A Branch and Bound Algorithm for Numerical MAX-CSP

Jean-Marie Normand[1], Alexandre Goldsztejn[1,2], Marc Christie[1,3], and Frédéric Benhamou[1]

[1] University of Nantes, LINA UMR CNRS 6241
[2] CNRS, France
[3] INRIA Rennes Bretagne-Atlantique
firstname.lastname@univ-nantes.fr

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 maximise constraint satisfaction. 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 extension. 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 computes under and over approximations of its solution set and determines the maximum number $m_{\mathcal{P}}$ of satisfied constraints. The technique is applied on three numeric applications and provides promising results.

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 maximise constraint satisfaction. These approximations ensure that the

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 205–219, 2008.

inner approximation contains only solutions of the Max-CSP while the outer approximation guarantees that no solution is lost during the solving process.

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 defined on the objects of a virtual scene.

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 one, or to remodel the problem.

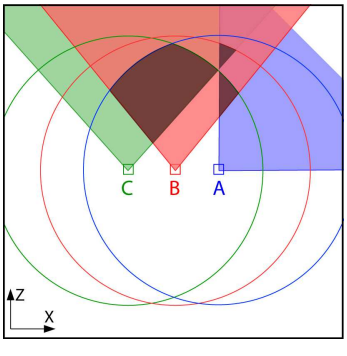

Fig. 1. Top-view of a 3D scene representing camera's position search space for a VCP problem. Orientations of objects are represented by colored areas, *shot* distances are represented as circles. Dark areas represent the max-solution regions of the over-constrained problem.

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.

In Figure 1, a sample scene is displayed with three characters, 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. The solution of the Max-CSP are represented as dark areas. The solving process associated to this simple example is provided in section 7.1.

3 The Max-CSP Framework

This section is dedicated to the Max-CSP framework and its related concepts. According to [1,2], a Max-CSP $\mathcal{P}$ is a triplet $\langle V, \mathbf{D}, C\rangle$ where $V = \{x_1, \ldots, x_n\}$, $\mathbf{D} = \{D_1, \ldots, D_n\}$ 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 $\mathbf{D}$ is interpreted as the Cartesian product of the variables domains. To define solutions of a Max-CSP, a function s_i is first defined for each constraint c_i as follows: $\forall \mathbf{x} \in \mathbf{D}, s_i(\mathbf{x}) = 1$ if $c_i(\mathbf{x})$ is true and $s_i(\mathbf{x}) = 0$ otherwise. Then, the maximum number of satisfied constraint is denoted by $m_{\mathcal{P}}$ and formally defined as follows:

$$m_{\mathcal{P}} := \max_{\mathbf{x} \in D} \sum_i s_i(\mathbf{x}). \tag{1}$$

We denote MaxSol($\mathcal{P}$) the *solution set* of the numerical Max-CSP. Formally, the following set of solution vectors:

$$\text{MaxSol}(\mathcal{P}) = \Big\{\mathbf{x} \in D : \sum_i s_i(\mathbf{x}) = m_{\mathcal{P}}\Big\}. \tag{2}$$

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.* [3,4]) is a key framework in this context: soundness and completeness properties of real values belonging to some given intervals can be ensured 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 representable bounds 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 $\underline{x}, \overline{x} \in \mathbb{F}$. The set of all floating-point intervals is denoted by $\mathbb{IF}$. 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* having the *containment property* (see [5]).

Definition 1 (Interval extension [3]). *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*

$$[f]([\mathbf{x}]) \supseteq \{f(\mathbf{x}) : \mathbf{x} \in [\mathbf{x}]\}. \tag{3}$$

The 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 [5] states that the interval evaluation of an expression gives rise to an interval extension of the corresponding real function: For example

$$[x] + \exp([x] + [y]) \supseteq \{x + \exp(x + y) : x \in [x], y \in [y]\}. \tag{4}$$

4.2 Interval Contractors

Discarding all inconsistent values from a box of variables domains is intractable when the constraints are real ones. There exists many techniques to obtain contractors, like 2B consistency (also called hull consistency) [6,7], box consistency [8] or interval Newton operator [3]. 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 [9].

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

$$\mathbf{x} \in [\mathbf{x}] \wedge c(\mathbf{x}) \implies \mathbf{x} \in \mathrm{Contract}_c([\mathbf{x}]). \tag{5}$$

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 f

$$\mathrm{Contract}_{f(\mathbf{x})\leq 0}(\mathbf{x}) = \begin{cases} \emptyset & \text{if } [f]([\mathbf{x}]) > 0 \\ [\mathbf{x}] & \text{otherwise.} \end{cases} \tag{6}$$

The second, called the *hull contractor*, is based on the hull consistency, *cf.* [6,7] for details[1].

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* as follows: $\mathsf{idiff}([\mathbf{x}],[\mathbf{y}])$ is 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 in Figure 2. In this way, $\mathsf{idiff}([\mathbf{x}],\mathrm{Contract}_{f(\mathbf{x})\leq 0}([\mathbf{x}]))$ constructs a set of boxes which are all proved to contain no solution. Following [10,11] this technique can also be used to compute boxes that contain only solutions, applying the contractor to the negation of the constraint: $\mathsf{idiff}([\mathbf{x}],\mathrm{Contract}_{f(\mathbf{x})>0}([\mathbf{x}]))$ is a set of boxes that contain only solutions. Let us illustrate this process on a simple example. Consider the CSP $\langle x,[-1,1],x\leq 0\rangle$. Then $\mathrm{Contract}_{x\leq 0}([-1,1])=[-1,0]$ and $\mathsf{idiff}([-1,1],[-1,0])=[0^{\oplus},1]$, where $0^{\oplus}$ is the smallest representable number bigger than 0. The constraint $x\leq 0$ is therefore proved to be false inside $[0^{\oplus},1]$. The domain $[-1,0^{\oplus}]$ remains to be processed, and we look for solutions therein by computing $\mathrm{Contract}_{x>0}([-1,0^{\oplus}])=[0,0^{\oplus}]$. We use again $\mathsf{idiff}([-1,0^{\oplus}],[0,0^{\oplus}])=[-1,0^{\ominus}]$ (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 $[0^{\ominus},0^{\oplus}]$.

5 The Algorithm

The main idea of the algorithm is to attach to each box 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 ($\mathcal{U}$). Then, using inner or outer contractions and bisections, we can split the box into smaller boxes where more constraints are decided. We thus propose the definition of SU-boxes representing a triplet consisting of a box $[\mathbf{x}]$ and the two sets of constraints described above. The formal definition of a SU-box is given in the first subsection and the usage of outer and inner contractions is described in the next three subsections. Finally the branch and bound algorithm for numerical Max-CSP is given 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 in $[\mathbf{x}]$ and the constraints which remain to be treated. Such a box, altogether with the satisfaction information of each constraints 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 CSP:

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

[2] *SU-boxes* have been introduced in [12] in a slightly but equivalent way.

Algorithm 1. Branch and Bound Algorithm for MAX-CSP

Input: $\mathcal{P} = \langle \mathbf{x}, \mathcal{C}, [\mathbf{x}] \rangle$, ϵ
Output: $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B})$

$\mathcal{L} \leftarrow \{\langle [\mathbf{x}], \emptyset, \mathcal{U} \rangle\}$; /* SU-boxes to be processed */
$\mathcal{E} \leftarrow \{\}$; /* Epsilon SU-boxes */
$\mathcal{D} \leftarrow \{\}$; /* SU-boxes where every constraint is decided */
$\underline{m} \leftarrow \text{MultiStartLocalSearch}(\mathcal{P}, [\mathbf{x}])$[3]; /* Lower bound on max */
while ($\neg$empty($\mathcal{L}$)) **do**
  $\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \leftarrow \text{extract}(\mathcal{L})$[4];
  if ($\text{wid}([\mathbf{x}]) < \epsilon$) **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle\}$;
  else
    $\mathcal{T} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle\}$;
    foreach ($c \in \mathcal{U}$) **do**
      $[\mathbf{y}] \leftarrow \text{Contract}_c([\mathbf{x}])$;
      $\mathcal{T} \leftarrow \text{InferLSU}_{\text{outer}}(\mathcal{T}, [\mathbf{y}], c)$;
      $\mathcal{T} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{T} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
      $[\mathbf{y}] \leftarrow \text{Contract}_{\neg c}([\mathbf{x}])$;
      $\mathcal{T} \leftarrow \text{InferLSU}_{\text{inner}}(\mathcal{T}, [\mathbf{y}], c)$;
    end
    foreach ($\langle [\mathbf{x}'], \mathcal{S}', \mathcal{U}' \rangle \in \mathcal{T}$) **do**
      if ($(\#\mathcal{S}') > \underline{m}$) **then** $\underline{m} \leftarrow (\#\mathcal{S}')$;
      if ($(\#\mathcal{U}') = 0$) **then**
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle [\mathbf{x}]', \mathcal{S}', \emptyset \rangle\}$;
      else
        $\{[\mathbf{x}^1], [\mathbf{x}^2]\} \leftarrow \text{Bisect}([\mathbf{x}'])$[5];
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{\langle [\mathbf{x}^1], \mathcal{S}', \mathcal{U}' \rangle, \langle [\mathbf{x}^2], \mathcal{S}', \mathcal{U}' \rangle\}$;
      end
    end
  end
end
$\overline{m} \leftarrow \max\{(\#\mathcal{S}) + (\#\mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E}\}$;
if ($\underline{m} = \overline{m}$) **then**
  $\mathcal{M} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} : (\#\mathcal{S}) \geq \underline{m}\}$;
  $\mathcal{B} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
else
  $\mathcal{M} \leftarrow \emptyset$;
  $\mathcal{B} \leftarrow \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}$;
return $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B})$;

[6] Constraints satisfaction is considered on uniformly distributed random points in $[\mathbf{x}]$, the lower bound $\underline{m}$ being updated accordingly.
[7] SU-boxes are stored in $\mathcal{L}$ according to $(\#\mathcal{S}) + (\#\mathcal{U})$, most interesting (*i.e.*, highest $(\#\mathcal{S}) + (\#\mathcal{U})$) being extracted first.
[8] Bisect splits the box $[\mathbf{x}']$ into two equally sized smaller boxes $[\mathbf{x}^1]$ and $[\mathbf{x}^2]$.

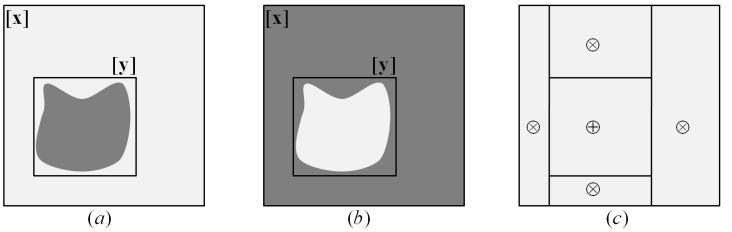

Fig. 2. Diagrams (a) and (b) show the constraint c (satisfied in dark gray regions and unsatisfied in light gray regions), the initial SU-box $\langle[\mathbf{x}],\mathcal{S},\mathcal{U}\rangle$ and the contracted box $[\mathbf{y}]$ for two different situations: (a) outer contraction and (b) inner contraction for constraint c. Diagram (c): Five new SU-boxes 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.

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 called the *domain* of the SU-box.

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 computing $[\mathbf{y}] = \text{Contract}_c([\mathbf{x}])$. Therefore, every box of $\mathsf{idiff}([\mathbf{x}],[\mathbf{y}])$ does not contain any solution of c. This is illustrated by Figure 2. The SU-boxes formed using these boxes can thus be improved discarding c from their set of constraint $\mathcal{U}$. 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:*

$$\{\langle[\mathbf{x}'],\mathcal{S},\mathcal{U}'\rangle : [\mathbf{x}']\in \mathit{idiff}([\mathbf{x}],[\mathbf{y}])\}\cup\{\langle[\mathbf{y}]^{\oplus}\cap[\mathbf{x}],\mathcal{S},\mathcal{U}\rangle\}, \tag{7}$$

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* $\mathbf{x} \in ([\mathbf{x}]\backslash[\mathbf{y}]) \Rightarrow \neg\, c(\mathbf{x})$. *Define* $\mathcal{T} := \mathrm{InferSU}_{\mathrm{outer}}(\langle\mathbf{x}\rangle,[\mathbf{y}],c)$. *Then the SU-boxes of* $\mathcal{T}$ *are all consistent with* $\mathcal{P}$. *Furthermore,* $\cup\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\}$ *is equal to* $[\mathbf{x}]$.

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

This function is the same as $\mathrm{InferSU}_{\mathrm{outer}}$, but dealing with inner contraction. As a consequence, the only difference is that, instead of just discarding the constraint c from $\mathcal{U}$ for the SU-boxes outside $[\mathbf{y}]$, this constraint is stored in $\mathcal{S}$. Figure 2 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:*

$$\{\langle[\mathbf{x}'],\mathcal{S}',\mathcal{U}'\rangle : [\mathbf{x}'] \in \mathit{idiff}([\mathbf{x}],[\mathbf{y}])\} \;\cup\; \{\langle[\mathbf{y}]^{\oplus} \cap [\mathbf{x}],\mathcal{S},\mathcal{U}\rangle\}, \tag{8}$$

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* $\mathbf{x} \in ([\mathbf{x}]\backslash[\mathbf{y}]) \Rightarrow c(\mathbf{x})$. *Define* $\mathcal{T} := \mathrm{InferSU}_{\mathrm{inner}}(\langle\mathbf{x}\rangle,[\mathbf{y}],c)$. *Then the SU-boxes of* $\mathcal{T}$ *are all consistent with* $\mathcal{P}$. *Furthermore,* $\cup\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\}$ *is equal to* $[\mathbf{x}]$.

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:

$$\mathrm{InferLSU}_{\mathrm{type}}(\mathcal{T},[\mathbf{y}],c) := \bigcup_{\langle\mathbf{x}\rangle\in\mathcal{T}} \mathrm{InferSU}_{\mathrm{type}}(\langle\mathbf{x}\rangle,[\mathbf{y}],c), \tag{9}$$

where *type* is either *outer* or *inner*. This process is illustrated by Figure 3. 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* $\mathbf{x} \in ([\mathbf{x}]\backslash[\mathbf{y}]) \Rightarrow \neg\, c(\mathbf{x})$ *(respectively* $\mathbf{x} \in ([\mathbf{x}]\backslash[\mathbf{y}]) \Rightarrow c(\mathbf{x})$*). Define* $\mathcal{T}' := \mathrm{InferLSU}_{\mathrm{outer}}(\langle\mathbf{x}\rangle,[\mathbf{y}],c)$ *(respectively* $\mathcal{T}' := \mathrm{InferLSU}_{\mathrm{inner}}(\langle\mathbf{x}\rangle,[\mathbf{y}],c)$*). Then the SU-boxes of* $\mathcal{T}'$ *are all consistent with* $\mathcal{P}$. *Furthermore,*

$$\cup\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}'\} = \cup\{\mathrm{box}(\langle\mathbf{x}'\rangle) : \langle\mathbf{x}'\rangle \in \mathcal{T}\}. \tag{10}$$

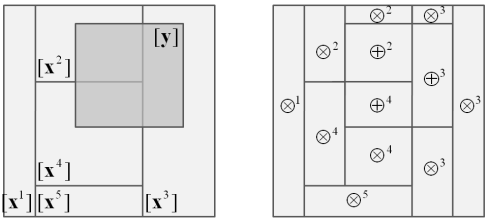

Fig. 3. Left hand side: Boxes obtained from Figure 2. The box $[\mathbf{y}]$ represents the contraction (inner or outer) of a box. Right hand side: The process illustrated by Figure 2 is repeated for each box $[\mathbf{x}^k]$, leading to twelve new SU-boxes, among which $\otimes^k$ have their sets of constraints updated.

5.5 The Branch and Bound Algorithm

Algorithm 1 uses three sets of SU-boxes $\mathcal{L}$ (SU-boxes to be proceeded), $\mathcal{D}$ (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 is to maintain the following property:

$$\text{MaxSol}(\mathcal{P}) \subseteq \bigcup \{\text{box}(\langle \mathbf{x} \rangle) : \langle \mathbf{x} \rangle \in \mathcal{L} \cup \mathcal{D} \cup \mathcal{E}\}, \tag{11}$$

while using inner and outer contractions and the function $\text{InferLSU}_{\text{type}}$ 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, (11) still holds and hence:

$$\text{MaxSol}(\mathcal{P}) \subseteq \cup\{\text{box}(\langle \mathbf{x} \rangle) : \langle \mathbf{x} \rangle \in \mathcal{D} \cup \mathcal{E}\}. \tag{12}$$

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 NCSP), and an interval $[\underline{m}, \overline{m}]$ for $m_{\mathcal{P}}$. We already have a lower bound $\underline{m}$ on $m_{\mathcal{P}}$. Now, as (11) holds, an upper bound can be computed in the following way:

$$\overline{m} = \max\{(\#\mathcal{S}) + (\#\mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E}\}. \tag{13}$$

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-box of $\mathcal{D}$ are proved to contain only max-solutions of $\mathcal{P}$:

$$\forall \langle \mathbf{x} \rangle \in \mathcal{D} \ , \ \text{box}(\langle \mathbf{x} \rangle) \subseteq \text{MaxSol}(\mathcal{P}). \tag{14}$$

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

$$\mathcal{B} = \{\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m}\}. \tag{15}$$

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

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:

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

As a consequence of Proposition 3, constraints are correctly removed $\mathcal{U}$ 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 lack of space).

Remark 1. The worth 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 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 7.2) have shown that t_c should be set between 5 and 10 in order to speed up the solving process.

6 Related Work

A large literature is related to Max-CSP frameworks and solving techniques. Most formulate the problem as a combinatorial optimisation 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 [13] and Random-walk [14] 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 optimisation problems (*cf. e.g.*, Continuous-GRASP[15]), 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 in continuous domains are less numerous than 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.* [16] 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 an evaluation-bisection.

7 Experiments

7.1 Camera Control Problem

This subsection is dedicated to the virtual camera placement example presented in section 2. Let us recall briefly that this motivating example involves 2 variables and 6 constraints expressed as dot products and Euclidian distances in 2D. The corresponding paving is shown in Figure 4 while results are presented in Table 1 within the search space $\{[-30; 30], [1; 30]\}$ with $\epsilon = 0.01$.

Although Christie *et al.* [18] 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.

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

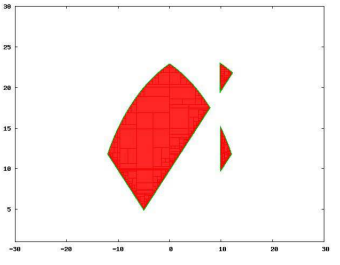

Fig. 4. Pavings generated by our algorithm on the VCP problem. Red boxes represent the max solution set (*cf.* Figure 1), while green boxes illustrate boundary boxes encompassing the solution set.

Table 1. Approximated timings thanks to [17] for the VCP example between our approach (PentiumM750 laptop, 1Go RAM) and Christie *et al.* [18](Pentium T7600 2.33Ghz 2Go RAM)

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

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. Indeed Figure 4 shows that the search space is divided into three connected components each encompassing a different set of equivalent solutions. An automatic connected component identification algorithm (*cf.* [19]) 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 constraint he wants to be relaxed into the original problem.

7.2 The Facility Location Problem

This example is based on a well-known problem in Operational Research: the facility location problem [20]. 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 problem consists in finding the smallest number of facilities such that each customer can be served by at least one facility.

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 maximises 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 $\mathcal{S}_m$ 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.

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

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 2. 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.

7.3 Parameter Estimation with Bounded Error Measurements

This experiment is based on a parameter estimation problem presented in [16]. The problem consists in determining unknown parameters $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2}) \in$

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

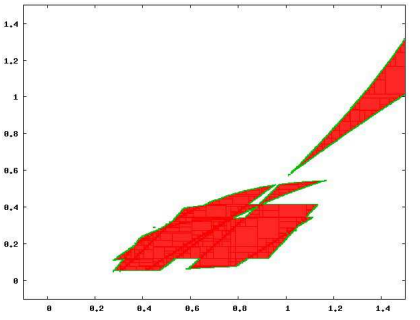

Fig. 5. Paving generated for 3 possible outliers. Red boxes represent inner approximation (*i.e.*, satisfying at least 7 of the 10 constraints) while green boxes illustrate boundary boxes encompassing the solution set.

Table 2. Timings for the facility location problem for 500 customers randomly generated in $\{[-5;5],[-5;5],[-5;5]\}$ on a PentiumM750 laptop, 1Go RAM

Method	Time (seconds)	Number of facilities
Greedy Num. Max-CSP	69.25	39
Basic OR technique	$\approx$ 140	45

Table 3. Original and approximate timings obtained from a Pentium M750 laptop processor (Max-NCSP) and from a 486 DX4-100 processor (Jaulin *et al.*) thanks to [17]

Method	ϵ bisection	T_{original} (seconds)	T_{approx} (seconds)
Max-NCSP	0.005	0.46	0.46
Jaulin *et al.*	0.005	$\approx$ 180	$\approx$ 1.9

($[-\mathbf{0.1},\mathbf{1.5}],[-\mathbf{0.1},\mathbf{1.5}]$) of a physical model thanks to a set of experimental data obtained from a system.

Here the model used is a two-parameter estimation problem taken from Jaulin and Walter [21] 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. This means that an outlier is compatible with the other error measurements. Hence, the Max-CSP cannot directly help solving this problem. From 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 1 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 [16], we set a maximum of 3 outliers, Figure 5 shows the corresponding paving. Although Jaulin *et al.* do not compute an inner and outer approximation of the solution set, Table 3 compares our results. Approximate timings have been computed using [17] 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 [16] are clearly less accurate than the pavings computed by our approach). Moreover, we want to emphasize on 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. Our method can address any kind of CSP: well-constrained, under-constrained and over-constrained. 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.

Our algorithm could be extended to manage hierarchical constraints in a predicate *locally-better* way [23]. 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.

Acknowledgments

The authors are grateful to Frédéric Saubion and to the anonymous referees whose comments significantly improved the paper.

References

1. Petit, T., Régin, J.C., Bessière, C.: Range-based algorithm for max-csp. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 280–294. Springer, Heidelberg (2002)
2. de Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving max-sat as weighted csp. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 363–376. Springer, Heidelberg (2003)

3. Neumaier, A.: Interval Methods for Systems of Equations (1990)
4. Hayes, B.: A Lucid Interval. American Scientist 91(6), 484–488 (2003)
5. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs (1966)
6. Lhomme, O.: Consistency techniques for numeric csps. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI), pp. 232–238 (1993)
7. Benhamou, F., Older, W.J.: Applying interval arithmetic to real, integer and boolean constraints. Journal of Logic Programming 32(1), 1–24 (1997)
8. Benhamou, F., McAllester, D., van Hentenryck, P.: Clp(intervals) revisited. In: ILPS 1994, pp. 124–138. MIT Press, Cambridge (1994)
9. Mackworth, A.K.: Consistency in networks of relations. AI 8(1), 99–118 (1977)
10. Collavizza, H., Delobel, F., Rueher, M.: Extending Consistent Domains of Numeric CSP. In: Proceedings of IJCAI 1999 (1999)
11. Benhamou, F., Goualard, F., Languenou, E., Christie, M.: Interval Constraint Solving for Camera Control and Motion Planning. ACM Trans. Comput. Logic 5(4), 732–767 (2004)
12. Normand, J.M.: Placement de caméra en environnements virtuels. PhD thesis, Université de Nantes (2008)
13. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. AI 58(1-3), 161–205 (1992)
14. Wallace, R.J.: Analysis of heuristic methods for partial constraint satisfaction problems. In: Principles and Practice of Constraint Programming, pp. 482–496 (1996)
15. Hirsch, M.J., Meneses, C.N., Pardalos, P.M., Resende, M.G.C.: Global optimization by continuous grasp. Optimization Letters 1, 201–212 (2007)
16. Jaulin, L., Walter, E.: Guaranteed tuning, with application to robust control and motion planning. Automatica 32(8), 1217–1221 (1996)
17. Dongarra, J.: Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee (2007)
18. Christie, M., Normand, J.M., Truchet, C.: Computing inner approximations of numerical maxcsp. In: IntCP 2006 (2006)
19. Delanoue, N., Jaulin, L., Cottenceau, B.: Counting the number of connected components of a set and its application to robotics. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 93–101. Springer, Heidelberg (2006)
20. Drezner, Z., Hamacher, H. (eds.): Facility Location. Applications and Theory. Springer, New-York (2002)
21. Jaulin, L., Walter, E.: Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. Math. Comput. Simul. 35(2), 123–137 (1993)
22. Milanese, M., Vicino, A.: Estimation theory for nonlinear models and set membership uncertainty. Automatica 27(2), 403–408 (1991)
23. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. Lisp Symb. Comput. 5(3), 223–270 (1992)