three following conditions hold:

1. $(\mathcal{S} \cup \mathcal{U}) \subseteq \mathcal{C}$ and $(\mathcal{S} \cap \mathcal{U}) = \emptyset$
2. $\forall \mathbf{x} \in [\mathbf{x}]\ ,\ \forall c \in \mathcal{S}\ ,\ c(\mathbf{x})$ (the constraints of $\mathcal{S}$ are true everywhere in $[\mathbf{x}]$)
3. $\forall \mathbf{x} \in [\mathbf{x}]\ ,\ \forall c \in (\mathcal{C} \backslash (\mathcal{S} \cup \mathcal{U}))\ ,\ \neg\, c(\mathbf{x})$ (the constraints of $\mathcal{C}$ which are neither in $\mathcal{S}$ nor in $\mathcal{U}$ are false everywhere in $[\mathbf{x}]$).

A SU-box $\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$ will be denoted by $\langle\mathbf{x}\rangle$, without any explicit definition when there is no confusion. Given a SU-box $\langle\mathbf{x}\rangle := \langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$, $[\mathbf{x}]$ is denoted by $\text{box}(\langle\mathbf{x}\rangle)$ and is called the *domain* of the SU-box. The interest of the two sets of constraints $\mathcal{S}$ and $\mathcal{U}$ is to update the bounds of the branch and bound algorithm. Indeed, the cardinality of $\mathcal{S}$ (denoted by $\#\mathcal{S}$) represents the lower bound on the number of satisfied constraint by a SU-box $\langle\mathbf{x}\rangle$, while the cardinality of both $\mathcal{S}$ and $\mathcal{U}$ (*i.e.*, $\#\mathcal{S} + \#\mathcal{U}$) will bound the maximum number of constraints that can be satisfied by $\langle\mathbf{x}\rangle$.

5.2 The function InferSU$_{\text{outer}}$

Let us consider a SU-box $\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$, a box $[\mathbf{y}] \subseteq [\mathbf{x}]$ and a constraint c, where the box $[\mathbf{y}]$ is obtained by computing $[\mathbf{y}] = \text{Contract}_c([\mathbf{x}])$. Therefore, as described in Section 4.3, every box of $\mathsf{idiff}([\mathbf{x}], [\mathbf{y}])$ does not contain any solution of c. This is illustrated by Fig. 3. The SU-boxes created with the boxes from the inflated set difference can thus be updated by discarding c from their set of constraints $\mathcal{U}$. This is formalized by the following definition:

Definition 4 The function $\text{InferSU}_{\text{outer}}(\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle, [\mathbf{y}], c)$ returns the following set of SU-boxes:

$$\left\{\langle[\mathbf{x}'], \mathcal{S}, \mathcal{U}'\rangle : [\mathbf{x}'] \in \mathsf{idiff}([\mathbf{x}], [\mathbf{y}])\right\} \cup \left\{\langle[\mathbf{y}]^{\oplus} \cap [\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle\right\}, \tag{10}$$

where $\mathcal{U}' = \mathcal{U} \backslash \{c\}$.

Here, the box $[\mathbf{x}]$ is partitioned to $\mathsf{idiff}([\mathbf{x}], [\mathbf{y}])$ and $[\mathbf{y}]^{\oplus} \cap [\mathbf{x}]$. Each of these boxes is used to form new SU-boxes, with a reduced set of constraints $\mathcal{U}'$ for the SU-boxes coming from $\mathsf{idiff}([\mathbf{x}], [\mathbf{y}])$. This is formalized by the following proposition, which obviously holds:

Proposition 1 *Let* $\langle\mathbf{x}\rangle := \langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$ *be a SU-box consistent with a CSP* $\mathcal{P}$, $c \in \mathcal{U}$ *and* $[\mathbf{y}]$ *satisfying:* $\forall \mathbf{x} \in \mathsf{idiff}([\mathbf{x}], [\mathbf{y}]) \Rightarrow \neg\, c(\mathbf{x})$.

We define $\mathcal{T} := \text{InferSU}_{\text{outer}}(\langle\mathbf{x}\rangle, [\mathbf{y}], c)$. *As a consequence, every SU-box of* $\mathcal{T}$ *is consistent with* $\mathcal{P}$. *Furthermore,* $\cup\{\text{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\}$ *is equal to* $[\mathbf{x}]$.

[2]*SU-boxes* have been introduced in [27] in a slightly different but equivalent way.

5.3 The function $\mathrm{InferSU}_{\mathrm{inner}}$

This function is equivalent to $\mathrm{InferSU}_{\mathrm{outer}}$, but deals with inner contraction. As a consequence, the only difference is that, instead of discarding the constraint c from $\mathcal{U}$ for the SU-boxes outside $[\mathbf{y}]$, this constraint is stored in $\mathcal{S}$. Figure 3 also illustrates these computations. The following definition and proposition mirror Definition 4 and Proposition 1.

Definition 5 The function $\mathrm{InferSU}_{\mathrm{inner}}(\langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle, [\mathbf{y}], c)$ returns the following set of SU-boxes:

$$\left\{\langle[\mathbf{x}'], \mathcal{S}', \mathcal{U}'\rangle : [\mathbf{x}'] \in \mathsf{idiff}([\mathbf{x}], [\mathbf{y}])\right\} \cup \left\{\langle[\mathbf{y}]^{\oplus} \cap [\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle\right\}, \tag{11}$$

where $\mathcal{S}' = \mathcal{S} \cup \{c\}$ and $\mathcal{U}' = \mathcal{U} \backslash \{c\}$.

Proposition 2 *Let* $\langle\mathbf{x}\rangle := \langle[\mathbf{x}], \mathcal{S}, \mathcal{U}\rangle$ *be a SU-box consistent with a CSP* $\mathcal{P}$*,* $c \in \mathcal{U}$ *and* $[\mathbf{y}]$ *satisfying:* $\forall \mathbf{x} \in \mathit{idiff}([\mathbf{x}], [\mathbf{y}]) \Rightarrow c(\mathbf{x})$.

We define $\mathcal{T} := \mathrm{InferSU}_{\mathrm{inner}}(\langle\mathbf{x}\rangle, [\mathbf{y}], c)$*. As a consequence, every SU-box of* $\mathcal{T}$ *is consistent with* $\mathcal{P}$*. Furthermore,* $\cup\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\}$ *is equal to* $[\mathbf{x}]$.

Following principle of $\mathrm{InferSU}_{\mathrm{outer}}$, the procedure $\mathrm{InferSU}_{\mathrm{inner}}$ is applied to a unique constraint. It computes a set difference between the current search space $[\mathbf{x}]$ and $[\mathbf{y}]$ which is obtained by inner contraction on $\neg c$. The set of resulting SU-boxes $\{\langle[\mathbf{x}'], \mathcal{S}', \mathcal{U}'\rangle\}$ have both their sets of $\mathcal{S}$atisfied and $\mathcal{U}$nknown constraints updated accordingly, *i.e.*, the constraint c is added to the former while being removed from the latter.

To sum up, merging information on constraints satisfaction obtained on the SU-boxes computed through the outer and inner contractions steps is done by applying a set difference. Indeed, when computing these intersections, we can update the sets $\mathcal{S}$ and $\mathcal{U}$ and thus maintain the bounds of the algorithm. Uninteresting SU-boxes will be automatically rejected when processed since they cannot satisfy at least $\underline{m}$ constraints (current lower bound of the Branch and Bound algorithm).

5.4 The function $\mathrm{InferLSU}_{\mathrm{type}}$

Finally, the functions $\mathrm{InferSU}_{\mathrm{outer}}$ and $\mathrm{InferSU}_{\mathrm{inner}}$ are naturally applied to a set of SU-boxes $\mathcal{T}$ in the following way:

$$\mathrm{InferLSU}_{\mathrm{type}}(\mathcal{T}, [\mathbf{y}], c) := \bigcup_{\langle\mathbf{x}\rangle \in \mathcal{T}} \mathrm{InferSU}_{\mathrm{type}}(\langle\mathbf{x}\rangle, [\mathbf{y}], c), \tag{12}$$

where *type* is either *outer* or *inner*. This process is illustrated by Fig. 4. As a direct consequence of Propositions 1 and 2, the following proposition holds:

Proposition 3 *Let* $\mathcal{T}$ *be a set of SU-boxes, each of them being consistent with a CSP* $\mathcal{P}$*, and* $[\mathbf{y}]$ *satisfying* $\forall \mathbf{x} \in \mathit{idiff}([\mathbf{x}], [\mathbf{y}]) \Rightarrow \neg c(\mathbf{x})$ *(respectively* $\forall \mathbf{x} \in \mathit{idiff}([\mathbf{x}], [\mathbf{y}]) \Rightarrow c(\mathbf{x})$*). We define* $\mathcal{T}' := \mathrm{InferLSU}_{\mathrm{outer}}(\langle\mathbf{x}\rangle, [\mathbf{y}], c)$ *(respectively* $\mathcal{T}' := \mathrm{InferLSU}_{\mathrm{inner}}(\langle\mathbf{x}\rangle, [\mathbf{y}], c)$*). As a consequence, the SU-boxes of* $\mathcal{T}'$ *are all consistent with* $\mathcal{P}$*. Furthermore,*

$$\cup\left\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}'\right\} = \cup\left\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\right\}. \tag{13}$$

5.5 The function MultiStartRandomSearch

Like every Branch and Bound method, our algorithm greatly benefits from having a good lower bound on the number of satisfied constraints of the problem. Indeed, the sooner one gets a *good* lower bound $\underline{m}$, the more uninteresting boxes will be rejected during the search. An uninteresting box being a box that cannot satisfy at least $\underline{m}$ constraints. This remark applied to a SU-box meaning that a SU-box $\langle \mathbf{x} \rangle$ can be rejected as soon as $(\#\mathcal{S} + \#\mathcal{U}) < \underline{m}$, which means that $\langle \mathbf{x} \rangle$ will never be able to satisfy $\underline{m}$ constraints. As a direct consequence, those regions of the search space can be rejected because they will not be part of the *solution set* MaxSol($\mathcal{P}$) of the numerical Max-CSP.

The procedure presented in this section is very simple but will greatly improve the performance of our algorithm by allowing the computation of an early good lower bound for the algorithm. Indeed, MultiStartRandomSearch($\mathcal{C}, \mathbf{D}, n$) is a function that inputs the set of constraints $\mathcal{C}$ of the numerical Max-CSP $\mathcal{P}$, the search space $\mathbf{D}$ and a number n of samples that will be generated in order to approximate a first lower bound on the maximum number of satisfied constraints of $\mathcal{P}$. Thus, the function outputs the maximum number of satisfied constraints from the n samples, giving an insight of a lower bound on the number of constraints that can be satisfied.

Algorithm 1 gives the pseudo-code of the MultiStartRandomSearch function as implemented in our method. Each vector $\mathbf{p}$ is obtained thanks to a function GenerateRandomPoint($\mathbf{D}$) that generates a random value in each of the dimensions of the search space $\mathbf{D}$.

Let us recall here that the present algorithm only aims at computing the *outer* and *inner* solution sets of a numerical Max-CSP, the former set ensuring that no solution is lost during the solving process while the latter will be composed of boxes that contain only solutions of the Max-CSP. As a consequence, our algorithm requires the CSPs to be composed of numerical inequalities only.

Algorithm 1: The MultiStartRandomSearch function computes a lower bound $\underline{m}$ for a set of constraints $\mathcal{C}$ in a search space $\mathbf{D}$.

```
  Input: $\mathcal{C}, \mathbf{D}, n$
  Output: $\underline{m}$
1 $\underline{m} \leftarrow 0$;
2 for $i \leftarrow 0$ to $n$ do
3     $\mathbf{p} \leftarrow$ GenerateRandomPoint($\mathbf{D}$);
4     $b \leftarrow \mathrm{sat}_{\mathcal{C}}(\mathbf{p})$;
5     if ( $\underline{m} < b$ ) then $\underline{m} \leftarrow b$;
6 end
7 return $\underline{m}$;
```

Remark 1 When dealing with inequality constraints, $sat_{\mathcal{C}}(\mathbf{p})$ is rigorously computed using the interval extension of the constraint, *i.e.*, $f(\mathbf{p}) \leq 0 \Longleftarrow_{sup} [f]([\mathbf{p}, \mathbf{p}]) \leq 0$.

Section 7.1 highlights the impact of this preprocessing step on the performance of the branch and bound algorithm.

5.6 The branch and bound algorithm

The pseudo-code of our technique is presented in Algorithm 2. Our algorithm computes three sets of SU-boxes: $\mathcal{L}$ (SU-boxes to be processed), $\mathcal{D}$ (SU-boxes where all constraints are decided, *i.e.*, where $(\#\mathcal{U}) = 0$) and $\mathcal{E}$ (SU-boxes that will not be processed anymore because considered as too small). The mainline of the while-loop aims at maintaining the following property:

$$\text{MaxSol}(\mathcal{P}) \subseteq \bigcup \big\{ \text{box}(\langle \mathbf{x} \rangle) \ : \ \langle \mathbf{x} \rangle \in \mathcal{L} \cup \mathcal{D} \cup \mathcal{E} \big\}, \tag{14}$$

while using inner and outer contractions as well as the function $\text{InferLSU}_{\text{type}}$ in order to decide more and more constraints (*cf.* lines 11–17), and hence moving SU-boxes from $\mathcal{L}$ to $\mathcal{D}$ (*cf.* line 21). Note that a lower bound $\underline{m}$ for $m_{\mathcal{P}}$ is updated (*cf.* line 19) and used to drop SU-boxes that are proved to satisfy less constraints than $\underline{m}$ (*cf.* line 14). When $\mathcal{L}$ is finally empty, (14) still holds and hence:

$$\text{MaxSol}(\mathcal{P}) \subseteq \cup \big\{ \text{box}(\langle \mathbf{x} \rangle) : \langle \mathbf{x} \rangle \in \mathcal{D} \cup \mathcal{E} \big\}. \tag{15}$$

Eventually, the sets of SU-boxes $\mathcal{D}$ and $\mathcal{E}$ are used to obtain new sets of SU-boxes $\mathcal{M}$ and $\mathcal{B}$, which describe the max-solution set (*i.e.*, both the inner and outer approximation of the solution set of the numerical CSP), and an interval $[\underline{m}, \overline{m}]$ for $m_{\mathcal{P}}$.

In other words the sets $\mathcal{M}$ and $\mathcal{B}$ encompass the solution set of the numerical Max-CSP $\mathcal{P}$. $\mathcal{M}$ is the set of SU-boxes that have been proved to contain only solutions (*i.e.*, for which $(\#\mathcal{S}) \geq \underline{m})$) while $\mathcal{B}$ ensures not to lose any solutions of the $\mathcal{P}$ by keeping boundary boxes that encompass the border of the solution set, namely SU-boxes such that $(\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}$.

At this stage of the algorithm, we already have a lower bound $\underline{m}$ on $m_{\mathcal{P}}$. Now, as (14) holds, an upper bound can be computed in the following way:

$$\overline{m} = \max \big\{ (\#\mathcal{S}) + (\#\mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} \big\}. \tag{16}$$

Two cases can then happen, which are handled at lines 29–36:

1. If $\underline{m} = \overline{m}$ then the algorithm has succeeded in computing $m_{\mathcal{P}}$, which is equal to $\underline{m} = \overline{m}$. In this case the domains of the SU-boxes of $\mathcal{D}$ are proved to contain only max-solutions of $\mathcal{P}$:

$$\forall \langle \mathbf{x} \rangle \in \mathcal{D} \ , \ \text{box}(\langle \mathbf{x} \rangle) \subseteq \text{MaxSol}(\mathcal{P}). \tag{17}$$

   Furthermore, the max-solutions which are not in $\mathcal{D}$ obviously have to be in the SU-boxes of:

$$\mathcal{B} = \big\{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m} \big\}. \tag{18}$$

   As a consequence, the algorithm outputs both an inner and an outer approximation of MaxSol($\mathcal{P}$).

Algorithm 2: Branch and Bound Algorithm for Numerical Max-CSP.

```
Input: $\mathcal{P} = \langle \mathbf{x}, \mathcal{C}, [\mathbf{x}] \rangle$, $\varepsilon$
Output: $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B})$
1  $\mathcal{L} \leftarrow \{\langle [\mathbf{x}], \emptyset, \mathcal{U} \rangle\}$; /* SU-boxes to be processed */
2  $\mathcal{E} \leftarrow \{\}$; /* Epsilon SU-boxes */
3  $\mathcal{D} \leftarrow \{\}$; /* SU-boxes where each constraint is decided */
4  $\underline{m} \leftarrow$ MultiStartLocalSearch($\mathcal{P}, [\mathbf{x}]$)[3]; /* Lower bound on $\underline{m}$ */
5  while ( $\neg$empty($\mathcal{L}$) ) do
6      $\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \leftarrow$ extract($\mathcal{L}$)[4];
7      if ( wid ($[\mathbf{x}]$) $< \varepsilon$ ) then
8          $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle\}$;
9      else
10         $\mathcal{T} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle\}$;
11         foreach ( $c \in \mathcal{U}$ ) do
12             $[\mathbf{y}] \leftarrow \mathrm{Contract}_c([\mathbf{x}])$;
13             $\mathcal{T} \leftarrow \mathrm{InferLSU}_{\mathrm{outer}}(\mathcal{T}, [\mathbf{y}], c)$;
14             $\mathcal{T} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{T} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
15             $[\mathbf{y}] \leftarrow \mathrm{Contract}_{\neg c}([\mathbf{x}])$;
16             $\mathcal{T} \leftarrow \mathrm{InferLSU}_{\mathrm{inner}}(\mathcal{T}, [\mathbf{y}], c)$;
17         end
18         foreach ( $\langle [\mathbf{x}'], \mathcal{S}', \mathcal{U}' \rangle \in \mathcal{T}$ ) do
19             if ( $(\#\mathcal{S}') > \underline{m}$ ) then $\underline{m} \leftarrow (\#\mathcal{S}')$;
20             if ( $(\#\mathcal{U}') = 0$ ) then
21                 $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle [\mathbf{x}]', \mathcal{S}', \emptyset \rangle\}$;
22             else
23                 $\{[\mathbf{x}^1], [\mathbf{x}^2]\} \leftarrow$ Bisect($[\mathbf{x}']$)[5];
24                 $\mathcal{L} \leftarrow \mathcal{L} \cup \{\langle [\mathbf{x}^1], \mathcal{S}', \mathcal{U}' \rangle, \langle [\mathbf{x}^2], \mathcal{S}', \mathcal{U}' \rangle\}$;
25             end
26         end
27     end
28 end
29 $\overline{m} \leftarrow \max\{(\#\mathcal{S}) + (\#\mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E}\}$;
30 if ( $\underline{m} = \overline{m}$ ) then
31     $\mathcal{M} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} : (\#\mathcal{S}) \geq \underline{m}\}$;
32     $\mathcal{B} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
33 else
34     $\mathcal{M} \leftarrow \emptyset$;
35     $\mathcal{B} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
36 return $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B})$;
```

[3]The constraints satisfaction is tested on uniformly distributed random points, the lower bound $\underline{m}$ being updated accordingly.

[4]SU-boxes are stored in $\mathcal{L}$ thanks to $(\#\mathcal{S}) + (\#\mathcal{U})$, most interesting first.

[5]Bisect splits the box $[\mathbf{x}']$ into 2 equally sized smaller boxes: $[\mathbf{x}^1]$ and $[\mathbf{x}^2]$.

2. If $\underline{m} < \overline{m}$, $m_{\mathcal{P}}$ is still proved to belong to the interval $[\underline{m}, \overline{m}]$. However, the algorithm has not been able to compute any solution of the Max-CSP $\mathcal{P}$, hence $\mathcal{M} = \emptyset$. In this case, all the max-solutions are obviously in the domains of the SU-boxes of:

$$\mathcal{B} = \left\{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m} \right\}. \tag{19}$$

As a consequence of Proposition 3, constraints are correctly removed from $\mathcal{U}$ and possibly added to $\mathcal{S}$ while no solution is lost. This ensures the correctness of the algorithm (a formal proof is not presented here due to its burdensomeness).

5.7 Discussion on the complexity and on the success of the algorithm

The worst case complexity of the foreach-loop at lines 11–17 is exponential with respect to the number of constraints in $\mathcal{U}$. Indeed, the worth case complexity of the loop running through lines 11–17, is $(2n)^q$, where n is the dimension and q is the number of constraints. This upper bound is very pessimistic, and not met in any typical situation. Nonetheless, we expect a possibly high number of boxes generated during the inflated set difference process when the number of constraints is important.

In order to bypass this complexity issue, that could lead the algorithm to generate too many boxes within the outer and inner contractions steps, we have chosen to switch between different contracting operators. As soon as the number of constraints becomes greater than a threshold (t_c), contractions are performed with the *evaluation contractor* instead of the *hull contractor*. This will reduce creation of boxes within the inner and outer contracting steps, thus preventing the algorithm from falling into a complexity pit. Experiments on relatively high number of constraints (*cf.* Section 7.3) have shown that t_c should be set between 5 and 10 in order to speed up the solving process.

The question of the success of the algorithm, *i.e.*, $\underline{m} = \overline{m}$, naturally arises. Even if ε is taken arbitrarily small, there are situations where the algorithm does not succeed. There are typically two cases where this will happen:

1. First, if the max-solution set has no interior, e.g. $\mathcal{P} = \left\langle \mathbf{x}, \{f(\mathbf{x}) \leq 0,\ f(\mathbf{x}) \geq 0\}, \mathbb{R}^n \right\rangle$, with $f(\mathbf{x}) = x_1{}^2 + x_2{}^2 - 1$; for which $m_{\mathcal{P}} = 2$ and $\mathrm{MaxSol}(\mathcal{P}) = \{\mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) = 0\}$.
2. Second, it can happen that even when the solution set has a non empty interior, interval contractors are unable to prove the existence of solutions, e.g. $\mathcal{P} = \left\langle x, \{|x| + x \leq 0\}, \mathbb{R} \right\rangle$ for which $m_{\mathcal{P}} = 1$ and $\mathrm{MaxSol}(\mathcal{P}) = \mathrm{Sol}(\mathcal{P}) = (-\infty, 0\,]$.

Both cases are considered as untypical in the sense that any arbitrarily small perturbation of the constraints make the max-solution set empty. Very different algorithms, like formal techniques, have to be used to solve these untypical situations.

6 Related work

A large literature is related to Max-CSP frameworks and solving techniques. Most formulate the problem as a combinatorial optimization one by minimizing the number of violated constraints. Approaches are then shared between complete and incomplete techniques depending on the optimality criterion and the size of the problem. For large problems, heuristic-based techniques such as Min-conflict [23] and Random-walk [29] have proved to be efficient as well as meta-heuristic approaches (Tabu, GRASP, ACO). However all are restricted to the study of binary or discrete domains, and their extension to continuous values requires to redefine the primitive operations (*e.g.*, move, evaluate, compare, *etc.*). Though some contributions have explored such extensions for local optimization problems (*cf.* for example Continuous-GRASP [16]), these are not dedicated to manage numerical Max-CSPs. More general techniques built upon classical optimization schemes such as Genetic Algorithms, Simulated Annealing or Generalized Gradient can be employed to express and solve Max-CSPs, but do not guarantee the optimality of the solution, nor compute a paving of the search space.

Regardless the fact that applications of CSP techniques might be either in continuous or in discrete domains, both present the same need for Max-CSP models. Interestingly very few contributions have explored this class of problems when optimality is required. Here, we report the contribution of Jaulin et al. [17, 19] that looks at a problem of parameter estimation with bounded error measurements. The paper considers the number of error measurements as known (*i.e.*, $m_{\mathcal{P}}$ is given beforehand) and the technique computes a Max-CSP approximation over a continuous search space by following an incremental process that runs the solving for each possible value of $m_{\mathcal{P}}$ and decides which is the correct one. At each step the technique relies on a classical evaluation-bisection process.

7 Experiments

In order to illustrate the pertinence of our approach, we propose a set of three different benchmarks, ranging from a virtual camera placement (VCP) problem, as described in Section 2 to parameter estimation with bounded error measurements. Furthermore, we wanted to emphasize the flexibility offered by our method, by applying a slightly modified version of our algorithm to a classical Operational Research problem, namely the facility location problem.

Before presenting the results we have obtained, we first detail the usefulness of the simple function MultiStartRandomSearch that aims at giving a first good lower bound on the number of satisfied constraints in a numerical Max-CSP, thus boosting the performance of our method.

7.1 Impact of the MultiStartRandomSearch function

Before presenting the benchmarks of our algorithm, and in order to emphasize the benefits of the function MultiStartRandomSearch presented in Section 5.5, we have realized some experiments on a simple toy problem. The problem consists in generating random 3D balls in a search space $\mathbf{D} = [-20, 20] \times [-20, 20] \times [-20, 20]$ and trying to find the regions of the search space that maximize ball intersections.

Table 1 presents the results obtained when the MultiStartRandomSearch was used in order to get as soon as possible a good lower bound, and second when the first lower bound was set to 0 and updated classically by the algorithm (*cf.* line 19 of the

Table 1 Impact of the MultiStartRandomSearch preprocessing step on the performance of our branch and bound algorithm for numerical Max-CSPs

Benchmark	ε	$\#smp$	$\underline{m}_{\text{MSRS}}$	$[\underline{m}, \overline{m}]$	T_{MSRS} (s)	T_{total} (s)
Balls3D-50	0.5	0	0	[30, 31]	0	256
Balls3D-50	0.5	50	29	[30, 31]	0.007	1.297
Balls3D-50	0.1	0	0	[0, 50]	0	T.O.
Balls3D-50	0.1	50	29	[31, 31]	0.006	11.272
Balls3D-150	0.5	0	0	[85, 87]	0	1412.2
Balls3D-150	0.5	50	85	[85, 87]	0.016	1.219
Balls3D-150	0.1	0	0	[0, 150]	0	T.O.
Balls3D-150	0.1	50	85	[86, 86]	0.012	2.478
Balls3D-500	0.5	0	0	[0, 500]	0	T.O.
Balls3D-500	0.5	50	279	[280, 291]	0.047	7.719
Balls3D-500	0.5	200	283	[283, 291]	0.2	4.053
Balls3D-500	0.1	0	0	[0, 500]	0	T.O.
Balls3D-500	0.1	50	279	[284, 288]	0.045	22.522
Balls3D-500	0.1	200	283	[284, 288]	0.167	8.551

Random 3D balls are generated in $\mathbf{D} = [-20, 20] \times [-20, 20] \times [-20, 20]$ with a minimum bisection size set to ε. Timings obtained on an IntelCore 2 Duo 2.4 GHz laptop 4 GB RAM

Algorithm 2). Updating the lower bound will allow rejecting as many uninteresting boxes as possible during the search, thus increasing the performance of the algorithm.

Different values of random points have been chosen in order to evaluate the impact on the global performance of the algorithm. The timings are represented in seconds as well as the maximum number of satisfied constraints for each problem. The number of balls generated in **D** is denoted after the name of the benchmark, *e.g.*, Balls3D-150 represents a benchmark where 150 random 3D balls have been generated in **D**. The number of random sample points generated is represented in the *#smp* column. If this number is set to 0, then the function is not called and the classical Branch and Bound algorithm is applied without preliminary searching for a good lower bound. Finally, the lower bound found by the MultiStartRandomSearch method is given in the $\underline{m}_{\text{MSRS}}$ column, while the interval $[\underline{m}, \overline{m}]$ of satisfied constraints computed by the algorithm is represented in the eponym column. Timings are listed in the last two columns, total and time spent in MultiStartRandomSearch (T_{MSRS}) are represented in seconds.

Results (*cf.* Table 1) show significant improvements when using the function MultiStartRandomSearch. Indeed while the original algorithm fails at solving a number of benchmarks (T.O. stands for time out, which results from a saturation of the memory of the computer), one can observe that the preprocessing step greatly improves the results. Of course, the number of sample points (*#smp*) that should be generated in order to get a good lower bound is problem dependent. Results only aim at emphasizing the usefulness of such a preprocessing step. As a consequence, while the first two problems are solved with our algorithm (*i.e.*, $\underline{m} = \overline{m}$), the benchmark Random-Balls3D-500 should be tackled with a lower minimum bisection size ε in order to solve it. Indeed Table 1 shows that our algorithm has only computed an outer approximation of the solution set of the Random-Balls3D-500 benchmark.

The simple procedure presented here (evaluation of the constraints on random points in the search space) could obviously be improved by adding a local search procedure. However, such random search for Max-CSP in continuous domains is, to the best authors knowledge, yet to be proposed.

The authors are aware that generating random points in the search space is certainly not the best heuristic when tackling continuous problems, nevertheless this straightforward improvement has shown interesting results. As future work, the authors will consider replacing this procedure by state of the art continuous local optimizers such as MINOS [25] or KNITROS [6] for example.

Those solvers will certainly prove to be much more efficient than our simple procedure, however it could be interesting to study the impact of those solvers on the global performance of our algorithm. The overhead due to their use could reduce the benefits provided by the MultiStartRandomSearch procedure. Indeed, the aim of this procedure is not to find an optimum value for the number of satisfied constraints, but rather to be able to find a relatively good bound on this number in order to reject as many boxes as possible during the exhaustive search performed by our algorithm.

7.2 Camera control problem

This subsection is dedicated to the virtual camera placement (VCP) example presented in Section 2. Let us recall briefly that this motivating example involves 2 variables and 6 constraints expressed as dot products and Euclidean distances in 2D. The constraints involved in the numerical Max-CSP $\mathcal{P}$ are defined over three objects A, B and C abstracted as 2D points representing $(x_A, y_A, x_B, y_B, x_C, y_C)$ their positions in the scene and three 2D vectors $(x_{OA}, y_{OA}, x_{OB}, y_{OB}, etc.)$ representing their orientations. The variables of $\mathcal{P}$ are the camera 2D position, namely x and y. Finally, the acceptable distances are denoted by r_A, r_B and r_C while accepted orientations are represented by the values of the dot products $\alpha_A, \alpha_B, \alpha_C$. The constraints of $\mathcal{P}$ are thus the following:

$$c_1 : \sqrt{(x - x_A)^2 + (y - y_A)^2} \leq r_A$$

$$c_2 : \sqrt{(x - x_B)^2 + (y - y_B)^2} \leq r_B$$

$$c_3 : \sqrt{(x - x_C)^2 + (y - y_C)^2} \leq r_C$$

$$c_4 : \frac{(x - x_A) * x_{OA} + (y - y_A) * y_{OA}}{\sqrt{(x - x_A)^2 + (y - y_A)^2)}} \geq \alpha_A$$

$$c_5 : \frac{(x - x_B) * x_{OB} + (y - y_B) * y_{OB}}{\sqrt{(x - x_B)^2 + (y - y_B)^2)}} \geq \alpha_B$$

$$c_6 : \frac{(x - x_C) * x_{OC} + (y - y_C) * y_{OC}}{\sqrt{(x - x_C)^2 + (y - y_C)^2)}} \geq \alpha_C$$

The corresponding paving is shown in Fig. 5 while results are presented in Table 2 within the search space $\{[-30; 30], [1; 30]\}$ and with $\varepsilon = 0.01$. The algorithm outputs

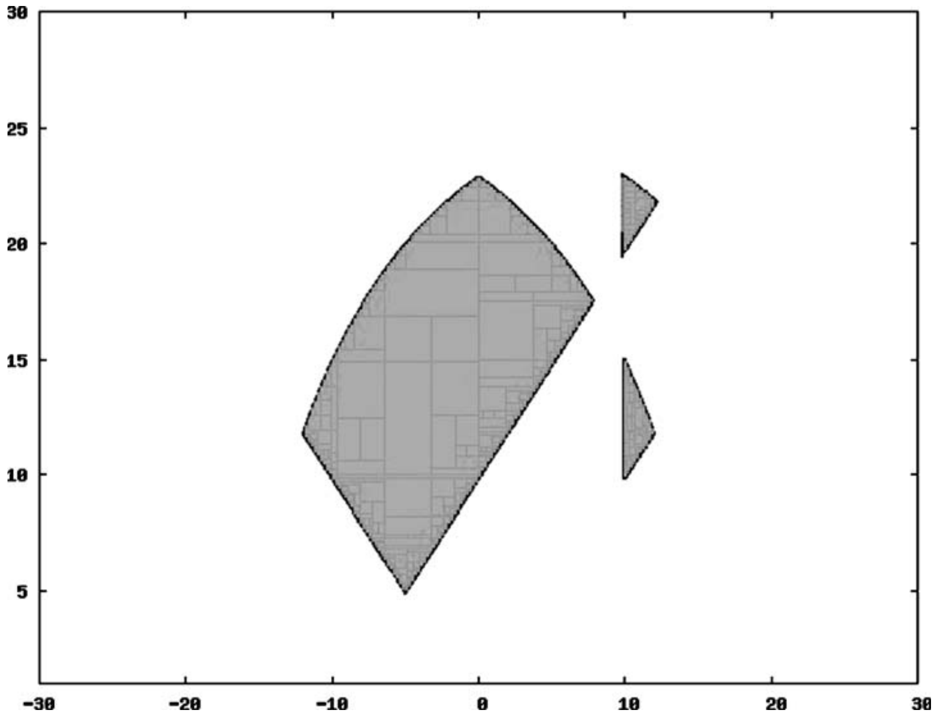

Fig. 5 Pavings generated by our algorithm on the VCP problem. *Grey boxes* represent the max solution set (*cf.* Fig. 1), while *black boxes* illustrate boundary boxes encompassing the solution set

$\underline{m} = \overline{m} = 5$ meaning that the set of inner boxes satisfy five constraints out of six, let us recall that the problem is over-constrained as shown in Fig. 1.

Although Christie et al. [8] do not compute an outer and and inner approximation of the solution set of the Max-CSP (they only aim at computing an inner approximation), their approach is similar enough to compare timings obtained on the VCP example.

As discussed in Section 5.7, this example illustrates the powerfulness of the gathering of the solution sets in connected components. Indeed Fig. 5 shows that the search space is divided into three connected components each encompassing a different set of equivalent solutions. Presenting the user such useful information on the solution sets might help him improving the modeling of his problem or be able to chose solutions that fulfill specific constraints interesting for the user.

Table 2 Approximated timings thanks to [12] for the VCP example between our approach (PentiumM750 laptop, 1GB RAM) and Christie et al. [8] (Pentium T7600 2.33Ghz 2GB RAM)

Method	ε	T_{original} (seconds)	T_{approx} (seconds)
Numerical Max-CSP	0.01	3.2	3.2
Christie et al.	0.01	4.8	≈ 15

7.3 The facility location problem

This example is based on a well-known geometric problem in Operational Research: the facility location problem [13]. The problem is defined by a number of customer locations that must be delivered by some facilities. We consider that a customer c can be served by a facility f if the distance between f and c is smaller than a delivery distance d. The locations of the customers are inputs of the facility location problem, which thus consists in positioning the smallest number of facilities such that each customer can be served by at least one facility.

Strictly speaking, the facility location problem in Operational Research is not exactly posed as this benchmark. Indeed, the original problem is defined as a combinatorial one, that is knowing a set of facilities locations and a set of customers locations, which facilities should be "opened" in order that at least every customer is delivered by a facility. Nevertheless, the authors believe that this slightly modified version of the facility location problem is realistic enough to be used in our set of experiments.

Although the facility location problem is not strictly speaking a Max-CSP, we can use the Max-CSP framework to build a greedy algorithm to find an estimation of the number of facilities. The idea consists in computing the Max-CSP areas of the problem which corresponds to the areas of the search space that maximizes customers delivery. Instead of computing the whole MaxSol paving, we stop the algorithm as soon as we have found a MaxSol SU-box $\langle[\mathbf{x}_m], \mathcal{S}_m, \mathcal{U}_m\rangle$. We then remove from the original problem all the constraints of $\mathcal{S}_m$ (which corresponds to all the customer that have been delivered during the iteration) and re-run the algorithm again. We stop when the problem is empty, *i.e.*, when all the constraints have been solved. The numbers of runs of the algorithm then correspond to a number of facilities that solves the problem. Of course this is a quite naive approach but it is very easy to implement thanks to our algorithm. Our greedy implementation might not for now be as effective as OR approaches but it could be improved in order to reduce the number of runs needed to solve the problem.

In order to test our greedy algorithm, we have implemented a basic OR technique to solve the same problem (*i.e.*, discretization of the search space and resolution of a linear integer programming problem) in Wolfram Mathematica.[6]

Indeed, classical OR algorithms compute the intersection areas of all deliverable areas around the customers (*e.g.*, circles in 2D) before using a linear programming algorithm to solve the problem. We believe that the higher the dimensions of the problem, the more difficult the intersection computation will be and the more our algorithm could compete the OR approach.

The classical facility location problem is, to the best of the authors knowledge, only defined in 2D in Operational Research. Nonetheless, in order to challenge our hypothesis that the greater the number of dimensions, the better our algorithm will perform, we propose a 3D version of the facility location benchmark. The definition of the problem remains unchanged, only instead of trying to position the facilities in a 2D space, we propose to try to find their coordinates in a 3D space.

[6]The corresponding Mathematica notebook is available for download at: http://www.goldsztejn.com/src/FacilityLocation3D.nb.

Table 3 Timings for the facility location problem for 500 customers randomly generated in $\{[-5;5],[-5;5],[-5;5]\}$ on a PentiumM750 laptop, 1GB RAM

Method	Time (seconds)	Number of facilities
Greedy numerical Max-CSP	69.25	39
Basic OR technique	≈ 140	45

In 3D this problem boils down to modeling each customer by a random 3D position in the search space $\{[-5;5],[-5;5],[-5;5]\}$ and trying to position some facilities such that the distance between the facility and the customer is inferior to $d = 2$. Results are shown in Table 3. The solution found by our algorithm involves 3×39 variables (3 coordinates per computed facility). Each iteration of our algorithm removes on average 13 constraints (ranging from 33 to 1 constraint removed at each iteration) of the 500 original ones.

Although, this basic OR technique seems less efficient than our greedy algorithm, this is at least encouraging and thorough experiments need to be carried out to compare our greedy approach with more specific OR techniques.

7.4 Parameter estimation with bounded error measurements

This experiment is based on a parameter estimation problem presented in [19]. The problem consists in determining unknown parameters $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2}) \in ([\mathbf{-0.1}, \mathbf{1.5}], [\mathbf{-0.1}, \mathbf{1.5}])$ of a physical model thanks to a set of experimental data obtained from a system. The physical model is defined as follows and has been taken from [17, 19, 22]. The vector of original available data $\mathbf{y}$ is:

$$\mathbf{y} = \left(7.39, 4.09, 1.74, 0.097, -2.57, -2.71, -2.07, -1.44, -0.98, -0.66\right)^T$$

It corresponds to ten scalar values taken at times:

$$\mathbf{t} = \left(0.75, 1.5, 2.25, 3, 6, 9, 13, 17, 21, 25\right)^T$$

But here in order to simulate outliers, Jaulin et al. [19] arbitrarily replaced two data points by zero values and thus the array of data becomes:

$$\mathbf{y} = \left(7.39, 0, 1.74, 0.097, -2.57, -2.71, -2.07, 0, -0.98, -0.66\right)^T$$

We assume that these data are to be described by a vector $\mathbf{y}_m \in \mathbb{R}^{n_y}$ described by a model with a fixed structure but unknown parameter vector $\mathbf{p} \in \mathbb{R}^{n_p}$. The purpose of parameter estimation, is to find $\mathbf{p}$ such that $\mathbf{y}_m(\mathbf{p})$ fits $\mathbf{y}$ best. In this experiment, we are talking of bounded-error estimation, thus we consider that the parameters $\mathbf{p}$ are considered admissible if the error $\mathbf{e}(\mathbf{p})$, defined as:

$$\mathbf{e}(\mathbf{p}) = \mathbf{y} - \mathbf{y}_m(\mathbf{p})$$

belongs to some set of admissible errors.

Here we consider the physical system being defined (*cf.* [19]) such that the ith component of $\mathbf{y}_m(\mathbf{p})$ is given by:

$$\mathbf{y}_{m,i}(\mathbf{p}) = 20\exp(-p_1 t_i) - 8\exp(-p_2 t_i)$$

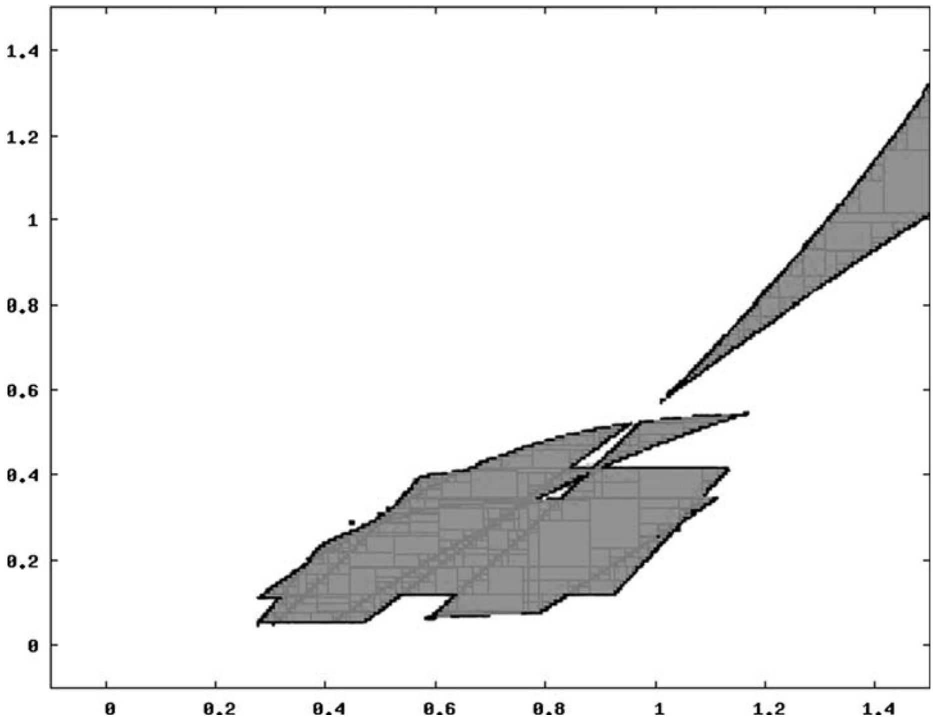

Fig. 6 Paving generated for 3 possible outliers. *Grey boxes* represent inner approximation (*i.e.*, satisfying at least 7 of the 10 constraints) while *black boxes* illustrate boundary boxes encompassing the solution set

The set of all feasible model outputs is the box defined by:

$$\mathbf{y} = \left[\mathbf{y} - \mathbf{e}_{max}, \mathbf{y} + \mathbf{e}_{max}\right]$$

with

$$\mathbf{e}_{max} = \left(4.695, 1., 1.87, 1.0485, 2.285, 2.355, 2.035, 1., 1.49, 1.33\right)^T$$

Here the model used is a two-parameter estimation problem taken from Jaulin et al. [17, 19] which is a 2D extension of a problem presented by Milanese and Vicino [22]. In fact, the solution set of the Max-CSP does not contain the true parameters values. Indeed, the maximum number of satisfied constraints is 9, while there are 2 outliers and the model contains 10 constraints (corresponding to the vectors given above). This means that an outlier is compatible with the error measurements. Hence, the Max-CSP cannot directly help solving this problem. From

Table 4 Original and approximate timings obtained from a Pentium M750 laptop processor (Numerical Max-CSP) and from a 486 DX4-100 processor (Jaulin et al.) thanks to [12]

Method	ε bisection	T_{original} (seconds)	T_{approx} (seconds)
Numerical Max-CSP	0.005	0.46	0.46
Jaulin et al.	0.005	≈ 180	≈ 1.9

an application point of view, it is possible to provide an upper bound on the number of outliers (this assumption is realistic since manufacturers are able to ensure a percentage of maximum failures of their probes). Algorithm 2 is easily modified to output the set of vectors which satisfy at least a fixed number of constraints instead of the solution set of the Max-CSP.

In order to compare our approach with [19], we set a maximum of 3 outliers, Fig. 6 shows the corresponding paving. Although Jaulin et al. do not compute an inner and outer approximation of the solution set, Table 4 compares our results. Approximate timings have been computed using [12] in order to compare the results that were obtained on totally different computers. Time comparisons are presented to give an idea of the gain offered by our algorithm, which is approximately 4 times more efficient (although no exact comparison of the pavings computed by the two algorithms is possible, the pavings presented in [19] are clearly less accurate than the pavings computed by our approach). Moreover, we want to emphasize the fact that our algorithm is much more scalable since Jaulin et al. have to re-run their algorithm for each possible number of outliers whereas we only need one run to compute the solution set of the Max-CSP. Moreover our approach fully characterize this solution set w.r.t. the satisfied constraints of the algorithm.

8 Conclusion

In this paper we have presented a branch and bound algorithm for numerical Max-CSP that computes both an inner and an outer approximation of the solution set of a numerical Max-CSP. The algorithm is based on the notion of SU-box that stores the sets of constraints $\mathcal{S}$atisfied or $\mathcal{U}$nknown for each box of the search space. The algorithm outputs a set of boxes that encompass the inner and outer approximation of the solution set of a numerical Max-CSP. For each box, it computes the sets of constraints that are satisfied, thus giving the user additional information on the classes of solutions of the problem. An algorithm of connected component identification could then be applied on the solution SU-boxes in order to create regions of the search space that share similar characteristics (*i.e.*, that satisfies the same constraints). Users could thus benefit from this feature when modeling conceptual design problems for example.

In continuous domains, a CSP is generally defined as over-constrained when no solution can be found for it. Conversely, a CSP is said to be under-constrained when the number of solutions to the constraint set is infinite. Finally, a CSP is well constrained when the number of solutions of the constraint set is finite. The Branch and Bound algorithm presented in this paper has been designed to solve over-constrained problem. Nonetheless, our method can address any kind of CSP: well-constrained, under-constrained and over-constrained. Indeed, it is often a problem in itself to determine whether a CSP is well-constrained, over-constrained or under-constrained, this is the reason why our algorithm has been designed to be able to solve any kind of CSP, of course at the price of some overhead induced by additional computations.

Indeed, if the CSP is well-constrained, the only difference for our B&B algorithm is that the maximum number of satisfied constraints is equal to the number of constraints of the CSP. The whole solving process remains the same. As for the under-constrained case, since we never reject boxes unless we are certain that a box is better (*i.e.*, that is satisfies at least one more constraint), we are able to output the set of solutions to the CSP.

A drawback of our algorithm is the possible combinatorial explosion created within the inner and outer contractions steps. Indeed a set difference is computed between all the boxes created by inner and outer contractions applied to each constraint of the problem. This leads to poor performance when the number of constraints of the problem becomes too important. In order to bypass this limitation, we have modified our algorithm in order to perform the inner and outer contractions steps only when the number of constraints is not too high. This is achieved by testing the number of constraints contained within the studied SU-boxes. If this number is higher than an arbitrary threshold (t_c) we choose to use a simple box evaluation procedure instead of inner and outer contractors. A problem naturally arises on how to define this threshold, experiments have shown that for a problem consisting of 100 constraints this threshold could be set around 5–10 in order to speed the solving process.

Our algorithm could be extended to manage hierarchical constraints in a predicate *locally-better* way [5]. Hierarchical constraints could be modeled as different "layers" of numerical Max-CSP problems that could be solved in sequence, starting from the top-priority constraints down to the lowest ones and maximizing constraints satisfaction of each "layer" of hierarchical constraints.

Finally, an advantage of our algorithm is that each SU-box contains the set of constraints that are fulfilled within this box. We are thus able to give the user information on which constraint is fulfilled in each part of the search space, allowing him to choose between the different solutions of the over-constrained problem. An automatic connected component identification algorithm (*cf.* [10]) could be processed after solving a problem with our algorithm in order to present the user the sets of equivalent SU-boxes, allowing him to choose which constraints he wants to relax into the original problem.

Another perspective we are currently working on lies in the use of the *q-intersection* algorithm introduced by Jaulin and Walter [18], which is able to compute a set of boxes (Cartesian products of domains) that lies in the intersection of q boxes.

Acknowledgement The authors are grateful to the anonymous referees whose comments significantly improved the paper.

References

1. Benhamou, F., Goualard, F., Granvilliers, L., & Puget, J.-F. (1999). Revising hull and box consistency. In *Proceedings of the 1999 international conference on logic programming, ICLP'99* (pp. 230–244). Cambridge, MA, USA: Massachusetts Institute of Technology.
2. Benhamou, F., Goualard, F., Languenou, E., & Christie, M. (2004). Interval constraint solving for camera control and motion planning. *ACM Transactions on Computational Logic, 5*(4), 732–767.
3. Benhamou, F., McAllester, D., & van Hentenryck, P. (1994). CLP(intervals) revisited. In *International logic programming symposium, ILPS '94* (pp. 124–138). Cambridge: MIT.
4. Benhamou, F., & Older, W. J. (1997). Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming, 32*(1), 1–24.
5. Borning, A., Freeman-Benson, B., & Wilson, M. (1992). Constraint hierarchies. *Lisp and Symbolic Computation, 5*(3), 223–270.

6. Byrd, R. H., Nocedal, J., & Waltz, R. A. (2006). KNITRO: An integrated package for nonlinear optimization. In *Large-scale nonlinear optimization* (pp. 35–59). New York: Springer.
7. Christie, M., & Normand, J.-M. (2005). A semantic space partitionning approach to virtual camera control. In *Proceedings of the annual eurographics conference* (Vol. 24, pp. 247–256).
8. Christie, M., Normand, J.-M., & Truchet, C. (2006). Computing inner approximations of numerical MaxCSP. In *Interval analysis, constraint propagation, applications, IntCP 2006*.
9. Collavizza, H., Delobel, F., & Rueher, M. (1999). Extending consistent domains of numeric CSP. In *IJCAI '99: Proceedings of the sixteenth international joint conference on artificial intelligence* (pp. 406–413).
10. de Givry, S., Larrosa, J., Meseguer, P., & Schiex, T. (2003). Solving Max-SAT as weighted CSP. In *Principles and practice of constraint programming, CP 2003* (pp. 363–376). New York: Springer.
11. Delanoue, N., Jaulin, L., & Cottenceau, B. (2004). Counting the number of connected components of a set and its application to robotics. In *PARA* (pp. 93–101).
12. Dongarra, J. (2007). *Performance of various computers using standard linear equations software*. Technical report CS-89-85, University of Tennessee.
13. Drezner, Z., & Hamacher, H. W. (Eds.) (2002). *Facility location. Applications and theory*. New York: Springer.
14. Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys, 23*(1), 5–48.
15. Hayes, B. (2003). A lucid interval. *American Scientist, 91*(6), 484–488.
16. Hirsch, M. J., Meneses, C. N., Pardalos, P. M., & Resende, M. G. C. (2007). Global optimization by continuous grasp. *Optimization Letters, 1*, 201–212.
17. Jaulin, L., & Walter, E. (1993). Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. *Mathematics and Computers in Simulation, 35*(2), 123–137.
18. Jaulin, L., & Walter, E. (2002). Guaranteed robust nonlinear minimax estimation. *IEEE Transaction on Automatic Control, 47*(11), 1857–1864.
19. Jaulin, L., Walter, E., & Didrit, O. (1996). Guaranteed robust nonlinear parameter bounding. In *CESA'96, IMACS multiconference (symposium on modelling, analysis and simulation)* (pp. 1156–1161).
20. Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *IJCAI '93: Proceedings of the thirteenth international joint conference on artificial intelligence* (pp. 232–238).
21. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence, 8*(1), 99–118.
22. Milanese, M., & Vicino, A. (1991). Estimation theory for nonlinear models and set membership uncertainty. *Automatica, 27*(2), 403–408.
23. Minton, S., Johnston, M. D., Philips, A. B., & Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence, 58*(1–3), 161–205.
24. Moore, R. E. (1966). *Interval analysis*. Englewood Cliffs: Prentice-Hall.
25. Murtagh, B. A., & Saunders, M. A. (1998). *Minos 5.5 user's guide*. Technical report, Systems Optimization Laboratory, Department of Operations Research, Stanford University.
26. Neumaier, A. (1990). *Interval methods for systems of equations*. Cambridge: Cambridge University Press.
27. Normand, J.-M. (2008). *Placement de caméra en environnements virtuels*. PhD thesis, Université de Nantes.
28. Petit, T., Régin, J.-C., & Bessière, C. (2002). Range-based algorithm for Max-CSP. In *Principles and practice of constraint programming, CP 2002* (pp. 280–294). New York: Springer.
29. Wallace, R. J. (1996). Analysis of heuristic methods for partial constraint satisfaction problems. In *Principles and practice of constraint programming, CP 1996* (pp. 482–496). New York: Springer.