Clustering for Disconnected Solution Sets of Numerical CSPs

Xuan-Ha Vu, Djamila Sam-Haroud, and Boi Faltings

Artificial Intelligence Laboratory,
Swiss Federal Institute of Technology in Lausanne (EPFL),
CH-1015, Lausanne, Switzerland
{xuan-ha.vu, jamila.sam, boi.faltings}@epfl.ch
http://liawww.epfl.ch

Abstract. This paper considers the issue of postprocessing the output of interval-based solvers for further exploitations when solving numerical CSPs with continuum of solutions. Most interval-based solvers cover the solution sets of such problems with a large collection of boxes. This makes it difficult to exploit their results for other purposes than simple querying. For many practical problems, it is highly desirable to run more complex queries on the representations of the solution set. We propose to use clustering techniques to regroup the output in order to provide some characteristics of the solution set. Three new clustering algorithms based on connectedness and their combinations are proposed.

1 Introduction

Many practical applications involve the solving of constraint satisfaction problems (CSPs). A numerical CSP (NCSP) is stated as a set of variables taking their values in domains over the reals and subject to a finite set of constraints. In practice, constraints can be equalities or inequalities of arbitrary type, usually expressed using arithmetic and logical expressions. NCSPs with *non-isolated solutions* are often encountered in real-world engineering applications. In such applications, a set of non-isolated solutions often expresses relevant alternatives that need to be identified as precisely and completely as possible. Interval constraint-based solvers take as input an NCSP and generate a collection of boxes which *conservatively* encloses the solution set. They have shown their ability to solve some complex instances of NCSPs with non-isolated solutions, especially in low-dimensional space. However, they provide enclosures that are still prohibitively verbose for further exploitations rather than just a simple query[1]. More complex queries, on connectedness, satisfaction of universal quantification or intersection for example, are however central to many applications. Although the running time of interval-based solvers is getting gradually improved, the space complexity (the number of boxes) is at least proportional to the number of boxes needed to cover the boundary of the solution set, therefore still very verbose.

[1] In [1], worst-case query time of a *box-tree* in d-dimensions is $\Theta(N^{1-1/d} + k)$, where N is the number of boxes and k is the number of boxes intersecting the *query range*.

One of the applications of interval-based solvers we are investigating is a cooperation scheme of optimization and constraint satisfaction, where the verbose output data of the solvers is usually simplified by imposing a limit on number of boxes to be produced or by restricting the solving process to a low predefined precision. However, in case the solution set is complex and disconnected, the above simplifications affect significantly the quality of the output and make it unsuitable for practical use. We propose to use fast clustering techniques to regroup a large collection of output boxes into a reduced collection of boxes each of which tightly covers one or several connected subsets. The needs for progress on this direction are also ubiquitous in real-world applications such as collision detection, self-collision detection in molecular dynamics and dynamic systems, where the systems have multiple components in low-dimensional space and the reduced collection of boxes is useful for soon guaranteeing that systems have no collisions, hence further expensive computations can be avoided in many cases. We just cite as example some recent works in collision detection ([2], [3], [4], [5], [6], [7], [8]) which have showed that using *bounding-volume techniques* could gain good results. These techniques aim at computing bounding boxes which are very similar to the output from interval-based solvers for NCSPs. In such applications, interval-based solvers can be used to produce *conservative* approximations of the solution sets within acceptable time and clustering techniques can be used to bridge the gap between the verboseness in the output of interval-based solvers and the compactness required by the applications.

It is known that general computation of an optimal set of k clusters is NP-complete [9]. For this reason, fast clusterings usually can be achieved only by using heuristic algorithms or by restricting to approximations. Very recently, a fast clustering algorithm named `ANTCLUST` [10] was proposed. This algorithm is close to, but still not suitable for addressing our requirements. Though successful, the general clustering techniques are not suitable for directly applying to our case because they are not guaranteed to be convergent and the homogeneity information in our problems is not available as required. Therefore, specific clustering techniques for data produced by interval-based solvers are needed.

In Section 3, we first propose a basic clustering algorithm, called `COLONIZATION`, that takes $O(dN^2)$ time to cluster N boxes in d-dimensions. The basic method computes homogeneity (i.e. the connectedness in this case) for each pair of boxes. Moreover, most search techniques implemented in interval-based solvers follow the *branch-and-prune* scheme, hence they essentially produce boxes in the tree structure.[2] Taking advantages of the tree structure, we propose two approximate algorithms to cluster a large collection of boxes organized in form of tree. The first algorithm, called `MCC`, very quickly performs clusterings such that no connected subsets are partitioned. The second algorithm, called `SDC`, generates more adaptive clusterings in very short time. It can be seen as a further process of `MCC`. In Section 3.5, we will also discuss about combining the proposed algorithms to reduce the running time of the basic algorithm.

[2] Even if the tree structure is not available, we still can construct a good bounding-box tree from a list of boxes in $O(N \log N)$ time [1].

2 Background and Definitions

2.1 Interval Arithmetic

The finite nature of computers precludes an exact representation of the *reals*. The real set, $\mathbb{R}$, is in practice approximated by a finite set $\mathbb{F}_\infty = \mathbb{F} \cup \{-\infty, +\infty\}$, where $\mathbb{F}$ is a finite set of reals corresponding to the floating-point numbers. In this paper, we restrict the notion of *interval* to refer to real intervals with bounds in $\mathbb{F}_\infty$ no matter they are open or closed (see [11] for more details). The lower and upper brackets of intervals are taken in $\mathcal{L} = \{\text{“(”}, \text{“[”}\}$ and $\mathcal{U} = \{\text{“)”}, \text{“]”}\}$ respectively. A total ordering $\prec$ is defined on the set of brackets as follows: “)” $\prec$ “[” $\prec$ “]” $\prec$ “(”. The set of floating-point bounds is $\mathbb{F}_\diamond = \mathbb{F}_\triangleleft \cup \mathbb{F}_\triangleright$, where $\mathbb{F}_\triangleleft = \mathbb{F} \times \mathcal{L} \cup \{(-\infty, \text{“(”})\}$ and $\mathbb{F}_\triangleright = \mathbb{F} \times \mathcal{U} \cup \{(+\infty, \text{“)”})\}$. $\mathbb{F}_\diamond$ is totally ordered by the ordering $<$ defined as follows: $\forall \beta_1 = (x_1, \alpha_1), \beta_2 = (x_2, \alpha_2) \in \mathbb{F}_\diamond : \beta_1 < \beta_2 \Leftrightarrow x_1 < x_2 \vee (x_1 = x_2 \wedge \alpha_1 \prec \alpha_2)$. The bounds in $\mathbb{F}_\diamond$ are used to construct the set of intervals $\mathbb{I} = \{\langle \beta_1, \beta_2 \rangle \mid \beta_1 < \beta_2, \beta_1 \in \mathbb{F}_\triangleleft, \beta_2 \in \mathbb{F}_\triangleright\}$. We can also use the usual notations of real interval by following the next conventions (where $x, y \in \mathbb{R}$):

$$
\begin{aligned}
[x, y] &= \langle (x, \text{“[”}), (y, \text{“]”}) \rangle &&= \{r \in \mathbb{R} \mid x \leq r \leq y\} \\
(x, y] &= \langle (x, \text{“(”}), (y, \text{“]”}) \rangle &&= \{r \in \mathbb{R} \mid x < r \leq y\} \\
[x, y) &= \langle (x, \text{“[”}), (y, \text{“)”}) \rangle &&= \{r \in \mathbb{R} \mid x \leq r < y\} \\
(x, y] &= \langle (x, \text{“(”}), (y, \text{“)”}) \rangle &&= \{r \in \mathbb{R} \mid x < r < y\}
\end{aligned}
$$

For convenience, for each $\beta = (x, \alpha) \in \mathbb{F}_\diamond$ we define $\neg\beta = (x, \neg\alpha)$, where $\neg\text{“)”} = \text{“[”}, \neg\text{“[”} = \text{“)”}, \neg\text{“]”} = \text{“(”}, \neg\text{“(”} = \text{“]”}$.

Interval arithmetic is an arithmetic defined on sets of intervals, rather than sets of real numbers. According to a survey paper [12], a form of interval arithmetic perhaps first appeared in 1924 in [13], then later in [14]. Modern development of interval arithmetic began with Moore's dissertation [15]. Interval arithmetic has been being used to solve numerical problems with guaranteed accuracy. Readers are referred to [16], [17] and [18] for more details on basic interval methods. Most of the interval methods can be easily extended to accept the notation of interval in this paper.

2.2 Relations and Approximations

A relation can be approximated by a computer-representable superset or subset. The former is a *complete* approximation but may contain points that are not solutions. Conversely, the latter is a *sound* approximation but may lose certain solutions. In practice, a relation can be approximated coarsely by the smallest box, called the *(interval) hull* and denoted by *hull*(.), containing it.

A collection of objects representing points is called *disjoint* if every two objects have no common points. Two objects are called *hull-disjoint* if the two hulls of them are disjoint. A set is called a *disconnected set* if it can be partitioned into two nonempty subsets such that each subset has no points in common with the *set closure*[3] of the other, otherwise it is called a *connected set*.

[3] The set closure of a set S is the smallest closed set containing S.

Approximating a relation usually requires a decomposition into simple components such as convex subsets. Checking for disjoint is trivial with boxes but not trivial with general convex sets. Therefore, in this paper we only investigate the decompositions that partition the relation into disjoint boxes.

2.3 Numerical Constraint Satisfaction Problems

A *numerical constraint satisfaction problem* (NCSP) is a CSP, $P = (\mathcal{V}, \mathcal{C}, \mathcal{D})$, where $\mathcal{V}$ consists of d variables taking values in domains $\mathcal{D} = D_1 \times \ldots \times D_d \subseteq \mathbb{R}^d$. In practice, each domain is usually given as one (or several) intervals. A solution to P is a point in $\mathcal{D}$ which satisfies all the constraints in $\mathcal{C}$. In general, the computation of the solution set of an NCSP is intractable. Instead of that, one may consider the computation of inner and outer approximations in form of unions of disjoint boxes, they are therefore called the *inner* and *outer union approximations* [19, 20], respectively. Various bisection techniques were described in [18] to compute outer union approximations. Recently, [11] has computed inner union approximations for universally quantified constraints by using the *negation test*, [21] has used the negation test in combination with enhanced splitting strategies to compute outer union approximations for NCSPs. Recently, the work in [19, 20] has proposed an improvement on the works in [11, 21] to compute inner and/or outer union approximations.

In general, the computation of the union approximations relies on combining the *local consistency* and *global search* techniques to construct a collection of disjoint boxes covering the solution set or contained in it. In numerical domain, the local consistency techniques usually employed are *Box*, *Hull*, *kB*, *Bound* consistency and variants ([22], [23], [24], [25], [26]). Interval-based search techniques usually use the *branch-and-prune* scheme that combines local consistency techniques with orthogonal-splitting strategies, where the *orthogonal-splitting* refers to the split performed in succession by hyper-planes that are orthogonal to axes. In solving NCSPs with non-isolated solutions, local consistency techniques are employed to implement *contracting operators* ([18], [11], [21], [19, 20]) in order to narrow the domains during search. In the group of initial search techniques, output can be structured in form of uniform trees like 2^k-tree ([27], [28]) with $k = 3$, or a binary/ternary tree [18], while in another group of recent techniques the output can be structured in form of more general trees ([11], [21], [19], [20]). The number of children of a search node in the latter group does not exceed $2d + 1$ while the number of search nodes in the latter is usually much less than that in the former.

3 Clustering for Disconnected Solution Set

Concepts. When solving NCSPs with non-isolated solutions, interval-based search techniques following the branch-and-prune scheme produce a large collection of boxes that can be maintained in a *bounding-box tree*, where child nodes represent for branchings in search. The bounding boxes stored at search

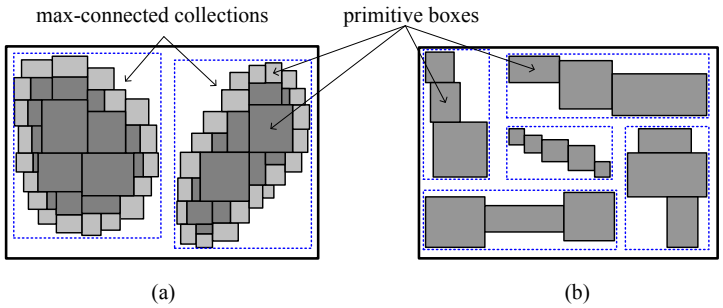

Fig. 1. (a) This tree is orthogonal-separable: grey boxes are primitive boxes, they form two max-connected collections; (b) This tree is not orthogonal-separable

nodes are results of the pruning phase. In literature, there are some variants of bounding-box tree like *bounding-volume tree*, *interval tree*, *box-tree* and *AABB tree*. We call the boxes at the tree leaves that is produced by the solvers the *primitive boxes*. In this paper, we use the notion of *hull of boxes* to refer to the smallest box that contains all the given boxes and also, for convenience, to refer to the collection of the involved boxes if not confusing. A collection of primitive boxes is called *connected* if the union of the boxes is a connected set. If this is the case, we say that the primitive boxes connect to each other. A collection of primitive boxes is called *max-connected* (w.r.t to the set of all primitive boxes) if it is connected and there is no other primitive boxes that connect to the given boxes. A clustering is called *max-connected* if each connected collection in a cluster is max-connected. A bounding-box tree is called *orthogonal-separable* if every decomposition of each bounding-box into pairwise disconnected collections (of primitive boxes) can be performed by sequentially using separating hyper-planes that are orthogonal to axes (see Figure 1).

Goals Of Clustering. The solution set of an NCSP with non-isolated solutions usually consists of one or more connected subsets each of which is a continuum. In many applications, there is a need for enclosing the solution set by a collection of disjoint bounding boxes such that each of the connected subsets is contained in only one bounding box. To describe the connected subsets as well as possible, the number of bounding boxes should be as big as possible. Interval-based search techniques usually produce approximations that are either very poor (if being stopped at low precision), or prohibitively verbose (if being stopped at medium/high precision) when solving problems with non-isolated and disconnected solution sets. When a certain quality is required, we obviously need to take the latter setting and then do a clustering on the large collection of boxes. However, the exact solution set is unknown yet, only a collection of primitive boxes is known. The above need is then translated into the need for a max-connected clustering. If a max-connected clustering has the maximum

number of clusters, we call it an *optimal max-connected clustering*[4]. The optimal max-connected clustering of an orthogonal-separable bounding-box tree provides pairwise hull-disjoint clusters each of which is a hull of exact one max-connected collection of primitive boxes (see Figure 13-a). In case the bounding-box tree is not orthogonal-separable, the clusters in the optimal max-connected clustering may not be hull-disjoint (see Figure 13-b). In this case, we may need a further decomposition in order to obtain hull-disjoint, if this property is required by applications (see Figure 13-b).

In the next subsections, we propose three new algorithms and their combinations to address different goals. The first subsection focuses on a basic algorithm that computes the optimal max-connected clustering. It is however not very efficient in practice, we then incrementally propose two alternative algorithms. Each algorithm has two phases, but they have the same first phase. The second subsection describes this common phase. The third subsection describes an algorithm to compute max-connected clusters which are hull-disjoint. The fourth subsection gives an algorithm to compute more adaptive clusterings. In the last subsection, we discuss about combining the proposed algorithms in order to reduce the running time of computing the optimal max-connected clustering.

3.1 Optimal Max-Connected Clustering

As far as we know, there does not exist any algorithm that are suitable to find the optimal max-connected clustering. Fortunately, there exists a recent clustering algorithm named `ANTCLUST` [10] that exploits the phenomenon known as *colonial closure* of ants to create homogeneous groups of individuals. The `ANTCLUST` algorithm addresses the general clustering problem in the way that we can borrow a part for addressing our goals. Inspired by Seidel's *invasion* algorithm [29] and some ideas in the `ANTCLUST` algorithm, we propose a simple deterministic algorithm, called `COLONIZATION`, which is given in Figure 2 to compute the optimal max-connected clustering. This basic algorithm check if each unprocessed box connects to existing collections. The check is taken on each member box of each collection to see if the unprocessed box connect to the collection. The other processes are described in detailed in Figure 2. It is easy to see that in the worst-case the first phase (lines 01-12) takes $d(1+2+\ldots+(N-1)) = dN(N-1)/2$ checks for connectedness of two intervals, where N is the number of primitive boxes in d-dimensions. As a result, the time complexity of the first phase is $O(dN^2)$. At the end of the first phase, all produced collections are max-connected and they have no common points, but their hulls are not guaranteed to be disjoint. The number of collections produced by the first phase is therefore equal to the maximum number, p, of max-connected collections. The second phase (lines 14-16) is optional. It is only for the applications that requires hull-disjoint. The second phase has the time complexity $O(dp^2)$. We obviously have $p \leq N$, hence the total time complexity of the `COLONIZATION` algorithm is

[4] It is easy to prove that the optimal max-connected clustering exists uniquely.

00: **algorithm** COLONIZATION
01: *First phase to obtain optimal max-connectedness:*
02: $\mathcal{L} := \emptyset$;
03: **for** each box, $\mathbf{B}$, not in any collection in $\mathcal{L}$ **do**
04: $C := \{\mathbf{B}\}$;
05: **for** each $C_i \in \mathcal{L}$ **do**
06: **if** $\{\mathbf{B}\} \cup C_i$ is connected **then**
07: $C := C \cup C_i$;
08: $\mathcal{L} := \mathcal{L} \setminus \{C_i\}$;
09: **end-if**
10: **end-for**
11: $\mathcal{L} := \mathcal{L} \cup \{C\}$;
12: **end-for**
13: *Optional phase to obtain hull-disjoint:*
14: Replace each collection by the hull of its primitive boxes.
15: **while** there exist two hulls that have nonempty intersection **do**
16: Combine the two hulls into a single hull.
17: **end-while**
18: **end**

Fig. 2. The COLONIZATION algorithm

$O(dN^2)$, or $O(N^2)$ if d is fixed. In practice, $p \ll N$ and p is bounded for fixed problems.

3.2 Separator Computation

In order to cluster the primitive boxes into a number of disjoint hulls, we can use hyper-planes that are orthogonal to axes to separate the primitive boxes. We then define new notations for computing such separating hyper-planes as follows.

Definition 1 (Separator, SPT). *Given a hull,* $\mathbf{H}$*, of primitive boxes, an axis* $x \in \mathbb{N}$*, and an interval,* $I \in \mathbb{I}$*. The couple* (x, I) *is called a* separator *of* $\mathbf{H}$ *if* I *is a maximal (w.r.t. the set inclusion) interval that satisfies the following condition for each primitive box,* $\mathbf{B}$*, in* $\mathbf{H}$*:* $I \subseteq \mathbf{H}|_x \wedge I \cap \mathbf{B}|_x = \emptyset$*. When this holds,* I *is called a* separating interval *and* x *is called a* separating axis*. The set of all separators of* $\mathbf{H}$ *on axis* x *is denoted by* $\mathtt{SPT}(\mathbf{H}, x)$*.* $\mathtt{SPT}(\mathbf{H}) = \cup_x \mathtt{SPT}(\mathbf{H}, x)$*.*

Definition 2 (Extension, EXT). *Given a hull,* $\mathbf{H}$*, of primitive boxes, an axis* $x \in \mathbb{N}$*, and a box* $\mathbf{B} \subseteq \mathbf{H}$*. A couple* (x, I) *is called an* extension *of* $\mathbf{B}$ *w.r.t.* $\mathbf{H}$ *(on axis* x*) if* I *is an interval that is maximal (w.r.t. the set inclusion) in* $\mathbf{H}|_x \setminus \mathbf{B}|_x$*. When this holds,* I *is called an* extending interval *and* x *is called an* extending axis*. We denote by* $\mathtt{EXT}(\mathbf{B}, x)$ *the set of extensions of* $\mathbf{B}$ *(w.r.t.* $\mathbf{H}$*) on axis* x*.*

Figure 3 gives an illustration of the two above notions. It is easy to see that, on each axis, a hull has at most two extensions w.r.t. its parent hull and that all separators of the hull lie between the two extensions. In a bounding-box tree of

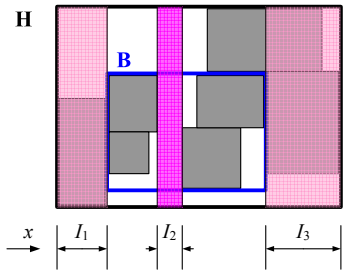

Fig. 3. Grey boxes are primitive; (x, I_2) is a separator of **B** and **H**; (x, I_1) and (x, I_3) are extensions of **B** to **H** on axis x

primitive boxes, we do a process in a bottom-up manner (e.g. in post-order) to make each bounding box to be the hull of its child bounding boxes. Hence, each node's box becomes the hull of primitive boxes contained in it. A bounding-box tree whose bounding boxes are the hulls of primitive boxes contained in the bounding boxes is called a *fitted tree*.

Definition 3 (Separating Set, SE). *In a fitted tree, given an axis $x \in \mathbb{N}$ and a hull,* **H**, *at a node. The union of the intervals of all separators and extensions*[5] *of* **H** *on axis x is called the* separating set *of* **H** *on axis x and denoted by* $\texttt{SE}(\mathbf{H}, x)$.

For simplicity, we use the same notions SPT, EXT, SE for the tree node corresponding to the hull **H**. The computation of extensions of a hull w.r.t. its parent hull is trivial and takes $O(d)$ time. We can see that, in a fitted tree, a couple (x, I) is a separator of a hull if and only if I is a maximal (w.r.t. the set inclusion) interval that can be contained in all the separating sets on axis x of all children of the hull. It means that all the separators of the hull can be computed from separators and extensions of its children by intersection. Moreover, this is true even during the above-mentioned bottom-up process. The ordering relation among separators and extensions is given by the following proposition.

Proposition 1. *In a fitted tree, given a hull,* **H**, *of primitive boxes. We have that any two different separating/extending intervals of* **H** *on the same axis do not connect to each other. Moreover, for any two separators/extentions, s_1 and s_2, of* **H**, *either $s_1 = s_2$, $s_1 < s_2$ or $s_2 < s_1$ holds. Where the ordering relation between separators/extentions is defined as follows.*

$$\textit{(i)}\ \ (x_1, I_1) = (x_2, I_2) \Leftrightarrow x_1 = x_2 \wedge I_1 = I_2$$
$$\textit{(ii)}\ (x_1, I_1) < (x_2, I_2) \Leftrightarrow x_1 < x_2 \vee (x_1 = x_2 \wedge I_1 < I_2)$$

Proof. By Definition 1 and Definition 2, one can easily see that every two different separators/extensions of **H** on the same axis do not connect to each other, otherwise either the separators/extensions are not maximal w.r.t to the set inclusion or **H** is not the hull of its children. As a result, the set of all separating

[5] If not specified, the extensions of a hull are taken w.r.t. the parent hull in the tree.

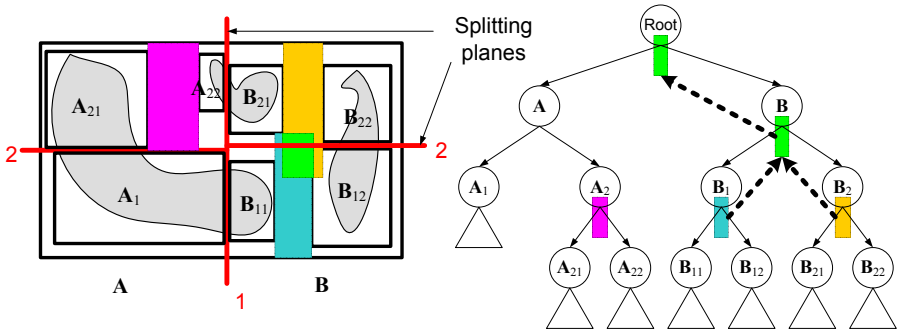

Fig. 4. In the fitting process on a bounding-box tree: the computation of separators is processed in bottom-up manner

and extending intervals on an axis is totally ordered. This results in what we have to prove.

To compute separators of all nodes in a bounding-box tree, we only need to do a single bottom-up process, called the *fitting process*. In summary, in the fitting process, operations at each node consist of (i) making the box at the current node to be the hull of its children; and (ii) computing the ordered set of separators by taking the ordered intersection of ordered separating sets of its children. The details of the fitting process is given in Figure 6. We denote by m the maximum number of children of a tree node. The operation (i) is trivial and takes $O(md)$ time. The leaf nodes of the tree have no separators. Each node in the tree has at most two extensions that straddle the box stored at the node, hence straddle the set of separators. Moreover, the bottom-up process starts at leaf nodes, therefore the separating sets can be computed by the intersection in the operation (ii) and maintained sorted by the total order in Proposition 1.

At the current node, we denote by $q_{i,j}$ the number of elements in the separating set on axis $i \in \mathbb{N}$ of the j-th child hull, where $1 \leq i \leq d, 1 \leq j \leq m$. We denote $\bar{q}_j = \sum_i q_{i,j}$ and $\bar{q} = \max_j\{\bar{q}_j\}$, then $\bar{q}$ is the maximum number of separators/extensions that a child of the current node can have. Computing the intersection of a sorted collection of k_1 pairwise disconnected intervals and another sorted collection of k_2 pairwise disconnected intervals takes $O(k_1+k_2)$ time (see Figure 5). By Proposition 1 and the result of *pullSeparators* in Figure 6, $q_{i,j}$ separating/extending intervals in the separating set on axis i of the j-th child are pairwise disconnected and totally ordered. Therefore, the time complexity of computing the intersection of $\sum_j q_{i,j}$ separating/extending intervals on axis i of all children is linear in the number of intervals, i.e. $O(\sum_j q_{i,j})$. As a result, the time complexity of the operation (ii) is $O(\sum_i \sum_j q_{i,j}) = O(\sum_j \sum_i q_{i,j}) = O(\sum_j \bar{q}_j)$, i.e. not greater than $O(m\bar{q})$. Because the number of nodes in the tree is $O(N)$, the total time complexity of the fitting process is $O(mdN + Q)$, where Q is the total number of separators/extensions in the tree except in the root. This can not exceed $O((md + q)N)$, where q is the maximum number of sepa-

```
function intersect({⟨α_i^1, β_i^1⟩ ∈ 𝕀 | 1 ≤ i ≤ k_1}, {⟨α_j^2, β_j^2⟩ ∈ 𝕀 | 1 ≤ j ≤ k_2})
    𝒯 := ∅; i := 1; j := 1;
    while i ≤ k_1 ∧ j ≤ k_2 do
        if ¬β_i^1 ≤ α_j^2 then {i := i + 1; continue while;}
        if ¬β_j^2 ≤ α_i^1 then {j := j + 1; continue while;}
         α := max{α_i^1, α_j^2}; β := min{β_i^1, β_j^2};
         𝒯 := 𝒯 + {⟨α, β⟩};
         if β = β_i^1 then i := i + 1;
         if β = β_j^2 then j := j + 1;
    end-while
    return 𝒯;
end
```

Fig. 5. This function computes the intersection of two ordered collections of pairwise disconnected intervals. The intersection of multiple collections is computed by calling this function

```
function FittingProcess(in/out : 𝒯_0)        /* 𝒯_0 is a bounding-box tree */
    for each node P of 𝒯_0 in a post-order visit do
        if P is leaf then
            for each axis x do SPT(P, x) := ∅;
        else
            pullSeparators(P);
        end-if
    end-for
end

function pullSeparators(in/out : P)        /* P is a tree node */
    if P is leaf then return;        /* Needed for the calls in Figure 9 */
    𝒞 := children(P);
    Set the box at P to hull({B ∈ 𝕀^d | B is a box at a node in 𝒞}).
    for each C ∈ 𝒞 and each axis x do
        Compute EXT(C, x) and denote it by {E_l, E_u}.        /* may be empty */
        SE(C, x) := {E_l, SPT(C, x), E_u};        /* ordered set */
    end-for
    for each axis x do
        SPT(P, x) := intersect{SE(C, x) | C ∈ 𝒞}; /* Ignore x to get the intervals */
    SPT(P) := {SPT(P, 1), ..., SPT(P, d)};        /* ordered set */
end
```

Fig. 6. The functions for the fitting process

rators/extensions that each node in the tree has. In most existing solvers, m is not greater than $2d+1$, and it is usually small in comparison with $2d+1$ (e.g. $m=2$ if bisection is used). We conjecture that q is bounded for fixed problems, particularly it is bounded by $O(p)$, where p is given in Section 3.1. If this conjecture is true, as our experiments show, the time complexity of the fitting process is $O(N)$ for fixed problems.

3.3 Max-Connected Clustering

The second phase is called the *separating process*. We now compute a max-connected clustering based on next propositions. During the separating process, we maintain max-connected collections of primitive boxes in form of fitted trees.

Proposition 2. *Any separator stored at the root of a fitted tree representing a max-connected collection can be used to partition this tree into two fitted subtrees each of which represents a max-connected collection.*

Proposition 3. *If a set of primitive boxes represented by a fitted tree can be partitioned by a hyper-plane that is orthogonal to axis x into two collections whose projections on x do not connect to each other, then the root of the fitted tree has some separating interval that contains the projection of the hyper-plane on x.*

Proof of these propositions is trivial due to Definition 1 and the definition of fitted tree. By these propositions, we can compute a max-connected clustering from the separators computed in Section 3.2. The process can be done in bottom-up manner, or simply by a recursive call at root. For simplicity, we describe the process in recursive mode. A recursive call starts from a root node of a fitted tree created during the separating process. Recursively, for each node, $\mathbf{P}$, and the current separator, $S \in \mathtt{SPT}(\mathbf{P})$, we construct two fitted trees from the subtree rooted at $\mathbf{P}$. One tree is called *lower-bound tree* (denoted by $l_{\mathbf{P}}$), the other is *upper-bound tree* (denoted by $u_{\mathbf{P}}$) with single roots being copied from $\mathbf{P}$ at first. If a child node's box lies on lower-bound (respectively upper-bound) side of S, the subtree rooted at the child node is moved to $l_{\mathbf{P}}$ ($u_{\mathbf{P}}$, respectively). Otherwise the child node, called $\mathbf{M}$, is processed similarly to construct its lower-bound and upper-bound trees, $l_{\mathbf{M}}$ and $u_{\mathbf{M}}$ respectively. The trees $l_{\mathbf{M}}$ and $u_{\mathbf{M}}$ are then attached to $l_{\mathbf{P}}$ and $u_{\mathbf{P}}$ respectively. If any tree root in this process has only one child, this child is shifted to the root. Additionally, we make roots of lower-bound and upper-bound trees to be hulls of their child nodes. The separators of the root nodes of new lower- and upper-bound trees are updated from its children.

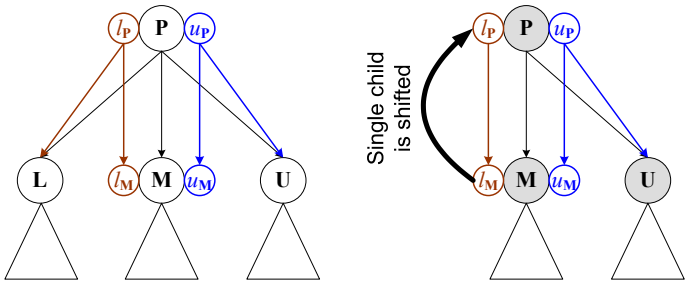

Fig. 7. The separating process: construct lower- and upper-bound trees from the subtrees on lower side and upper side respectively of a separator, and from lower- and upper-bound trees of the children that cut the separator

/* $\mathcal{T}_0$ is the initial bounding-box tree, $\mathcal{L}$ is a list of fitted tree to be return */
algorithm MCC(**in** : $\mathcal{T}_0$) → **out** : $\mathcal{L}$
    $FittingProcess(\mathcal{T}_0)$; $\mathcal{L} := \{\mathcal{T}_0\}$;
    while $\exists \mathcal{T} \in \mathcal{L}$: $\mathcal{T}$ has at least one separator, S, at root **do**
        $\mathcal{L} := (\mathcal{L} \setminus \mathcal{T}) \cup separateSubtree(root(\mathcal{T}), S)$; /* In Figure 9 */
    end-while
end

Fig. 8. The Max-Connected Clustering (MCC) algorithm

/* Input: **P** is a node of a fitted tree, separator $S \in \mathtt{SPT}(\mathbf{P})$ */
function $separateSubtree$(**in** : $\mathbf{P}, S$) → **out** : $\{l_{\mathbf{P}}, u_{\mathbf{P}}\}$
    Remove the separator S from $\mathtt{SPT}(\mathbf{P})$.
    Create two trees, $l_{\mathbf{P}}$ and $u_{\mathbf{P}}$, each has a single node copied from **P**.
    for each $\mathbf{C} \in children(\mathbf{P})$ **do**
        if **C** lies on lower side of S **then**
            Move the subtree rooted at **C** to a new subtree of the root of $l_{\mathbf{P}}$.
        else if **C** lies on upper side of S **then**
            Move the subtree rooted at **C** to a new subtree of the root of $u_{\mathbf{P}}$.
        else
            Find the separator $S' \in \mathtt{SPT}(\mathbf{C}) : S \subseteq S'$.
            $\{l_{\mathbf{C}}, u_{\mathbf{C}}\} := separateSubtree(\mathbf{C}, S')$;
            Attach $l_{\mathbf{C}}$ (resp. $u_{\mathbf{C}}$) as a new subtree of the root of $l_{\mathbf{P}}$ ($u_{\mathbf{P}}$ resp.).
        end-if
    end-for
    If any root in trees $l_{\mathbf{P}}$ or $u_{\mathbf{P}}$ has only one child, shift this child to the root.
    $pullSeparators(root(l_{\mathbf{P}}))$; $pullSeparators(root(u_{\mathbf{P}}))$;
end

Fig. 9. The separating process of a node/subtree

In Figure 8, we describe the *Max-Connected Clustering* (MCC) algorithm that uses the separating process to get a hull-disjoint max-connected clustering. For simplicity, in the MCC algorithm we use only one list, $\mathcal{L}$, however in implementation it should be maintained as two lists: one list for the trees that have no separators at root and the other list for the rest. In Figure 9, we describe the details of the separating process for a subtree rooted at a node in a fitted tree. Figure 7 and Figure 10 give some illustrations of the separating process. Figure 13-b gives another example on which MCC provides the optimal max-connected clustering. Without difficult, Based on Proposition 3 we can prove that the obtained clustering is optimal max-connected if the bounding-box tree is orthogonal-separable. As proved in Section 3.2, the time complexity of the fitting process is $O(mdN + Q)$. There are at most Q separators in the trees, each separator is processed once, each process takes $O(m)$ time. Therefore, the time complexity of the separating process is $O(mQ)$. The total time complexity of MCC is hence $O(mdN + mQ)$. This is practically shown to be linear in N for fixed problems.

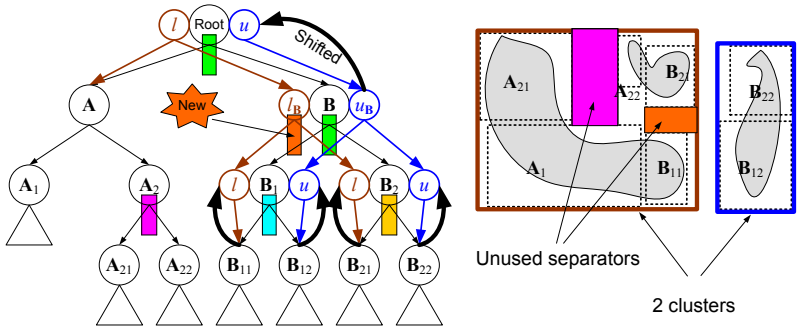

Fig. 10. MCC: The separators exist in the root of the fitted tree are used for recursive separating the tree into lower- and upper-bound subtrees. The figure on the right gives the result of this process. Two obtained clusters are hull-disjoint and max-connected

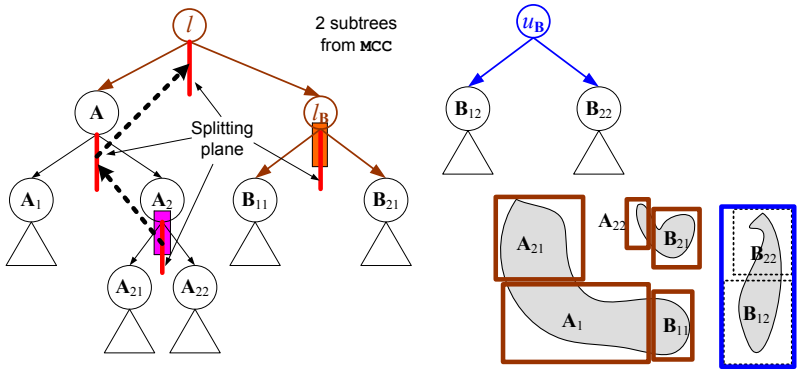

Fig. 11. SDC: The separators that still exist in subtrees after running MCC are considered as hints for decomposition. The process produces six clusters (boxes) that are pairwise hull-disjoint, but not all are max-connected

3.4 Separator-Driven Clustering

In some applications, the clustering obtained by the MCC algorithm characterizes the solution set not well enough (see the example in Figure 10). A remedy for this problem is given as an additional process for MCC so that applications can choose to whether to run it. We observe that the separators that still exist in output fitted trees of the MCC algorithm (e.g. the two separators on the right side in Figure 10) can be used as hints for further separations.

If we consider the remaining separators in the output fitted trees of MCC as hints for further decomposition, we will need to use all the splitting hyperplanes (i.e. the branchings in the trees) in the paths from the tree nodes of those separators upward to the roots of the trees for the separation. That is, all siblings of the nodes from the current node upward to the root are to be separated into

```
/* T_0 is the initial bounding-box tree, L is a list of fitted tree to be return */
algorithm SDC(in : T_0) → out : L
    L := ∅; L_0 := MCC(T_0);        /* call to the MCC algorithm */
    for each fitted tree T ∈ L_0 do
        L := L ∪ separateByAllSeparators(T);
    end-for
end

function separateByAllSeparators(in : T) → out : L
    Search in post-order, from left to right, for a node N_0 such that SPT(N_0) ≠ ∅.
    if not found then return L := {T};
    L := ∅; N := N_0; P := parent(N);        /* P may be null */
    while P ≠ ∅ do
        for each C ∈ children(P) do
            if C on the left of N then Move the subtree rooted at C to L;
            if C on the right of N then
                Detach the subtree rooted at C and make a new tree T_C.
                L := L ∪ separateByAllSeparators(T_C);
            end-if
            if C = N ≠ N_0 then Erase the node C;
            if C = N = N_0 then
                Detach the subtree rooted at C and make a new tree T_C.
                Find a separator S ∈ SPT(C);
                L := L ∪ separateSubtree(root(T_C), S);        /* In Figure 9 */
            end-if
        end-for
        N := P; P := parent(N);        /* P may be null */
    end-while
end
```

Fig. 12. The Separator-Driven Clustering (SDC) algorithm

different groups/collections in the clustering. The use of splitting hyper-planes for the separation does not guarantee that the clusters are pairwise disconnected. In Figure 12, we describe the main steps of an algorithm, called *Separator-Driven Clustering* (SDC), which performs the above idea as an additional process for the MCC algorithm. It is obvious that this algorithm produces a hull-disjoint clustering. Figure 11 gives the result of the further process for the subtrees in Figure 10. Six clusters (boxes) obtained in Figure 11 describe the solution set better than two boxes in Figure 10. In Figure 13-b, we give another example on the SDC algorithm where the solution set consisting of two connected subsets is well covered by 13 boxes each of which is a hull of primitive boxes. The SDC algorithm quickly provides an adaptive clustering as shown in our experiments. Following an argument similar to the argument in Section 3.3, we have the time complexity of SDC is $O(mdN + mQ)$. This is also practically shown to be linear in N for fixed problems.

Proposition 4. *Each cluster produced by* SDC *is a connected set.*

Proof. If there are two primitive boxes in the subtree of a cluster that form a disconnected set, then there must be a separator at their common ancestor. This contradicts the property of SDC: no separators exist in output subtrees.

3.5 Combinations of Algorithms

Three proposed algorithms produce the same results (See Figure 13-a) for orthogonal-separable bounding-box trees. However, for the bounding-box tree that is not orthogonal-separable (See Figure 10, Figure 11, Figure 13-a and Figure 14), the results of the proposed algorithms are different. Only COLONIZATION and MCC guarantee the max-connectedness of the produced clusters. COLONIZATION allows to get the best max-connected clustering but the time complexity is quadratic. MCC only allows to get a hull-disjoint max-connected clustering, it however has a very short running time in our experiments. SDC is the most adaptive clustering in very short running time, but it does not guarantee the max-connectedness. Therefore, we need to investigate the combinations of the proposed algorithms.

***Combination of* MCC *and* COLONIZATION.** As mentioned in Section 3.3, MCC provides a max-connected clustering such that the clusters are pairwise hull-disjoint, hence, if we apply COLONIZATION to every fitted tree produced by MCC, we will get the optimal max-connected clustering. The combined algorithm is therefore called the *Optimal Max-Connected Clustering* (OMCC) algorithm. Let N_i $(1 \leq i \leq p')$ is the number of primitive boxes in the i-th fitted tree produced by MCC, where p' $(\leq p)$ is the number of subtrees produced by MCC. The running time of COLONIZATION on each tree is $O(dN_i^2)$ $(1 \leq i \leq p')$. Hence, the total running time of OMCC is $time(\texttt{MCC}) + O(d\sum_i N_i^2) = O(mdN + mQ + d\sum_i N_i^2)$. Noting that $\sum_i N_i = N$, we have $\sum_i N_i^2 \leq N^2$, therefore $O(mdN + mQ + d\sum_i N_i^2)$ does not exceed $O(dN^2)$. In practice we often see that Q is much smaller than $O(dN^2)$, and $\sum_i N_i^2 \ll N^2$, hence the actual running time of OMCC is better than that of COLONIZATION for problems with highly disconnected solution set, i.e. when p' is big.

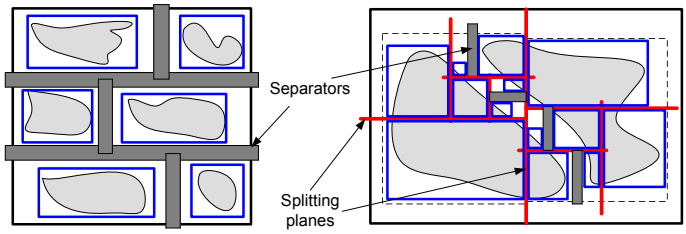

Fig. 13. The bounding-box tree is (a) orthogonal-separable: all algorithms produce the same result; (b) not orthogonal-separable: COLONIZATION produces 2 clusters (or 1 box if perform the optional phase), MCC produces 1 box, SDC produces 13 boxes.

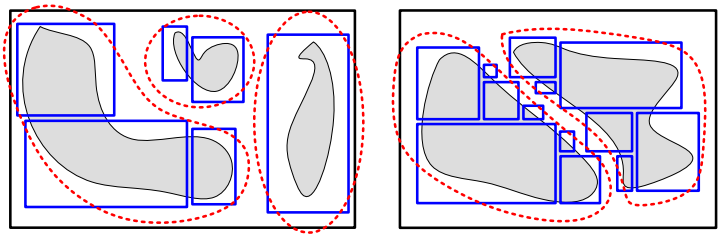

Fig. 14. (a) Applying OSDC to the problem in Figure 11 to get 3 clusters; (b) Applying OSDC to the problem in Figure 13-b to get 2 clusters

Combination of SDC ***and*** COLONIZATION. When p' is small, OMCC will not be efficient. Instead of running COLONIZATION after MCC, we continue MCC until the end of SDC to get n pairwise hull-disjoint clusters. Each of these clusters is a connected set by Proposition 4. Therefore, if we apply COLONIZATION to the hull boxes of these clusters, we will get the optimal max-connected clustering. This combined algorithm is then called *Optimal Separator-Driven Clustering* (OSDC), that combines the adaptiveness of SDC and the optimal max-connectedness of COLONIZATION. The complexity of the second phase of OSDC (i.e. the use of COLONIZATION) is $O(dn^2)$. The total running time of OSDC is $time(\texttt{SDC}) + O(dn^2) = O(mdN + mQ + dn^2)$. Note that $p' \leq p \leq n$. In practice of NCSPs, n is bounded for fixed problems, and if p' is very small then n is small, therefore the running time of OSDC is close to the running time of SDC.

4 Experiments

We now present an evaluation on 16 non-linear problems which have at most four dimensions and which are selected to reflect different topologies of solution set. The selected problems are categorized into three groups: (*i*) problems with only one connected subset; (*ii*) problems with several connected subsets; and (*iii*) problems with tens to hundreds connected subsets. Our experiments show the similarity in the running time of each algorithm in each group. Therefore, we only need to give the average running time and the average number of computed clusters in individual groups on the left and right of the cells, respectively, in Table 1. Figure 15 shows the graph of the average running time.

The results show that the running time of COLONIZATION is quadratic in N (the number of boxes) while the running times of MCC and SDC are very short (the average time of clustering 5000 boxes is less than 50ms, and the running time is always less than 120ms). The running time of the solver that uses the search algorithm in [20] is linear in N. For all problems, the running time of MCC and SDC much less than that of COLONIZATION, and close to zero. In all the tested problems, COLONIZATION and MCC provide the same clusterings. For most tested problems, the three algorithms produce the same clusterings, though SDC is bet-

Table 1. The average running times in milliseconds (on the left of cells) and the average number of clusters (on the right of cells) in three groups

$N \rightarrow$	100	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
(i) Solver	41	205	409	619	823	1044	1285	1519	1701	1952	2245
(i) COLONZ.	2|1.0	48|1.0	195|1.0	430|1.0	779|1.0	1251|1.0	1937|1.0	2762|1.0	3985|1.0	5328|1.0	6734|1.0
(i) MCC	1|1.0	2|1.0	5|1.0	8|1.0	10|1.0	14|1.0	16|1.0	19|1.0	21|1.0	26|1.0	29|1.0
(i) SDC	1|3.3	2|3.3	6|3.3	9|3.3	12|3.3	15|3.3	17|3.3	20|3.3	23|3.3	28|3.3	32|3.3
(ii) Solver	69	285	525	774	1024	1263	1538	1788	2002	2288	2601
(ii) COLONZ.	2|3.1	25|3.6	78|3.7	172|3.9	290|4.0	447|4.1	636|4.1	863|4.4	1120|4.4	1453|4.4	1850|4.4
(ii) MCC	0|3.1	2|3.6	4|3.7	6|3.9	8|4.0	10|4.1	12|4.1	15|4.4	16|4.4	19|4.4	21|4.4
(ii) SDC	0|5.3	2|11.8	4|11.3	7|11.2	9|10.4	11|10.3	13|11.0	15|9.7	17|9.0	19|9.0	22|9.0
(iii) Solver	66	415	860	1253	1683	2112	2557	2935	3371	3883	4271
(iii)COLONZ.	1|9.0	19|24.5	46|31.5	83|47.3	148|69.8	229|97.3	310|116.8	420|116.8	579|116.8	740|116.8	949|116.8
(iii) MCC	1|9.0	6|24.5	12|31.5	18|47.3	24|69.8	31|97.3	38|116.8	41|116.8	42|116.8	46|116.8	49|116.8
(iii) SDC	1|26.8	7|79.5	12|109.5	18|118.5	24|131.3	32|136.5	39|116.8	41|116.8	43|116.8	46|116.8	50|116.8

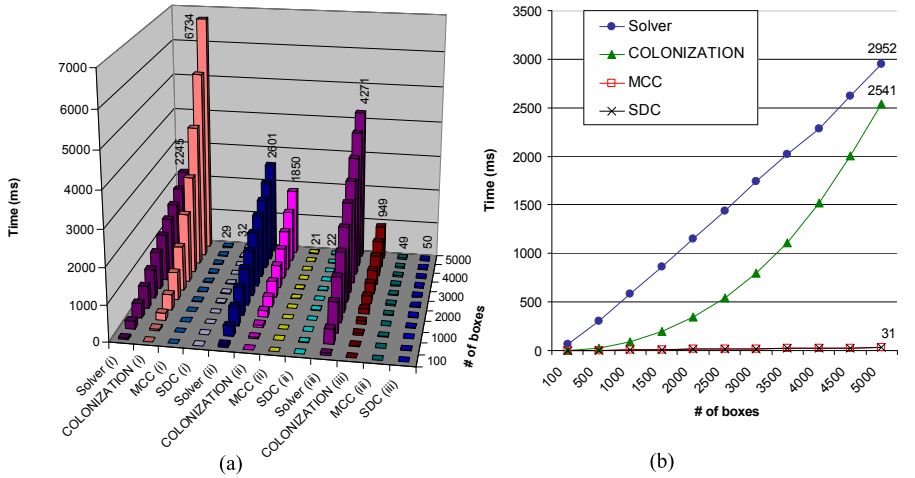

Fig. 15. The average running times in milliseconds (a) in three groups; (b) for all problems

ter than the others in the following cases. They only show significant differences in the following problems: $F2.4 = \{\sin(x \sin y) \geq \cos(y \cos x); -4 \leq x, y \leq 4\}$; $G1.2 = \{x_1^2 + 0.5x_2 + 2(x_3 - 3) \geq 0, x_1^2 + x_2^2 + x_3^2 \leq 25, \forall i : -8 \leq x_i \leq 8\}$; $H1.1 = \{x_1^2 + x_2^2 + x_3^2 \leq 9, (x_1 - 0.5)^2 + (x_2 - 1)^2 + x_3^2 \geq 4, x_1^2 + (x_2 - 0.2)^2 \geq x_3, \forall i : -4 \leq x_i \leq 4\}$. For example, when $N = 1000$, `COLONIZATION` and `MCC` provide six boxes for the problem $F2.4$, the volume ratio of the six boxes to the hull of primitive boxes is 0.761 while the ratio obtained by `MCC` is 0.385 with 30 boxes. When $N = 1000$ for problem $G1.4$ (and $H1.1$ respectively), `COLONIZATION` and `MCC` produce no reduction while `SDC` produces 4 (5, respectively) boxes with the volume ratio is 0.225 (0.191, respectively). The experiments show that even in case `COLONIZATION` and `MCC` cannot characterize the solution set well (e.g. the problems $G1.4$ and $H1.1$), `SDC` still can provide an adaptive clustering that are more suitable for applications. We have only done experiments with three fundamental algorithms (`COLONIZATION`, `MCC` and `SDC`), however the performance of `OMCC` and `OSDC` could be deduced from the performance of `MCC` and `SDC` respectively, and the performance of `COLONIZATION` for a small number of boxes (less than 120 boxes in our experiments, then takes less than 2-3ms).

5 Conclusion

Three algorithms and their combinations, that address different goals, have been proposed to regroup the verbose output of interval-based solvers into a fewer number of clusters. In our experiments, `SDC` shows to be the best among the techniques we proposed and tested. Moreover, `MCC` and `SDC` are very cheap (in running time) postprocessing techniques to get useful grouping information for further process. We also can deduce that, for applications that require the optimal max-connected clustering, either `OSDC` or `OMCC` could be alternative to `COLONIZATION` in case the number of clusters is small or big, respectively. We are investigating a cooperation of optimization and numerical constraint satisfaction that could exploit these fast postprocessing techniques to make the interval-based solvers useful in the context of optimization. Potentially, applying these postprocessing techniques to the output of interval-based solvers makes it possible to use the solvers in various applications (as mentioned in Section 1) that require concise representations of the solution set.

6 Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This research is funded by the European Commission and the Swiss Federal Education and Science Office through the COCONUT project (IST-2000-26063).

References

1. Agarwal, P., de Berg, M., Gudmundsson, J., Hammar, M., Haverkort, H.: Box-Trees and R-Trees with Near-Optimal Query Time. In: Proceedings of the 17th ACM Symposium on Computational Geometry, ACM Press (2001) 124–133
2. van den Bergen, G.: Efficient Collision Detection of Complex Deformable Models using AABB Trees. Journal of Graphics Tools **4(2)** (1997) 7–25
3. Fahn, C.S., Wang, J.L.: Efficient Time-Interrupted and Time-Continuous Collision Detection Among Polyhedral Objects in Arbitrary Motion. Journal of Information Science and Engineering **15** (1999) 769–799
4. Zhou, Y., Suri, S.: Analysis of a Bounding Box Heuristic for Object Intersection. In: Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA'99). (1999)
5. Ganovelli, F., Dingliana, J., O'Sullivan, C.: BucketTree: Improving Collision Detection Between Deformable Objects. In: Spring Conference on Computer Graphics (SCCG'2000). (2000)
6. Larsson, T., Akenine-Moller, T.: Collection Detection for Continuously Deforming Bodies. In: Eurographics'2001, Manchester, UK (2001)
7. Haverkort, H., de Berg, M., Gudmundsson, J.: Box-Trees for Collision Checking in Industrial Installations. In: Proceedings of the 18th ACM Symposium on Computational Geometry, ACM Press (2002) 53–62
8. Fahn, C.S., Wang, J.L.: Adaptive Space Decomposition for Fast Visualization of Soft Object. Journal of Visualization and Computer Animation **14(1)** (2003) 1–19

9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
10. Labroche, N., Monmarché, N., Venturini, G.: A New Clustering Algorithm Based on the Chemical Recognition System of Ants. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'2002), France, IOS Press (2002) 345–349
11. Benhamou, F., Goualard, F.: Universally Quantified Interval Constraints. In: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000). (2000) 67–82
12. Kearfott, B.: Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin **2(1)** (1996) 95–112
13. Burkill, J.: Functions of Intervals. Proceedings of the London Mathematical Society **22** (1924) 375–446
14. Sunaga, T.: Theory of an Interval Algebra and its Applications to Numerical Analysis. RAAG Memoirs **2** (1958) 29–46
15. Moore, R.: Interval Arithmetic and Automatic Error Analysis in Digital Computing. PhD thesis, Stanford University, USA (1962)
16. Moore, R.: Interval Analysis. Prentice Hall, Englewood Cliffs, NJ (1966)
17. Hickey, T., Ju, Q., Van Emden, M.: Interval Arithmetic: from Principles to Implementation. Journal of the ACM (JACM) **48(5)** (2001) 1038–1068
18. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. First edn. Springer (2001) ISBN 1-85233-219-0.
19. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Approximation techniques for non-linear problems with continuum of solutions. In: Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002). Volume 2371 of LNAI., Canada, Springer-Verlag (2002) 224–241
20. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In: Global Optimization and Constraint Satisfaction: First International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS 2002), France, October 2-4, 2002, Revised Selected Papers. Volume 2861 of LNCS., Springer-Verlag (2003) 194–210
21. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Search Techniques for Non-linear CSPs with Inequalities. In: Proceedings of the 14th Canadian Conference on AI. (2001)
22. Lhomme, O.: Consistency Techniques for Numeric CSPs. In: Proceedings of IJCAI'93. (1993)
23. Van Hentenryck, P.: A Gentle Introduction to NUMERICA. Artificial Intelligence **103** (1998) 209–235
24. Collavizza, H., Delobel, F., Rueher, M.: A Note on Partial Consistencies over Continuous Domains. In: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP'98). (1998)
25. Granvilliers, L., Goualard, F., Benhamou, F.: Box Consistency through Weak Box Consistency. In: Proceedings of ICTAI'99. (1999)
26. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising Hull and Box Consistency. In: Proceedings of the International Conference on Logic Programming (ICLP'99), Las Cruces, USA (1999) 230–244
27. Sam-Haroud, D., Faltings, B.: Consistency Techniques for Continuous Constraints. Constraints **1(1-2)** (1996) 85–118
28. Lottaz, C.: Collaborative Design using Solution Spaces. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland (2000)
29. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)