Branch-and-Prune Search Strategies for Numerical Constraint Solving

Xuan-Ha VU[1], Marius-Călin SILAGHI[2], Djamila SAM-HAROUD[3] and Boi FALTINGS[3]

[1] Cork Constraint Computation Centre, University College Cork (UCC),
14 Washington Street West, Cork, Ireland,
Email: ha.vu@4c.ucc.ie;

[2] Department of Computer Science, Florida Institute of Technology (FIT),
150 West University Boulevard, Melbourne, FL-32901-6975, USA
Email: marius.silaghi@cs.fit.edu;

[3] Artificial Intelligence Laboratory, Ecole Polytechniqe Fédéral de Lausanne (EPFL),
Batiment IN, Station 14, CH-1015 Lausanne, Switzerland,
Email: jamila.sam@epfl.ch, boi.faltings@epfl.ch.

When solving *numerical constraints* such as nonlinear equations and inequalities, solvers often exploit *pruning* techniques, which remove redundant value combinations from the domains of variables, at *pruning* steps. To find the *complete* solution set, most of these solvers alternate the *pruning* steps with *branching* steps, which split each problem into subproblems. This forms the so-called *branch-and-prune* framework, well known among the approaches for solving numerical constraints. The basic branch-and-prune search strategy that uses *domain bisections* in place of the branching steps is called the *bisection search*. In general, the bisection search works well in case (*i*) the solutions are isolated, but it can be improved further in case (*ii*) there are continuums of solutions (this often occurs when inequalities are involved). In this paper, we propose a new branch-and-prune search strategy along with several variants, which not only allow yielding better branching decisions in the latter case, but also work as well as the bisection search does in the former case. These new search algorithms enable us to employ various pruning techniques in the construction of inner and outer approximations of the solution set. Our experiments show that these algorithms speed up the solving process often by one order of magnitude or more when solving problems with continuums of solutions, while keeping the same performance as the bisection search when the solutions are isolated.

Categories and Subject Descriptors: F.4.1 [**Theory of Computation**]: Mathematical Logic and Formal Languages—*Logic and Constraint Programming*; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods, and Search*

General Terms: Algorithms, Theory, Computation

Additional Key Words and Phrases: Constraint Programming, Constraint Satisfaction, Search, Numerical Constraint, Interval Arithmetic

Preliminary parts of this paper have been published in [Silaghi et al. 2001; Vu et al. 2002, 2003]. This research was mainly carried out at the Artificial Intelligence Laboratory, EPFL, and was funded by the European Commission and the Swiss Federal Education and Science Office through the COCONUT project (IST-2000-26063).

1. INTRODUCTION

A *constraint satisfaction problem* (CSP) consists of a finite set of constraints specifying which value combinations from given *domains* of its variables are admitted. It is called a *numerical constraint satisfaction problem* (NCSP) if the domains are continuous. NCSPs such as systems of equations and inequalities arise in many industrial applications. A method for solving NCSPs is called *complete* if every solution can be found by it in the infinite time and can be approximated by it within an arbitrarily small positive tolerance after a finite time, provided that the underlying arithmetic is exact. Most available complete solution methods are instances of the *branch-and-prune* framework, which interleaves branching steps with pruning steps. Roughly speaking, a *branching* step divides a problem into subproblems whose union is equivalent to the initial problem in term of the solution set, and a *pruning* step reduces a problem in some measure. The reader can find a more detailed discussion on *branch-and-prune* methods in [Vu 2005, Section 3.2].

The need for completeness arises in many applications such as safety verification and computer-assisted proofs [Schichl 2003, Section 3.1]. In design applications such as estimation and robust control [Asarin et al. 2000; Jaulin et al. 2001], automation and robotics [Jaulin et al. 2001; Lee et al. 2002; Lee and Mavroidis 2002, 2004; Neumaier and Merlet 2002], civil engineering [Lottaz 2000; Vu 2005], and shape design [Snyder 1992], the solution set of an NCSP often expresses equally relevant choices that need to be identified as precisely and as completely as possible. In a design process, one often desires to find as many solutions as possible because investigating many solutions at earlier stages potentially increases the chance of success at later stages. This also allows identifying good design choices. Thus, the complete solution set is often sought for, provided that the response time is reasonable. Since a general NCSP is NP-hard, the time for finding all solutions at a high precision is often prohibitively long. In most cases, a low or medium precision is however sufficient for applications. Hence, there is a tradeoff between timely but less precise information and slow but more precise information.

The necessary background is presented in Section 2. The rest of the paper is organized as follows. We first prepare general settings about solution representations and formal definitions of domain reduction and splitting operators in Section 2.4 and Section 3, respectively. We then present, in Section 4, a generic branch-and-prune search algorithm, called **BnPSearch**, that enables the incorporation of domain reduction and splitting operators (Section 4.1), and then present, as instances of **BnPSearch**, the bisection search (Section 4.2) and two new search algorithms, called **UCA5** and **UCA6** (Section 4.3). Roughly speaking, the effectiveness of the new algorithms is mainly due to the following policies, if some constraint, C, is predicted to contain continuums of feasible points:

—**Working on the negation of constraints to find feasible regions.** Apply *domain reduction* techniques to the negation of C. Let $\mathbf{x}$ be the input domains and $\mathbf{x}'$ the resulting domains. Then $\mathbf{x}^* := \mathbf{x} \setminus \mathbf{x}'$ is feasible w.r.t. C. Thus, C can be immediately removed from subproblems defined on subregions of $\mathbf{x}^*$.

—**Reducing the number of variables in subproblems.** In a subproblem generated during the solving process, some variables may not occur in any constraint under consideration and thus no longer need to be considered.

—**Taking the influence of all constraints on each other into account at each iteration.** Potentially, this reduces the search space better than solving each constraint at a time does.

A simple heuristic to predict whether a constraint contains continuums of solutions is to check if this constraint is an inequality. In general, **UCA5** and **UCA6** allow better branching decisions than the basic domain bisection. Although providing an accurate representation of solutions, in some cases these algorithms are still slow and provide verbose representations. The first reason is that the orthogonal splitting policy in these algorithms generates a significant number of nearly aligned *boxes* near the boundaries of constraints. The second reason is that these algorithms often have to spend too much effort on producing too small boxes with respect to the precision ε, which is predetermined by users.

We later propose, in Section 5, an improved instance, called **UCA6**$^+$, of **BnPSearch** to tackle the above two limitations. The improvement in **UCA6**$^+$ is twofold. First, **UCA6**$^+$ utilizes domain reduction techniques better when the precision is recognized as being sufficient. Namely, it tells domain reduction techniques to avoid unnecessarily spending too much effort on reducing the domains whose sizes are smaller than ε. When the sizes of a certain number of domains are smaller than ε, **UCA6**$^+$ allows resorting to a simple solver that is more efficient for small and very low dimensional problems. The gain is then in both the computational time and the alignment of boxes. Second, **UCA6**$^+$ allows resorting to geometric representation techniques to combine aligned boxes produced in the previous stage into larger equivalent boxes. The representation of the solution set is therefore more concise. This potentially accelerates querying and complicated operations on the explicit representation of solutions.

In general, our new search algorithms improve the solving process when there are continuums of solutions, and keep the same procedure and performance as the bisection search when the solutions are isolated. In the former case, our search algorithms allow producing *inner* and *outer approximations* w.r.t. a predetermined precision ε. Moreover, a large percentage of the outcome are often proved to be *sound* solutions. Our experiments in Section 6 show that the new search algorithms significantly improve the efficiency as well as the conciseness of solution representations. The conclusion is finally given in Section 7.

2. BACKGROUND AND DEFINITION

2.1 Numerical Constraint Satisfaction

We recall in this section two central concepts of constraint programming.

Definition 2.1. A *constraint* on a finite sequence of variables, $(x_1, \ldots, x_k)$, taking their values in respective domains, $(D_1, \ldots, D_k)$, is a subset of the Cartesian product $D_1 \times \cdots \times D_k$, where $k \in \mathbb{N}$ ($\mathbb{N}$ is the set of natural numbers).

Definition 2.2 *CSP.* A *constraint satisfaction problem*, abbreviated to CSP, is a triple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ in which $\mathcal{V}$ is a finite sequence of variables $(v_1, \ldots, v_n)$, $\mathcal{D}$ is a finite sequence of the respective domains of the variables $(v_1, \ldots, v_n)$, and $\mathcal{C}$ is a finite set of constraints, each on a subsequence of $\mathcal{V}$. A *solution* of this problem is an

assignment of values from $\mathcal{D}$ to $\mathcal{V}$ respectively such that all constraints in $\mathcal{C}$ are satisfied. The set of all solutions is called the *solution set*.

The reader can find many more concepts in [Apt 2003]. In this paper, we only focus on numerical CSPs, which are defined as follows.

Definition 2.3 *NCSP.* A domain is said to be *continuous* if it is a real interval. A *numerical constraint* is a constraint on a sequence of variables whose domains are continuous. If all constraints of a CSP are numerical, it is called a *numerical constraint satisfaction problem* (abbreviated to NCSP).

An NCSP can be viewed as a constrained optimization problem with a constant objective function. Thus, it can be theoretically solved by using *mathematical programming* (MP) methods for solving constrained optimization problems. However, most of the efficient MP methods are heavily based on the influence between objective functions and constraints, thus not efficient for directly solving NCSPs.

Since thirty years ago, *constraint satisfaction* techniques have been being devised to solve CSPs with discrete domains. These techniques perform *reasoning* procedures on constraints and explore the search space by intelligently enumerating solutions. In order to solve NCSPs by means of constraint satisfaction, continuous domains have often been converted into discrete domains by using progressive *discretization* techniques [Sam-Haroud 1995; Lottaz 2000]. Still, these methods are often inefficient. Later on, many mathematical *computing* techniques for continuous domains have been integrated into the framework of constraint satisfaction in order to solve NCSPs more efficiently. Nowadays, these techniques are often referred to as *constraint programming* techniques, which imply the combination of *computing* and *reasoning* aspects. A much more extensive discussion on numerical constraint solving can be found in [Vu 2005].

2.2 Interval Arithmetic

Let $\mathbb{R}_\infty \equiv \mathbb{R} \cup \{-\infty, +\infty\}$. The *lower bound* of a real interval $\mathbf{x}$ is defined as $\inf(\mathbf{x}) \in \mathbb{R}_\infty$, and the *upper bound* of $\mathbf{x}$ is defined as $\sup(\mathbf{x}) \in \mathbb{R}_\infty$. Let denote $\underline{x} = \inf(\mathbf{x})$ and $\overline{x} = \sup(\mathbf{x})$. There are four possible intervals $\mathbf{x}$ with these bounds:

- The *closed interval* defined as $\mathbf{x} \equiv [\underline{x}, \overline{x}] \equiv \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$;
- The *open interval* defined as $\mathbf{x} \equiv]\underline{x}, \overline{x}[\equiv \{x \in \mathbb{R} \mid \underline{x} < x < \overline{x}\}$;
- The *left-open interval* defined as $\mathbf{x} \equiv]\underline{x}, \overline{x}] \equiv \{x \in \mathbb{R} \mid \underline{x} < x \leq \overline{x}\}$;
- The *right-open interval* defined as $\mathbf{x} \equiv [\underline{x}, \overline{x}[\equiv \{x \in \mathbb{R} \mid \underline{x} \leq x < \overline{x}\}$.

Let $\mathbb{I}$ be the set of all closed intervals and $\mathbb{I}_\circ$ the set of all intervals. The *interval hull* of a subset S of $\mathbb{R}$ is the smallest interval (w.r.t. the set inclusion), denoted as $\square S$, that contains S. For example, $\square(]1,3] \cup \{2,4\}) =]1,4]$. Given a nonempty interval $\mathbf{x}$, we define that the *midpoint* of $\mathbf{x}$ is $\mathrm{mid}(\mathbf{x}) \equiv (\inf(\mathbf{x}) + \sup(\mathbf{x}))/2$ and the *width* of $\mathbf{x}$ is $\mathrm{w}(\mathbf{x}) \equiv \sup(\mathbf{x}) - \inf(\mathbf{x})$. We also agree that $\mathrm{w}(\emptyset) = 0$ and $\mathrm{mid}(\emptyset) = 0$. A *box* is the Cartesian product of a finite number of intervals. The concepts of the midpoint and width are defined on *boxes* in a componentwise manner.

Fundamentally, if $\mathbf{x}$ and $\mathbf{y}$ are two (real) intervals, then the four elementary operations for *idealized interval arithmetic* obey the rule

$$\mathbf{x} \diamond \mathbf{y} \equiv \{x \diamond y \mid x \in \mathbf{x}, y \in \mathbf{y}\}, \quad \forall \diamond \in \{+, -, *, \div\}. \tag{1}$$

Thus, the results of the four elementary operations in interval arithmetic are exactly the ranges of their real-valued counterparts. Although the rule (1) characterizes these operations mathematically, the usefulness of interval arithmetic is due to the *operational definitions* based on interval bounds. For example, let $\mathbf{x} = [\underline{x}, \overline{x}]$ and $\mathbf{y} = [\underline{y}, \overline{y}]$ be two closed intervals, interval arithmetic shows that:

$$\mathbf{x} + \mathbf{y} \equiv [\underline{x} + \underline{y}, \overline{x} + \overline{y}]; \tag{2}$$

$$\mathbf{x} - \mathbf{y} \equiv [\underline{x} - \overline{y}, \overline{x} - \underline{y}]; \tag{3}$$

$$\mathbf{x} * \mathbf{y} \equiv [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]; \tag{4}$$

$$\mathbf{x} \div \mathbf{y} \equiv \mathbf{x} * (1/\mathbf{y}) \text{ if } 0 \notin \mathbf{y}, \text{ where } 1/\mathbf{y} \equiv [1/\overline{y}, 1/\underline{y}]. \tag{5}$$

Simple *arithmetic expressions* are composed of these four elementary operations.

An *interval form*, $\mathbf{f} : \mathbb{I}^m_\circ \to \mathbb{I}^n_\circ$, of a real function $f : D \subseteq \mathbb{R}^m \to \mathbb{R}^n$ is constructed conforming to the *inclusion property*: the value of the interval form encloses the exact range of the real function, that is, $\forall \mathbf{x} \in \mathbb{I}^m_\circ : f(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{x})$ or, equivalently, $\forall \mathbf{x} \in \mathbb{I}^m_\circ, x \in D : x \in \mathbf{x} \Rightarrow f(x) \in \mathbf{f}(\mathbf{x})$.

The finite nature of computers precludes an exact representation of the real numbers. The real set $\mathbb{R}$ is therefore approximated by a finite set $\mathbb{F}$ of *floating-point numbers* [Goldberg 1991], including $-\infty$ and $+\infty$. The set of real intervals is then replaced with the set $\mathbb{I}_\diamond$ of closed *floating-point intervals* with bounds in $\mathbb{F}$. The interval concepts are similarly defined on $\mathbb{I}_\diamond$ while conforming to the inclusion property. The power of interval arithmetic lies in its implementation on computers. In particular, *outwardly rounded* interval arithmetic allows computing *rigorous enclosures* of the ranges of functions. An interval is said to be *canonical* iff it does not contain two different intervals whose union is not an interval. A box is said to be *canonical* iff all its intervals are canonical.

The reader can find extended introductions to interval analysis in [Moore 1966, 1979; Alefeld and Herzberger 1983], interval methods for systems of equations in [Neumaier 1990], interval methods for optimization in [Hansen and Walster 2004], and some recent applications of interval arithmetic in [Jaulin et al. 2001].

Most interval arithmetic libraries have been implemented for closed intervals only. One however can use these libraries to perform some computations on open/closed intervals. Indeed, every interval $\mathbf{x}$ with bound values $\underline{x}$ and $\overline{x}$ is contained in the corresponding closed interval $[\underline{x}, \overline{x}] \in \mathbb{I}$. If the computations are domain reduction or complementary boxing, we can perform these computations on the corresponding closed intervals of the domains to get new domains, and then take the set intersection of these new domains with the initial general intervals. For example, after performing a domain reduction technique on the closed interval $[\underline{x}, \overline{x}]$, we get a closed interval $[\underline{y}, \overline{y}] \subseteq [\underline{x}, \overline{x}]$. The result to be obtained is the set intersection $\mathbf{x} \cap [\underline{y}, \overline{y}]$, thus may be open or closed.

2.3 A Short Overview of Branch-and-Prune Solution Methods

To be able to find the *complete* solution set of an NCSP, most solvers follow the *branch-and-prune* framework, a well known approach for solving numerical constraints. In this framework, a solver alternates pruning steps with branching steps until reaching the required precision, where a *pruning* step attempts to reduce each considered problem in some measure and a *branching* step splits each considered

problem into subproblems. The basic branch-and-prune search that uses *domain bisections* in place of the branching steps is called the *bisection search*.

A pruning technique that attempts to reduce the domains of problems without discarding any solution is called a *domain reduction* technique. In contrast to a domain reduction technique, a *test* (e.g., an existence test, a uniqueness test, an exclusion test and an inclusion test) does not change the domains of an input problem but maps this problem to a predetermined status. In particular, *existence*, *uniqueness* and *exclusion* tests are to check if a problem has at least one solution, a unique solution and no solution, respectively. The outcome of these tests is thus a Boolean value. Quite differently, an *inclusion test* is to check if all points under consideration are solutions, non-solutions, or else; which is defined as follows.

Definition 2.4. Let X be a sequence of n real variables. An *inclusion test* is a function τ that takes as input a (domain) box $\mathbf{x} \in \mathbb{I}_\circ^n$ and a finite set $\mathcal{C}$ of constraints on a subsequence Y of X (assuming Y is well defined on $\mathbf{x}$) and that returns either `feasible`, `infeasible`, or `unknown` such that:

(1) If $\tau(\mathbf{x}, \mathcal{C}) =$ `feasible`, then every point in $\mathbf{x}$ satisfies all constraints in $\mathcal{C}$.
(2) If $\tau(\mathbf{x}, \mathcal{C}) =$ `infeasible`, then no point in $\mathbf{x}$ satisfies all constraints in $\mathcal{C}$.

The inclusion test τ is said to be *trivial* if it always returns `unknown`. It is said to be ε-*strong* if, for every $\mathbf{x}$, the truth of $\mathrm{w}(\mathbf{x}) \leq \varepsilon \wedge \tau(\mathbf{x}, \mathcal{C}) =$ `unknown` implies the existence of a feasible point of $\mathcal{C}$ in $\mathbf{x}$.

An extended overview of fundamental and recent complete methods for solving NCSPs has been presented in [Vu 2005, Chapter 3]. In summary, those methods can be viewed as instances of the branch-and-prune framework. Many of them integrate existence, uniqueness or exclusion tests at pruning steps of the bisection search. At a branching step, they bisect a domain, which is often chosen as the largest with respect to some measure (e.g., domain size), provided that this domain is amenable to be split (e.g, its size is greater than a predetermined precision ε). Because most solution methods have been designed to solve a square system of equations, they only aim at generating a collection of tiny boxes, each encloses a solution. This approach is referred to as the *point-wise* approach. It may be reasonable for solving NCSPs with *isolated solutions* (see Figure 1a), but are often inefficient, when applied in a straightforward manner, for solving NCSPs with *continuums of solutions* (see Figure 1b). In the latter case, neither the computational time nor the compactness of the solution representation are satisfactory.

It is possible to enhance the solving process in the point-wise approach by replacing the existence, uniqueness and exclusion tests with domain reduction techniques in a straightforward manner. In the rest, the resulting search strategy will be called *dichotomous maintaining bounds by consistency* (**DMBC**). When solving NCSPs with non-isolated solutions, **DMBC** search techniques may be able to cover a spectrum of non-isolated solutions with a number of subsets of $\mathbb{R}^n$. However, it is often not possible to prove that a subset of the outcome are all solutions.

In contrast to the point-wise approach, another one, called the *set-covering* approach, has been developed in order to represent continuums of solutions more reasonably. It aims at covering, as accurately as possible, continuums of solutions with inner and outer approximations, each consists of a number of subsets of $\mathbb{R}^n$.

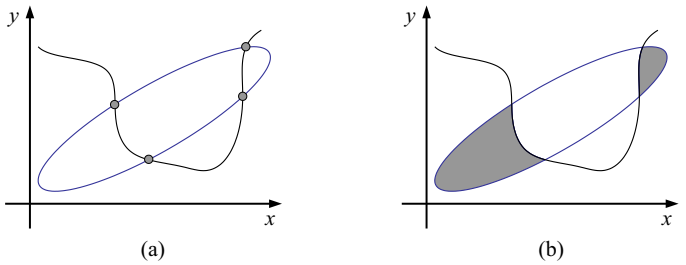

Fig. 1. (a) An NCSP with four isolated solutions (grey dots); (b) An NCSP with two continuums of solutions (grey regions).

All points represented by the inner approximation are proved to be solutions. Usually, the representation of inner and outer approximations is made simple such that the costs of usual operations on this representation are as cheap as possible. The subsets in these approximations are often chosen to be simple ones (e.g., boxes). Owning to their nice properties, boxes have been used in many set-covering techniques, thus forming the so-called *box-covering* techniques.

For simplicity, in this paper we restrict our attention to solution methods that use boxes as elements of the outcome. One has often employed inclusion tests at pruning steps of typical box-covering branch-and-prune methods to prove that all points in some subsets are solutions. In this paper, a **DMBC**-like search technique that combines both domain reduction techniques and inclusion tests at its pruning steps is called a **DMBC**$^+$ search technique (see Section 4.2). Therefore, **DMBC**$^+$ search techniques may, depending on the strong of employed inclusion tests, be able to provide inner approximations of the solution set. We hence say that **DMBC**$^+$ search techniques belong to the box-covering approach and that **DMBC** search techniques belong to the point-wise approach. Note that both **DMBC** and **DMBC**$^+$ search strategies do not remove constraints from consideration during the solving process. That is, they always consider the whole set of initial constraints when resorting to domain reduction techniques and tests.

When solving an NCSP with continuums of solutions at a high precision, **DMBC** techniques often provide a huge collection of tiny boxes as an outer approximation of the solution set while **DMBC**$^+$ techniques are usually able to provide both inner and outer approximations of the solution set, each approximation is a much more concise collection of boxes. However, the number of boxes used by **DMBC**$^+$ techniques to approximate the boundary of continuums of solutions is still very high in many cases. Therefore, either their applicability is restricted or the tractability limit is rapidly reached. The new search techniques proposed in this paper will tackle this limitation (see Section 4 and Section 5).

Because most computers were equiped with floating-point number systems, an important issue is how to deal with rounding errors occurring in computations on floating-point numbers. Owning to the inclusion property of outwardly rounded interval arithmetic, one has often been using it to tackle the issue of rounding errors. Recent interval arithmetic based methods are able to solve a number of NCSPs efficiently while still enjoying the completeness. Those methods can be

viewed as instances of the branch-and-prune framework, although they mainly focus on improving the pruning steps.

Interval constraint solvers such as CLP(BNR) [Benhamou and Older 1992, 1997], Numerica [Van Hentenryck 1998] and ILOG Solver [ILOG 2003] have shown their ability to efficiently find all solutions of certain instances of NCSPs within an arbitrary positive tolerance. They are instances of the branch-and-prune framework, and most of them use a simple branching policy like the domain bisection or dichotomization as their default branching policy while leaving more advanced branching policies to users. In other words, they essentially follow the point-wise approach and only aim at solving NCSPs with isolated solutions.

Recently, a number of box-covering methods have been developed in [Jaulin 1994; Jaulin et al. 2001], [Sam-Haroud and Faltings 1996], [Garloff and Graf 1999], [Collavizza et al. 1999] and [Benhamou and Goualard 2000; Benhamou et al. 2004] in order to represent continuums of solutions. Although those methods are more suitable for dealing with continuums of solutions than the point-wise methods are, their branching policies still have at least one of the following limitations:

—**They are not complete methods in general.** For example, Collavizza et al. [1999] have proposed a technique to extend a known feasible box of an inequality of the form $f(x) \leq 0$ by performing box consistency (a kind of domain reduction) on its *associated equation*, $f(x) = 0$. Unfortunately, their results (Proposition 1 and 2 in that paper) do not hold for general constraints.

—**They are only designed for special constraints.** For example, the technique proposed in [Garloff and Graf 1999] uses Bernstein polynomials to construct algebraic inclusion tests for use in a **DMBC**/**DMBC**$^+$ search, and is restricted to polynomial constraints. The technique proposed in [Sam-Haroud and Faltings 1996; Lottaz 2000] is restricted to the class of NCSPs with convexity properties. The technique proposed in [Benhamou and Goualard 2000; Benhamou et al. 2004] is originally designed for solving universally quantified constraints.

—**They do not fully exploit the power of domain reduction techniques.** Namely, they only interleave *inclusion tests* with uniformly splitting policies on all variables: each box produced by the splitting is tested for inclusion. The outcome can be structured into the form of a 2^k*-tree*. In [Jaulin 1994], this process is performed in the space of all variables. However, in [Sam-Haroud and Faltings 1996; Lottaz 2000], only binary and ternary constraints obtained by *ternarizing*[1] the initial NCSP are considered for the construction of 2^k-trees.

—**They do not sufficiently take the influence of constraints on each other into account during the solving process.** For example, the solution method in [Sam-Haroud and Faltings 1996; Lottaz 2000] construct *quadtrees* and *octrees* for individual binary and ternary constraints, respectively, and finally perform a constraint propagation on those trees. On the other hand, at each iteration the solution method in [Benhamou and Goualard 2000; Benhamou et al. 2004] considers a constraint and solves it within every (domain) box produced as the outcome of the previous iteration.

[1]Ternarizing an NCSP is to recursively replace each binary arithmetic subexpression with an auxiliary variable until the arity of all the resulting expressions is at most three.

Most methods represent the solution set of an NCSP *explicitly* in the space of initial variables, thus suffering from the high space complexity when there are continuums of solutions. The only one exception we know of is the work of Sam-Haroud and Faltings [1996] and Lottaz [2000], in which the authors have proposed to replace the explicit representation (in the space of initial variables) with a *semi-explicit* one, which is maintained by a number of *quadtrees* and *octrees*. Although that semi-explicit representation reduces the space complexity, it increases the querying time for a solution. Notice that the uniformity of splitting must be maintained in those methods in order to do propagation among 2^k-trees. Therefore, the power of domain reduction cannot be fully exploited in this method.

One of the most recent improvements to search is the work in [Benhamou and Goualard 2000; Benhamou et al. 2004], which is summarized as follows. In order to find feasible regions of universally quantified constraints of the form $\forall t \in D_t : f(x,t) \leq 0$, Benhamou and Goualard [2000]; Benhamou et al. [2004] have proposed to perform a kind of domain reduction on their negation, $f(x,t) > 0$. This operation is called a *negation test*. It encloses all possibly infeasible regions and the remaining is feasible. Now consider a subproblem living in a domain box $\mathbf{x}$. A procedure, called **ICAb3**$_c$ in [Benhamou and Goualard 2000], takes as input a constraint, C. The procedure **ICAb3**$_c$ performs a negation test on C to reduce $\mathbf{x}$ to a new domain $\mathbf{x}'$. If $\mathbf{x}'$ is an empty set, then every point in $\mathbf{x}$ satisfies C; otherwise, split $\mathbf{x} \setminus \mathbf{x}'$ into boxes and then dichotomize $\mathbf{x}'$. Now, C can be removed from all subproblems that do not have domains in $\mathbf{x}'$. The procedure **ICAb3**$_c$ recursively performs the above operations on all resulting subproblems that still have C as a *running constraint*[2] and that have domains larger than a predetermined precision ε. A search method, called **ICAb5** in [Benhamou and Goualard 2000], repeats the procedure **ICAb3**$_c$ for each constraint, one by one, until all constraints have been processed. See also Section 3.2 for more discussion on the negation-based approach.

2.4 Representation of Non-isolated Solutions

2.4.1 *Inner and Outer Approximations.* In case the solution set of an NCSP is empty or consists of isolated points, its representation is usually simple. The representation of the solution set is not simple in other cases, especially when the solution set contains continuums of solutions. In general, the solution set of an NCSP is a relation on $\mathbb{R}^n$, where n is the number of variables in the NCSP. A relation can be theoretically approximated by a superset and/or a subset.

Definition 2.5 *Inner Approximation.* Given a relation, $S \subseteq \mathbb{R}^n$, a set $S^- \subseteq \mathbb{R}^n$ is called an *inner approximation* of S if it is contained in S; that is, $S^- \subseteq S$.

Definition 2.6 *Outer Approximation.* Given a relation, $S \subseteq \mathbb{R}^n$, a set $S^+ \subseteq \mathbb{R}^n$ is called an *outer approximation* of S if it contains S; that is, $S^+ \supseteq S$.

When a relation on $\mathbb{R}^n$, such as the solution set of an NCSP, is approximated by an inner approximation and/or an outer approximation. The latter is a *sound approximation* (i.e., it only contains solutions), but may lose some solutions. Conversely, the former is a *complete approximation* (i.e., it contains all solutions), but

[2] A running constraint is a constraint that is currently still under consideration.

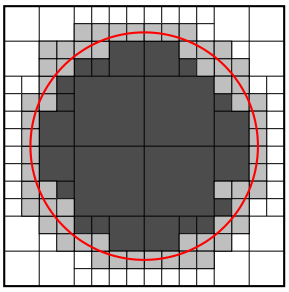

Fig. 2. An example of inner/outer/boundary union approximations of a circle with interior: the collection of the dark grey boxes is an inner union approximation ($\boxplus^{\mathcal{I}}$); the collection of the light grey boxes is a boundary union approximation ($\boxplus^{\mathcal{B}}$); the collection of the light and dark grey boxes is an outer union approximation ($\boxplus^{\mathcal{O}}$).

may contain some points that are not solutions. Given an exact representation $\mathcal{R}$, such as a collection of boxes or a tree of boxes, of a relation $S \subseteq \mathbb{R}^n$, we denote by $\mathsf{pts}(\mathcal{R})$ the set of points in S.

One often uses the volume difference, $\mathrm{vol}(S^+) \setminus \mathrm{vol}(S^-)$, to measure the degree of mismatch between inner and outer approximations. The exact approximation errors are then bounded by this measure.

2.4.2 *Union Approximations.* Since the time for querying a point in a box is constant, one often approximates a relation $S \subseteq \mathbb{R}^n$ by a collection of *pairwise disjoint boxes*, where two boxes are said to be (strictly)[3] *disjoint* if they have no common points. Such a collection is called a *collection of disjoint boxes* for short. The representation of a collection of disjoint boxes which is constructed by enumerating these boxes and storing their coordinates is called the *disjoint box representation* [Aguilera 1998]. Among the approximations by collections of boxes, the following three attract the most attention in practice because of their simplicity.

Definition 2.7. Given a relation $S \subseteq \mathbb{R}^n$. An *inner union approximation* of S, denoted by $\boxplus^{\mathcal{I}}[S]$, is a collection of disjoint boxes in $\mathbb{I}^n_\circ$ such that $S \supseteq \mathsf{pts}(\boxplus^{\mathcal{I}}[S])$.

Definition 2.8. Given a relation $S \subseteq \mathbb{R}^n$. An *outer union approximation* of S, denoted by $\boxplus^{\mathcal{O}}[S]$, is a collection of disjoint boxes in $\mathbb{I}^n_\circ$ such that $S \subseteq \mathsf{pts}(\boxplus^{\mathcal{O}}[S])$.

Definition 2.9. Given a relation $S \subseteq \mathbb{R}^n$. A *boundary union approximation*, denoted by $\boxplus^{\mathcal{B}}[S]$, of S (with respect to an inner union approximation $\boxplus^{\mathcal{I}}[S]$ and an outer union approximation $\boxplus^{\mathcal{O}}[S]$) is a collection of disjoint boxes in $\mathbb{I}^n_\circ$ such that $\mathsf{pts}(\boxplus^{\mathcal{B}}[S]) = \mathsf{pts}(\boxplus^{\mathcal{O}}[S]) \setminus \mathsf{pts}(\boxplus^{\mathcal{I}}[S])$.

Remark 2.10. Notice that $\boxplus^{\mathcal{X}}$ is not a function, where $\mathcal{X} \in \{\mathcal{I}, \mathcal{O}, \mathcal{B}\}$. In this paper, we will always refer to $\boxplus^{\mathcal{B}}[S]$ with respect to some $\boxplus^{\mathcal{I}}[S]$ and some $\boxplus^{\mathcal{O}}[S]$, even when not mentioned explicitly.

[3]The main results in this paper still hold if we relax the disjointness such that disjoint boxes may have common points on their facets but not in their interiors.

The concepts of union approximations are depicted in Figure 2. Note that we always have the identity $\mathsf{pts}(\boxplus^{\mathcal{I}}[S]) \cap \mathsf{pts}(\boxplus^{\mathcal{B}}[S]) = \emptyset$. In practice, one often computes $\boxplus^{\mathcal{I}}[S]$ and $\boxplus^{\mathcal{B}}[S]$ first, and then obtains $\boxplus^{\mathcal{O}}[S]$ simply by $\boxplus^{\mathcal{O}}[S] := \boxplus^{\mathcal{I}}[S] \cup \boxplus^{\mathcal{B}}[S]$.

The worst-case query time of a *bounding-box tree* in $\mathbb{R}^d$ 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* [Agarwal et al. 2001]. It is therefore useful to construct inner and/or outer union approximations of an unknown relation (e.g., the solution set of an NCSP) in the form of a *bounding-box tree*. That is, the box represented by any node of the tree contains the box represented by its child node, and all boxes represented by the children of any node are disjoint. Fortunately, this property is enjoyed by branch-and-prune solution methods assuming that the domains are intervals.

Several authors have recently addressed the construction of inner and/or outer union approximations of the solution set of an NCSP. In [Jaulin 1994], union approximations are hierarchically constructed in the form of a 2^k*-tree* in the space of initial variables. This technique has shown its practical usefulness in robotics, automation and robust control. The method in [Sam-Haroud and Faltings 1996; Lottaz 2000] also aims at the construction of 2^k-trees in a similar way. However, only binary and ternary constraints obtained by ternarizing the initial NCSP are considered for the construction. That is, only *quadtrees* and *octrees* are constructed. The solution set is finally approximated by a number of *quadtrees* and *octrees* rather than a single 2^k-tree. The space complexity of approximations is thus reduced. The approach is however restricted to the class of NCSPs with convexity properties.

Most recently, the method proposed in [Benhamou and Goualard 2000; Benhamou et al. 2004] corrects and extends the idea in [Collavizza et al. 1999] to construct inner union approximations of universally quantified constraints. Namely, when solving a universally quantified constraint of the form $\forall t \in D_t : f(x,t) \leq 0$, the application of a domain reduction technique on the constraint $f(x,t) > 0$ allows finding feasible regions on which an efficient search is based (see Section 3.2).

2.4.3 *The Precision and Accuracy of Union Approximations.* The cost for achieving a given accuracy of approximations is often very high. Alternatively, most constraint solvers stop splitting a box, which represents the domains of a subproblem, as soon as the size of this box is not greater than a given positive precision ε (and this box is called an *ε-bounded box*). Some other solvers may attempt to apply a pruning technique or a test to ε-bounded boxes before classifying them as *undiscernible*; thus, the name *undiscernible box* has come out.

In general, different constraint solvers use different criteria for leaving ε-bounded boxes unprocessed. If a technique that is applied to ε-bounded boxes before claiming them as undiscernible is used by a solver, then it can be used for the other solvers as well. Therefore, the comparison of search techniques should be based on the same criteria of classifying ε-bounded boxes as *undiscernible*. We propose to use monotonic inclusion tests defined in the following definition for this purpose. Let $\mathbf{x}[Y]$ denote the *projection* of a set $\mathbf{x}$ on the subsequence Y of variables.

Definition 2.11 *Monotonicity.* Let use the same notations as in Definition 2.4. The inclusion test τ is said to be *monotonic* if, for every box $\mathbf{x}'$ and every finite set

$\mathcal{C}'$ of constraints on Y such that $\mathbf{x}[Y] \subseteq \mathbf{x}'[Y] \subseteq \bigcap_{C\in\mathcal{C}'} C$, we have

$$\tau(\mathbf{x},\mathcal{C}) = \texttt{unknown} \quad \Rightarrow \quad \tau(\mathbf{x}',\mathcal{C}) = \tau(\mathbf{x}',\mathcal{C}\cup\mathcal{C}') = \texttt{unknown}. \tag{6}$$

Once the monotonicity holds for the value `unknown` of an inclusion test, it also holds for other values as shown below.

THEOREM 2.12. *Let use the same notations as in Definition 2.11. If τ is a monotonic inclusion test, then*

—*If $\tau(\mathbf{x},\mathcal{C}) =$ `feasible` holds, then $\tau(\mathbf{x}',\mathcal{C}) = \tau(\mathbf{x}',\mathcal{C}\cup\mathcal{C}') =$ `feasible` holds for every finite set $\mathcal{C}'$ of constraints on Y and every nonempty box $\mathbf{x}'$ such that $\mathbf{x}'[Y] \subseteq \mathbf{x}[Y] \subseteq \bigcap_{C\in\mathcal{C}'} C$ holds;*

—*If $\tau(\mathbf{x},\mathcal{C}) =$ `infeasible` holds, then $\tau(\mathbf{x}',\mathcal{C}) = \tau(\mathbf{x}',\mathcal{C}\cup\mathcal{C}') =$ `infeasible` holds for every finite set $\mathcal{C}'$ of constraints on Y and every box $\mathbf{x}'$ such that $\mathbf{x}'[Y] \subseteq \mathbf{x}[Y] \subseteq \bigcap_{C\in\mathcal{C}'} C$ holds.*

PROOF. This is easily proved by contradiction based on Definition 2.11. □

Based on the concept of a monotonic inclusion test, we define the precision of union approximations (or a solution algorithm computing them) as follows.

Definition 2.13. Given an NCSP $\mathcal{P} = (\mathcal{V},\mathcal{D},\mathcal{C})$, a precision (vector) $\varepsilon > 0$, and a monotonic inclusion test τ. A solution algorithm that computes inner and boundary union approximations is (and thus those approximations are) said to be of *the precision ε w.r.t. the monotonic inclusion test τ* if the boundary union approximation equals (w.r.t. the set union) to a collection $\mathcal{U}$ of disjoint ε-bounded or canonical boxes in $\mathbb{I}_\circ^n$ such that

$$\forall \mathbf{x} \in \mathcal{U} : \tau(\mathbf{x},\mathcal{C}) = \texttt{unknown}. \tag{7}$$

If τ is trivial, we say for short that the solution algorithm and the computed approximations are of *the precision ε*. If τ is ε-strong, we say that the solution algorithm and the computed approximations are *ε-accurate*.

It is easy to see that a solution algorithm is complete if it is ε-accurate (i.e., τ is ε-strong) for all sufficiently small $\varepsilon > 0$.

3. REDUCTION AND SPLITTING OPERATORS FOR EXHAUSTIVE SEARCH

Our improvements to the classic search strategies (i.e., **DMBC** and **DMBC**$^+$) will presented in Section 4 and Section 5. In order to present those improvements uniformly and concisely, we generalize and modify some previously existing concepts in the next four subsections.

3.1 Domain Reduction Operators

First, we define the concept of a domain reduction operator as follows.

Definition 3.1 *Domain Reduction Operator,* `DR`. Given a sequence X of n real variables associated with domains $\mathcal{D}$. A *domain reduction operator* `DR` for numerical constraints is a function that takes as input a box $\mathbf{x} \in \mathbb{I}_\circ^n$ contained in $\mathcal{D}$ and a

finite set $\mathcal{C}$ of constraints on X, and that returns a box in $\mathbb{I}_\circ^n$, denoted by $\mathtt{DR}(\mathbf{x},\mathcal{C})$, satisfying the following properties:

$$\text{(Contractiveness)} \qquad \mathtt{DR}(\mathbf{x},\mathcal{C}) \subseteq \mathbf{x}, \tag{8}$$

$$\text{(Correctness)} \qquad \mathtt{DR}(\mathbf{x},\mathcal{C}) \supseteq \mathbf{x} \cap \bigcap_{C\in\mathcal{C}} C. \tag{9}$$

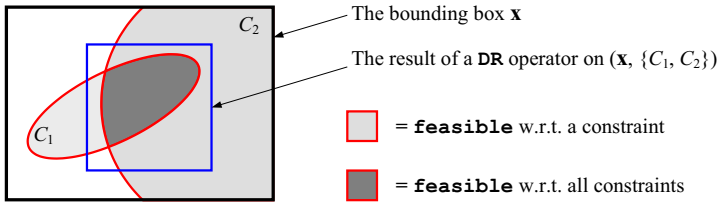

Fig. 3. A domain reduction (DR) operator is applied to a box $\mathbf{x}$ and a constraint set $\{C_1, C_2\}$.

A domain reduction operator has also been referred to as a narrowing operator, contracting operator, or contractor in literature. We adopt the terminology *domain reduction operator*, because it is mnemonic and the terminology *domain reduction* has been widely accepted in many fields, not only in contraint programming.

Definition 3.2 *Monotonicity.* Given a sequence X of n real variables associated with domains $\mathcal{D}$. A domain reduction operator μ is said to be *monotonic* if, for every set $\mathcal{C}$ of constraints on X, we have

$$\forall \mathbf{x},\mathbf{x}' \in \mathbb{I}_\circ^n,\ \ \mathbf{x}\subseteq\mathcal{D},\ \ \mathbf{x}'\subseteq\mathcal{D}\ \ :\ \ \mathbf{x}\subseteq\mathbf{x}'\ \ \Rightarrow\ \ \mu(\mathbf{x},\mathcal{C})\subseteq\mu(\mathbf{x}',\mathcal{C}). \tag{10}$$

In constraint programming, domain reduction operators are usually constructed by enforcing either box consistency [Benhamou et al. 1994], hull consistency [Benhamou and Older 1992, 1997], or kB-consistency [Lhomme 1993]. Although these domain reduction operators enjoy the monotonicity, many domain reduction operators do not enjoy the monotonicity but are still very efficient in practice. The concept of a domain reduction operator is depicted in Figure 3. Other examples of domain reduction operators are depicted in Figure 5. The following property is straightforward but interesting for constraint solving.

THEOREM 3.3. *Given a set $\mathcal{C}$ of constraints on a sequence of n real variables associated with domains $\mathcal{D}$. Suppose $\mathbf{x}\in\mathbb{I}_\circ^n$ is a box contained in $\mathcal{D}$. If there exists a domain reduction operator* DR *that maps $(\mathbf{x},\mathcal{C})$ to an empty set (i.e., $\mathtt{DR}(\mathbf{x},\mathcal{C})=\emptyset$), then $\mathcal{C}$ is inconsistent in $\mathbf{x}$; that is,*

$$\mathbf{x}\subseteq\neg\mathcal{C} \qquad (\textit{where } \neg\mathcal{C}\equiv\mathcal{D}\setminus\bigcap_{C\in\mathcal{C}} C). \tag{11}$$

PROOF. It follows from the correctness of domain reduction operators (Definition 3.1) that $\mathbf{x}\cap\bigcap_{C\in\mathcal{C}} C=\emptyset$. Thus, we have $\mathbf{x}\subseteq\neg\mathcal{C}$. □

3.2 Complementary Boxing Operators

We recall that in [Benhamou and Goualard 2000; Benhamou et al. 2004], a negation test takes as input a universally quantified constraint of the form $\forall t \in D_t : f(x,t) \leq 0$ and performs a kind of domain reduction operator on its negation, $f(x,t) > 0$. It encloses all possibly infeasible regions and the remaining is feasible.

The negation test has been extended to solve classic numerical constraints by Silaghi et al. [2001], in which a negation test is applied to inequalities of the form $f(x) \leq 0$. The search algorithm in [Silaghi et al. 2001], called **UCA6**, employs the negation test to enclose the negations of all individual constraints and then chooses the best result to guide the domain splitting during search. A concise description of the **UCA6** algorithm is presented in Section 4.3.

For convenience, we define a kind of operator to generalize the idea of a negation test, and then give several interesting properties. The generalized operator is called the *complementary boxing operator*, and the corresponding splitting operator is called the *box splitting operator*.

Definition 3.4 *Complementary Boxing Operator,* CB. Given a sequence X of n real variables associated with domains $\mathcal{D}$. A *complementary boxing operator* is a function CB that takes as input a box $\mathbf{x} \in \mathbb{I}_\circ^n$ contained in $\mathcal{D}$ and a finite set $\mathcal{C}$ of constraints on X, and that returns a box in $\mathbb{I}_\circ^n$, denoted by $\mathtt{CB}(\mathbf{x},\mathcal{C})$, satisfying the following properties:

$$\text{(Contractiveness)} \qquad \mathtt{CB}(\mathbf{x},\mathcal{C}) \subseteq \mathbf{x}, \tag{12}$$

$$\text{(Complementariness)} \qquad \mathbf{x} \setminus \mathtt{CB}(\mathbf{x},\mathcal{C}) \subseteq \bigcap_{C\in\mathcal{C}} C. \tag{13}$$

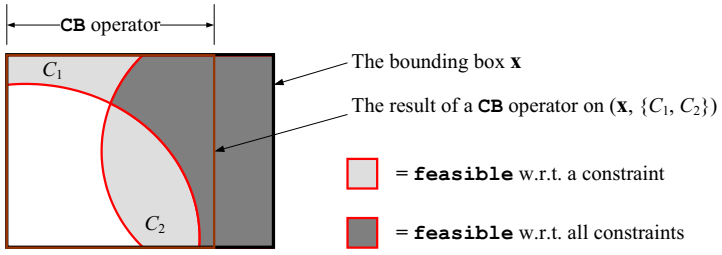

Fig. 4. An example of a complementary boxing (CB) operator applied to a box $\mathbf{x}$ and a set $\{C_1, C_2\}$ of two constraints.

A box resulting from the application of a complementary boxing operator to a bounding box $\mathbf{x}$ and a set $\mathcal{C}$ of constraints is called a *complementary box* of $\mathcal{C}$ within $\mathbf{x}$. The term *complementary boxing* refers to the process of computing a complementary box. The concept of a complementary boxing operator is depicted in Figure 4. Additionally, Figure 5 illustrates the outcomes of domain reduction operators and complementary boxing operators when applied to the same bounding boxes, in some typical situations.

The complementariness of complementary boxing operators means that the complementary boxing allows isolating certain regions, namely $\mathbf{x} \setminus \mathtt{CB}(\mathbf{x},\mathcal{C})$, of which the points entirely satisfy all the constraints in $\mathcal{C}$. Especially, if the application of a

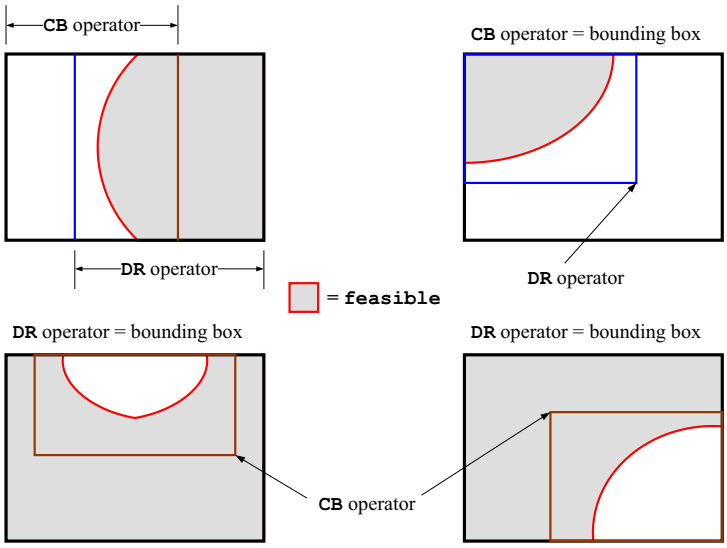

Fig. 5. Examples of domain reduction (DR) operators and complementary boxing (CB) operators.

complementary boxing operator to a box and a constraint results in an empty set, then the box completely satisfies that constraint. Similarly, if the application of a complementary boxing operator to a box with the whole set of constraints results in an empty set, then the box is completely feasible. The following theorem states this property formally.

THEOREM 3.5. *Given a set $\mathcal{C}$ of constraints on a sequence of n real variables associated with domains $\mathcal{D}$. Suppose $\mathbf{x}$ is a box contained in $\mathcal{D}$. If there exists a complementary boxing operator* CB *that maps $(\mathbf{x}, \mathcal{C})$ to an empty set (i.e., $\mathtt{CB}(\mathbf{x}, \mathcal{C}) = \emptyset$), then $\mathcal{C}$ is satisfied with every point in $\mathbf{x}$; that is, $\mathbf{x} \subseteq \bigcap_{C \in \mathcal{C}} C$.*

A complementary boxing operator can be constructed from a domain reduction operator as stated in the following theorem.

THEOREM 3.6. *Given a domain reduction operator* DR. *The function f defined as $f(\mathbf{x}, \mathcal{C}) \equiv \mathtt{DR}(\mathbf{x}, \neg\mathcal{C})$ is a complementary boxing operator.*

PROOF. By definition $\mathbf{x}_f = f(\mathbf{x}, \mathcal{C}) = \mathtt{DR}(\mathbf{x}, \neg\mathcal{C})$. The contractiveness of domain reduction operators implies that $\mathbf{x}_f \subseteq \mathbf{x}$. That is, f enjoys the contractiveness of complementary boxing operators. In addition to that, the correctness of domain reduction operators implies that

$$\mathbf{x} \cap \neg\mathcal{C} \subseteq \mathbf{x}_f \tag{14}$$

It follows from (14) that, for all $x \in \mathbf{x} \setminus \mathbf{x}_f$, we have $x \notin \mathbf{x} \cap \neg\mathcal{C}$; thus, $x \notin \neg\mathcal{C} \equiv \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C$, and $x \in \bigcap_{C \in \mathcal{C}} C$ because $x \in \mathbf{x} \subseteq \mathcal{D}$. That is, we have $\mathbf{x} \setminus \mathbf{x}_f \subseteq \bigcap_{C \in \mathcal{C}} C$. Thus, f enjoys the complementariness of complementary boxing operators. □

THEOREM 3.7. *Given a complementary boxing operator* CB. *The function f defined as $f(\mathbf{x}, \mathcal{C}) \equiv \mathtt{CB}(\mathbf{x}, \neg\mathcal{C})$ is a domain reduction operator.*

PROOF. By definition $\mathbf{x}_f = f(\mathbf{x}, \mathcal{C}) = \mathtt{CB}(\mathbf{x}, \neg\mathcal{C})$. The contractiveness of complementary boxing operators implies that $\mathbf{x}_f \subseteq \mathbf{x}$; that is, f enjoys the contractiveness of domain reduction operators. In addition to that, the complementariness of complementary boxing operators implies that

$$\mathbf{x} \setminus \mathbf{x}_f \subseteq \neg\mathcal{C} = \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C \tag{15}$$

It follows from (15) that, for all $x \in \mathbf{x}_f$, we have $x \notin \mathbf{x} \cap \neg\mathcal{C}$; thus, $x \notin \mathcal{D} \setminus \bigcap_{C\in\mathcal{C}} C$ and $x \in \bigcap_{C\in\mathcal{C}} C$ because $x \in \mathbf{x}_f \subseteq \mathcal{D}$. That is, we have $\mathbf{x} \setminus \mathbf{x}_f \subseteq \mathcal{C}$. This means that f enjoys the complementariness of complementary boxing operators. □

It follows from Theorem 3.6 and Theorem 3.7 that complementary boxing operators can be constructed from domain reduction operators and vice versa. In other words, they are dual to each other. In particular, let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a set of k constraints. A complementary boxing operator can be constructed by $\mathtt{CB}(\mathbf{x}, \mathcal{C}) := \mathtt{DR}(\mathbf{x}, \neg\mathcal{C}) = \mathtt{DR}(\mathbf{x}, C_0)$, where C_0 is the disjunction of constraints $\neg C_1, \ldots, \neg C_k$. In a system that does not accept disjunctive constraints, we can relax them by taking the (interval) union of complementary boxes, as stated in following theorem.

THEOREM 3.8. *Consider a sequence X of n real variables associated with domains $\mathcal{D}$. Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a set of constraints on X and $\{\mathtt{DR}_1, \ldots, \mathtt{DR}_k\}$ a set of domain reduction operators for X. Suppose $\{C'_1, \ldots, C'_k\}$ is a set of constraints on X such that $\neg C_i \equiv \mathcal{D} \setminus C_i \subseteq C'_i$ for all $i = 1, \ldots, k$. Then the operator defined by the following rule is a complementary boxing operator:*

$$\forall \mathbf{x} \in \mathbb{I}^n_\circ, \mathbf{x} \subseteq \mathcal{D} : f(\mathbf{x}, \mathcal{C}) \equiv \square \bigcup_{i=1}^{k} \mathtt{DR}_i(\mathbf{x}, C'_i), \tag{16}$$

PROOF. The contractiveness of f is obvious because $\mathtt{DR}_i(\mathbf{x}, C'_k) \subseteq \mathbf{x}$. We now prove the complementariness. For every $x \in \mathbf{x} \setminus \bigcup_{i=1}^{k} \mathtt{DR}_i(\mathbf{x}, C'_i)$ and $i \in \{1, \ldots, k\}$:

$$\begin{aligned}
& x \notin \mathtt{DR}_i(\mathbf{x}, C'_i) \supseteq \mathbf{x} \cap C'_i \\
\Rightarrow\ & x \notin \mathbf{x} \cap C'_i \\
\Rightarrow\ & x \notin C'_i \supseteq \neg C_i \\
\Rightarrow\ & x \notin \mathcal{D} \setminus C_i \\
\Rightarrow\ & x \in C_i \qquad\qquad (\text{since } x \in \mathcal{D}).
\end{aligned}$$

Thus, $x \in \bigcap_{i=1}^{k} C_i$. Therefore, we have

$$\mathbf{x} \setminus f(\mathbf{x}, \mathcal{C}) \subseteq \mathbf{x} \setminus \bigcup_{i=1}^{k} \mathtt{DR}_i(\mathbf{x}, C'_i) \subseteq \bigcap_{i=1}^{k} C_i.$$

This is the complementariness as required. □

The negation $\neg C$ of a numerical constraint C of the form $f(x) \diamond 0$ (where $\diamond$ is either $\leq$, $<$, $\geq$, $>$, $=$, or $\neq$) is the constraint $f(x)\ \widetilde{\diamond}\ 0$ (where $\widetilde{\diamond}$ is either $>$, $\geq$, $<$, $\leq$, $\neq$, or $=$, respectively). In practice, some implementations of domain reduction operators only accept constraints that are defined with the relations $\leq$

and $\geq$, but not with the relations $<$ and $>$. For example, a constraint C_i of the form $C_i \equiv (f(x) \leq 0)$ has the negation of the form $\neg C_i \equiv (f(x) > 0)$, which is not accepted in some implementations. Fortunately, we can safely use $C'_i \equiv (f(x) \geq 0)$ in the complementary boxing operator defined by (16) because $\neg C_i \subseteq C'_i$ holds.

The monotonicity of complementary boxing operators is defined similarly to that of domain reduction operators (see Definition 3.2). The following theorem gives a way to construct a (monotonic) inclusion test.

THEOREM 3.9. *Let X be a sequence of n real variables, $\{\mathtt{DR}_1, \ldots, \mathtt{DR}_n\}$ a set of (respectively, monotonic) domain reduction operators and $\{\mathtt{CB}_1, \ldots, \mathtt{CB}_n\}$ a set of (respectively, monotonic) complementary boxing operators, where $\mathtt{DR}_k$ and $\mathtt{CB}_k$ $(k = 1, \ldots, n)$ are defined in k dimensions. Let τ be a function that takes as input a box $\mathbf{x} \in \mathbb{I}^n_\circ$ and a finite set $\mathcal{C}$ of constraints on a subsequence Y of size k of X, and that returns either* `feasible`*,* `infeasible`*, or* `unknown` *such that:*

$$\tau(\mathbf{x}, \mathcal{C}) = \mathtt{infeasible} \quad \Leftrightarrow \quad \mathtt{DR}_k(\mathbf{x}[Y], \mathcal{C}) = \emptyset; \tag{17}$$

$$\tau(\mathbf{x}, \mathcal{C}) = \mathtt{feasible} \quad \Leftrightarrow \quad \mathtt{CB}_k(\mathbf{x}[Y], \mathcal{C}) = \emptyset. \tag{18}$$

Then τ is a (respectively, monotonic) inclusion test. If we restrict the codomain of τ to either $\{\mathtt{infeasible}, \mathtt{unknown}\}$ *or* $\{\mathtt{feasible}, \mathtt{unknown}\}$*, then the result still holds if we use either* (17) *or* (18)*, respectively, to construct τ.*

PROOF. It follows directly from Definition 2.4, Definition 2.11, Definition 3.2, Theorem 3.3 and Theorem 3.5. □

3.3 Domain Splitting Operators

First, we recall the concept of a bisection, where an interval (i.e., a side) of a box is dichotomized into two parts.

Definition 3.10 *Dichotomous Splitting Operator,* `DS`. A *dichotomous splitting* (`DS`) operator is a function $\mathtt{DS} : \mathbb{I}^n_\circ \to 2^{\mathbb{I}^n_\circ}$ that takes as input a box in $\mathbb{I}^n_\circ$, and that returns two (disjoint) boxes in $\mathbb{I}^n_\circ$ resulting from splitting a side of the input box into two halves.

Example 3.11. Consider a box $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}^n_\circ$, where $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i[$. A dichotomous splitting operator `DS` splits $\mathbf{x}$ into two disjoint boxes: $\mathbf{x}' = (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}'_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^{\mathrm{T}}$ and $\mathbf{x}'' = (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}''_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^{\mathrm{T}}$, where $\mathbf{x}'_i = [\underline{x}_i, \mathrm{mid}(\mathbf{x})[$ and $\mathbf{x}''_i = [\mathrm{mid}(\mathbf{x}), \overline{x}_i[$. Note that $\mathbf{x}'_i \cap \mathbf{x}''_i = \emptyset$; thus, $\mathbf{x}' \cap \mathbf{x}'' = \emptyset$.

Second, we define the concept of a box splitting operator, which splits around a complementary box in order to isolate *feasible regions* w.r.t. a subset of constraints.

Definition 3.12 *Box Splitting Operator,* `BS`. A *box splitting* (`BS`) operator is a function $\mathtt{BS} : \mathbb{I}^n_\circ \times \mathbb{I}^n_\circ \to 2^{\mathbb{I}^n_\circ}$ which takes as input two boxes such that the former contains the latter, and which sequentially splits the outer box along some facets of the inner one. The outcome is a sequence of disjoint boxes.

In fact, the concept of a box splitting operator is a slight generalization of the splitting operator proposed in [Van Iwaarden 1996]. The original splitting operator gives a way to split a region surrounding a box, provided that this box contains at most one optimal solution to a considered optimization problem.

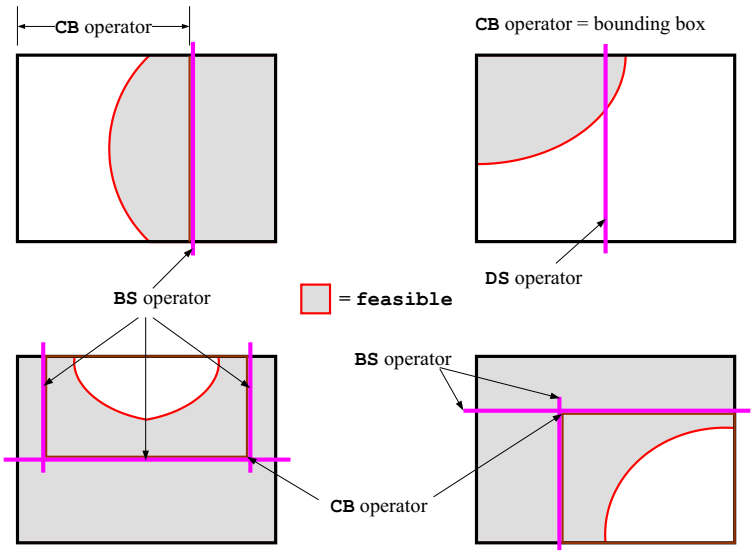

Fig. 6. Examples of box splitting (BS) and dichotomous splitting (DS) operators. In box splitting, all boxes excepted the complementary box are feasible w.r.t. the considered constraints.

In our algorithm, a box splitting operator that takes as input a domain box and a complementary box resulting from the application of a complementary boxing operator is applied in combination with a dichotomous splitting operator. The dichotomous splitting operator is used when either the complementary boxing operator produces no reduction or the box splitting operator results in too small boxes. Figure 6 illustrates the concept of a box splitting operator for this purpose.

4. BASIC BRANCH-AND-PRUNE SEARCH ALGORITHMS

In this section, we first present a generic search in the branch-and-prune framework and then present three search algorithms for NCSPs with emphasis on problems with continuums of solutions.

4.1 A Generic Branch-and-Prune Search Algorithm

We now present a generic search technique, called **BnPSearch**, for solving NCSPs in the branch-and-prune framework, the most common framework for the complete solution of NCSPs. The main steps of this generic search technique are described in Algorithm 1. Although this technique is not the most general one, it is still capable of generalizing the majority of the existing branch-and-prune techniques.

Taking a CSP as input, the **BnPSearch** algorithm produces two lists: $\mathcal{L}_\forall$ and $\mathcal{L}_\varepsilon$. The first list, $\mathcal{L}_\forall$, is an inner approximation of the solution set. The second list, $\mathcal{L}_\varepsilon$, consists of couples, each consists of a sequence of domains and a set of running constraints in these domains. When removing the running constraints, $\mathcal{L}_\varepsilon$ will become a boundary approximation of the solution set in association with the inner approximation $\mathcal{L}_\forall$. Each couple in $\mathcal{L}_\varepsilon$ constitues a CSP which will need to be explored further when reducing the value of the precision ε and continuing the

solving process. For simplicity, we may omit the running constraints in $\mathcal{L}_\varepsilon$. The pruning procedure **Prune** can be any combination of domain reduction techniques, which is not necessarily fixed for all steps of the algorithm.

Notation 4.1. The notations used in the algorithms in the rest of the paper follow the following conventions:

—The notations $\mathbf{B}$, $\mathbf{B}_i$ and $\mathbf{B}'$ denote relevant bounding boxes in $\mathbb{I}^n_\circ$; that is, the vectors of domains of the considered NCSPs.

—The notations $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}'$ and $\mathcal{C}''$ denote relevant sets of constraints.

—The notation $\mathbf{CB}_c$ denotes a complementary box w.r.t. a constraint, c.

—The notation $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ denote the global lists that accumulate computed boxes of inner and boundary union approximations, respectively.

—The notations DR, CB, and τ denote some domain reduction operator, complementary boxing operator, and inclusion test, respectively.

4.2 Classic Branch-and-Prune Search Algorithms

As mentioned in Section 2.3, most complete methods for solving NCSPs integrate domain reduction techniques, existence tests, uniqueness tests, exclusion tests, or inclusion tests into a *bisection* search strategy. The most successful solution methods enhance this process by applying domain reduction techniques such as *consistency* techniques to the constraint system after each split. This policy is referred to as *dichotomous maintaining bounds by consistency* (**DMBC**). A generic **DMBC** algorithm is presented in Algorithm 3, where ε is a positive precision (vector) and τ is a monotonic inclusion test (see Definition 2.11).

Algorithm 1: **BnPSearch** – a generic branch-and-prune search algorithm

Input: a CSP $\mathcal{P} \equiv (\mathcal{V}_0, \mathcal{D}_0, \mathcal{C}_0)$.
Output: $\mathcal{L}_\forall$, $\mathcal{L}_\varepsilon$. ◀ Lists of boxes in inner and boundary union approximations, respectively.
if **PruneCheck**($\mathcal{V}_0$, $\mathcal{D}_0$, $\mathcal{C}_0$, ε, τ, WAITLIST) **then return**; ◀ On page 19.
while WAITLIST $\neq \emptyset$ **do**
  Get a couple $(\mathcal{D}, \mathcal{C})$ from WAITLIST; ▼/* $\forall i \in \{1, \ldots, k\}$: $\mathcal{D}_i \subseteq \mathcal{D}$, $\mathcal{C}_i \subseteq \mathcal{C}$. */
  Split the CSP $(\mathcal{V}_0, \mathcal{D}, \mathcal{C})$ into sub-CSPs $\{(\mathcal{V}_0, \mathcal{D}_1, \mathcal{C}_1), \ldots, (\mathcal{V}_0, \mathcal{D}_k, \mathcal{C}_k)\}$;
  for $i := 1, \ldots, k$ **do** ◀ Do branching.
    if $\mathcal{C}_i = \emptyset$ **then**
      $\mathcal{L}_\forall := \mathcal{L}_\forall \cup \{\mathcal{D}_i\}$; ◀ All points in $\mathcal{D}_i$ are solutions.
      continue for;
    PruneCheck($\mathcal{V}_0$, $\mathcal{D}_i$, $\mathcal{C}_i$, ε, τ, WAITLIST); ◀ On page 19.

Function **PruneCheck**($\mathcal{V}$, $\mathcal{D}$, $\mathcal{C}$, ε, τ, WAITLIST)

$\mathcal{D} :=$ **Prune**$(\mathcal{V}, \mathcal{D}, \mathcal{C})$; ◀ Prune the domains in $\mathcal{D}$ by using domain reduction techniques.
if $\mathcal{D} = \emptyset$ **then return** `true`;
if all the domains in $\mathcal{D}$ are not amenable to be split (w.r.t. ε) **then**
  $\mathcal{L}_\varepsilon := \mathcal{L}_\varepsilon \cup \{(\mathcal{D}, \mathcal{C})\}$; **return** `true`;
put(WAITLIST $\leftarrow (\mathcal{D}, \mathcal{C})$);
return `false`; ◀ The problem has not been solved yet.

Algorithm 3: **DMBC** – an instance of the **BnPSearch** algorithm

Input: a bounding box $\mathbf{B}_0$, a constraint set $\mathcal{C}_0$, $\varepsilon \in \mathbb{R}^n_+$, a monotonic inclusion test τ.
Output: an inner union approximation $\boxplus^{\mathcal{I}}$, a boundary union approximation $\boxplus^{\mathcal{B}}$.
if **PruneCheckDMBC**($\mathbf{B}_0$, $\mathcal{C}_0$, ε, τ, WAITLIST) **then return**; ◀ Page 20.
while WAITLIST $\neq \emptyset$ **do**
  Get a box $\mathbf{B}$ from WAITLIST;
  $(\mathbf{B}_1, \mathbf{B}_2) :=$ Bisect($\mathbf{B}$); ◀ $\forall i \in \{1, 2\}$: $\mathbf{B}_i \subseteq \mathbf{B}$.
  PruneCheckDMBC($\mathbf{B}_1$, $\mathcal{C}_0$, ε, τ, WAITLIST); ◀ On page 20.
  PruneCheckDMBC($\mathbf{B}_2$, $\mathcal{C}_0$, ε, τ, WAITLIST); ◀ On page 20.

Function **PruneCheckDMBC**($\mathbf{B}$, $\mathcal{C}$, ε, τ, WAITLIST)

$\mathbf{B}' :=$ DR($\mathbf{B}$,$\mathcal{C}$); ◀ Reduce domains.
if $\mathbf{B}' = \emptyset$ **then return** `true`; ◀ $\mathbf{B}$ is infeasible, the problem has been solved.
if $\mathbf{B}$ is canonical or $\mathrm{w}(\mathbf{B}) \leq \varepsilon$ **then**
  CheckEpsilon($\mathbf{B}'$, $\mathcal{C}$, τ); ◀ On page 20.
  return `true`;
put(WAITLIST $\leftarrow \mathbf{B}'$); ◀ Put the current problem into the waiting list.
return `false`; ◀ The problem has not been solved yet.

Function **CheckEpsilon**($\mathbf{B}$, $\mathcal{C}$, τ)

if (RESULT $:= \tau(\mathbf{B},\mathcal{C})$) = `feasible` **then** ◀ Identify the feasibility of $\mathbf{B}$ w.r.t. $\mathcal{C}$.
  $\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}\}$; ◀ $\mathbf{B}$ is feasible, store it into the list of inner boxes.
else if RESULT = `unknown` **then**
  $\boxplus^{\mathcal{B}} := \boxplus^{\mathcal{B}} \cup \{\mathbf{B}\}$; ◀ $\mathbf{B}$ is undiscernible, store it into the list of boundary boxes.

In most of the known **DMBC** techniques, the search strategy is to perform splitting intervals until the intervals are canonical or their widths are not greater than a predetermined precision ε. That is, these techniques are able to achieve a predetermined precision ε (Definition 2.13). In general, the **DMBC** algorithm cannot detect feasible boxes, thus mainly addressing NCSPs with isolated solutions. When solving NCSPs with continuums of solutions, we can replace Function **PruneCheckDMBC** of the **DMBC** algorithm by Function **PruneCheckDMBC**$^+$ (on page 21). The obtained algorithm is thus called **DMBC**$^+$.

The difference (at Line 1) between Function **PruneCheckDMBC** and Function **PruneCheckDMBC**$^+$ is that the latter resorts to an inclusion test, τ' (not τ), to check if a box is feasible (in other words, it is an inner box). This can also be replaced with a complementary boxing operator, CB, checking if it returns an empty set. Hence, the **DMBC**$^+$ algorithm is able to detect feasible boxes, provided that the inclusion test τ' (or the complementary boxing operator CB) is sufficiently efficient. If the inclusion test τ' is implemented using an interval form of functions as in [Jaulin and Walter 1993; Jaulin et al. 2001], then the **DMBC**$^+$ will become the **SIVIA** algorithm[4] in [Jaulin and Walter 1993; Jaulin et al. 2001].

Because of the finite nature of floating-point numbers on computers, it is easy to

[4] **SIVIA** is the abbreviation of *set inverter via interval analysis*.

Function PruneCheckDMBC$^+$($\mathbf{B}$, $\mathcal{C}$, ε, τ, WAITLIST)

$\mathbf{B}' := \mathtt{DR}(\mathbf{B}, \mathcal{C})$; ◀ Reduce domains.
if $\mathbf{B}' = \emptyset$ **then return** `true`; ◀ $\mathbf{B}$ is infeasible, the problem has been solved.
if $\mathbf{B}$ is canonical **or** $\mathrm{w}(\mathbf{B}) \leq \varepsilon$ **then**
  CheckEpsilon($\mathbf{B}'$, $\mathcal{C}$, τ); ◀ On page 20.
  return `true`;
1 **if** $\tau'(\mathbf{B}', \mathcal{C}) =$ `feasible` **then** ◀ This can be optionally replaced with the check $\mathtt{CB}(\mathbf{B}', \mathcal{C}) = \emptyset$.
  $\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}'\}$; ◀ $\mathbf{B}$ is feasible, store it into the list of inner boxes.
  return `true`;
put(WAITLIST $\leftarrow \mathbf{B}'$); ◀ Put the current problem into the waiting list.
return `false`; ◀ The problem has not been solved yet.

prove that both the **DMBC** and **DMBC**$^+$ algorithms terminate (i.e., the waiting list WAITLIST becomes empty) after a finite number of steps and are of the precision ε w.r.t. the monotonic inclusion test τ (see Definition 2.13).

4.3 New Branch-and-Prune Search Algorithms

The **DMBC** and **DMBC**$^+$ algorithms often generate verbose inner and outer union approximations, especially when solving NCSPs with continuums of solutions. The first reason is that entirely feasible boxes may be unnecessarily split. The second reason is that all constraints are always considered in computations, even when some of them are completely satisfied by all points in considered domains.

Before presenting new search algorithms, we define the following terms.

Definition 4.2. Given a sequence X of n real variables $(x_1, \ldots, x_n)$, and a vector $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)^{\mathrm{T}} \in \mathbb{R}^n_+$. Consider a box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}^n_\circ$ and a finite set $\mathcal{C}$ of constraints on subsequences of X. A variable x_i is called an *active variable in* $\mathbf{x}$ *w.r.t.* $\mathcal{C}$ *and* ε if it occurs in at least one constraint in $\mathcal{C}$, $\mathrm{w}(\mathbf{x}_i) \equiv \sup(\mathbf{x}_i) - \inf(\mathbf{x}_i) > \varepsilon_i$ holds and $\mathbf{x}_i$ is not canonical. A variable x_i is called an *inactive variable in* $\mathbf{x}$ *w.r.t.* $\mathcal{C}$ *and* ε if it is not active in $\mathbf{x}$ w.r.t. $\mathcal{C}$ and ε.

Inspired by the idea of the **ICAb5** algorithm [Benhamou and Goualard 2000], we present in Algorithm 7 (on page 22) a simplified version, called **UCA5**, of the **UCA6** algorithm [Silaghi et al. 2001], where **UCA** stands for *union-constructing approximation*. The **UCA5** algorithm takes as input an NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{C}_0, \mathbf{B}_0)$ and returns an inner union approximation $\boxplus^{\mathcal{I}}$ and a boundary union approximation $\boxplus^{\mathcal{B}}$ of the solution set of $\mathcal{P}$. The outer union approximation of the solution set can be computed by $\boxplus^{\mathcal{O}} := \boxplus^{\mathcal{I}} \cup \boxplus^{\mathcal{B}}$. Roughly speaking, the **UCA5** algorithm proceeds by recursively repeating three main steps/processes:

(1) Use domain reduction (DR) operators to reduce the current bounding box, which plays the role of domains, to a narrower one (see Line 1 in Algorithm 7 and Line 5 in Function **PruneCheckUCA5**).

(2) Use complementary boxing (CB) operators to search for a complementary box w.r.t. some running constraint and the new bounding box obtained at Step 1 (Line 8 in Function **SplitUCA5**). During this search, the constraints that make empty complementary boxes are removed (Line 10 in Function **SplitUCA5**).

Algorithm 7: **UCA5** – a new branch-and-prune search algorithm

Input: a bounding box $\mathbf{B}_0$, a constraint set $\mathcal{C}_0$, $\varepsilon \in \mathbb{R}_+^n$, a monotonic inclusion test τ.
Output: an inner union approximation $\boxplus^{\mathcal{I}}$, a boundary union approximation $\boxplus^{\mathcal{B}}$.

$\boxplus^{\mathcal{I}} := \emptyset$; $\boxplus^{\mathcal{B}} := \emptyset$; WAITLIST $:= \emptyset$; ◀ The first two are global lists.

1 **if PruneCheckUCA5**($\mathbf{B}_0$, $\mathcal{C}_0$, ε, τ, WAITLIST) **then return**; ◀ On page 22.

while WAITLIST $\neq \emptyset$ **do**

2 $(\mathbf{B}, \mathcal{C}) :=$ getNext(WAITLIST); ◀ Get the next element from the waiting list.

3 $\mathcal{T} \equiv (\text{SPLITTER}, (\mathbf{B}_1, \ldots, \mathbf{B}_k), \mathcal{C}, c) :=$ **SplitUCA5**$(\mathbf{B}, \mathcal{C})$; ◀ On page 23.

if $\mathcal{T} = \emptyset$ **then continue while**; ◀ All points in $\mathbf{B}$ are solutions.

for $i := 1, \ldots, k$ **do** ◀ Do branching.

$\mathcal{C}_i := \mathcal{C}$;

if SPLITTER $=$ BS **and** $i > 1$ **then** ◀ $\mathbf{B}_i$ does not contain the complementary box.

$\mathcal{C}_i := \mathcal{C}_i \setminus \{c\}$; ◀ The constraint c is now redundant in $\mathbf{B}_i$ (Theorem 3.5).

if $\mathcal{C}_i = \emptyset$ **then** ◀ No running constraints.

$\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}_i\}$; ◀ $\mathbf{B}_i$ is an inner box.

continue for;

4 **PruneCheckUCA5**$(\mathbf{B}_i, \mathcal{C}_i, \varepsilon, \tau,$ WAITLIST$)$; ◀ On page 22.

Function PruneCheckUCA5($\mathbf{B}$, $\mathcal{C}$, ε, τ, WAITLIST)

5 $\mathbf{B}' :=$ DR$(\mathbf{B}, \mathcal{C})$; ◀ Prune the domains by using domain reduction operators.

if $\mathbf{B}' = \emptyset$ **then return** true; ◀ $\mathbf{B}$ is infeasible, the problem has been solved.

6 **if** there is no active variable in $\mathbf{B}'$ w.r.t. $\mathcal{C}$ and ε **then** ◀ Definition 4.2.

CheckEpsilon($\mathbf{B}'$, $\mathcal{C}$, τ); ◀ On page 20.

return true;

7 put(WAITLIST $\leftarrow (\mathbf{B}', \mathcal{C})$); ◀ Put the current problem into the waiting list.

return false; ◀ The problem has not been solved yet.

(3) Combine dichotomous splitting (DS) operators with box splitting (BS) operators to split the current problem into subproblems (see Line 3 in Algorithm 7 and Line 11 to Line 12 in Function **SplitUCA5**).

Remark 4.3. In practice, equality constraints usually define surfaces, we then do not need to perform the above Step 2 for these constraints.

The **UCA5** algorithm uses a waiting list, WAITLIST, to store the subproblems waiting to be processed further. The elements can be retrieved from, and be put to, WAITLIST by the functions getNext and put. WAITLIST can be handled as a queue or a stack. This allows for the breadth-first search in the former case and the depth-first search in the latter case.

In contrast to the **DMBC** and **DMBC**$^+$ algorithms, the **UCA5** algorithm restricts the DS operators at Line 12 in Function **SplitUCA5** to dichotomizing a domain of a variable only if this variable occurs in at least a running constraint. This avoids resulting in a huge number of tiny boxes. The reason is that, in the **UCA5** algorithm, constraints are removed from consideration whenever empty complementary boxes are computed w.r.t. the constraints (see Line 10 in Function **SplitUCA5**) and that, maybe, some variables no longer appear in any running constraints. For simplicity,

Function SplitUCA5($\mathbf{B}$, $\mathcal{C}$)

```
8  foreach c ∈ C do                ◀ Search for a constraint c for which complementary boxing results in a reduction.
9      CB_c := CB(B, c);           ◀ Enclose the negation of c by using complementary boxing operators.
10     if CB_c = ∅ then
           C := C \ {c};           ◀ c is redundant in B (Theorem 3.5).
           continue for;
       if CB_c ≠ B then break for; ◀ Thus, CB_c ⊂ B.
   if C = ∅ then                   ◀ No running constraints.
       ⊞^I := ⊞^I ∪ {B};           ◀ B is an inner box.
       return ∅;
11 if CB_c ≠ B then
       (B_1, ..., B_k) := BS(B, CB_c); SPLITTER := BS;   ◀ If BS did not fail, then B_1 ⊇ CB_c.
   if BS failed then SPLITTER := DS;
12 if SPLITTER = DS then (B_1, ..., B_k) := DS(B);       ◀ Bisect B, k = 2.
13 return (SPLITTER, (B_1, ..., B_k), C, c);
```

the interval (domain) with the greatest width is selected for DS operators.

For efficiency, the BS operators at Line 11 in Function **SplitUCA5** split along some facet of a complementary box only if this produces sufficiently large boxes; the complementary box itself is excepted. This estimation is done by using a predetermined parameter, fragmentation ratio. After the splitting phase (Line 13 in Function **SplitUCA5**), if the box splitting operator was chosen and successful (i.e., SPLITTER = BS), then the first resulting box $\mathbf{B}_1$ contains the complementary box $\mathbf{CB}_c$ and the constraint c is always satisfied in all the other boxes because of the complementariness of CB operators.

Function **PruneCheckUCA5** (on page 22) attempts to apply a DR operator to the input subproblem in order to reduce the domains of the subproblem. If this returns an empty box, the subproblem has no solutions. Afterwards, the procedure at Line 6 in Function **PruneCheckUCA5** is to check if the input subproblem has no active variable. If so, it uses a monotonic inclusion test, called τ, to check if the subproblem is either `infeasible`, `feasible`, or `unknown`. If τ returns `infeasible`, the subproblem is discarded. If τ returns `unknown`, the subproblem is classified as *undiscernible* w.r.t. ε and τ. In the other case, every point in the domains of the subproblem is a solution. Although the monotonic inclusion test τ in our implementation is a combination of DR and CB operators as described in Theorem 3.9, it is however not restricted to this kind of monotonic inclusion test.

In Algorithm 10, we present a slightly generalized and improved version of the **UCA6** algorithm – a search technique proposed by Silaghi et al. [2001]. Basically, this version is the same as the original version, but it is here improved by changing the order of the pruning steps (Function **PruneCheckUCA6**) such that each box is pruned before being put into the waiting list (WAITLIST). This change reduces the number of subproblems in the waiting list because some inconsistent subproblems can be discarded sooner than that in the original version in [Silaghi et al. 2001]. This version is general enough to be used with the heuristics in [Silaghi et al. 2001]. Those heuristics are represented by a generic function, called getSplitType, at Line 11 in Function **SplitUCA6**. Moreover, in this version, we make the stop condition more

Algorithm 10: **UCA6** – a new branch-and-prune search algorithm

Input: a bounding box $\mathbf{B}_0$, a constraint set $\mathcal{C}_0$, $\varepsilon \in \mathbb{R}_+^n$, a monotonic inclusion test τ.
Output: an inner union approximation $\boxplus^{\mathcal{I}}$, a boundary union approximation $\boxplus^{\mathcal{B}}$.

$\boxplus^{\mathcal{I}} := \emptyset$; $\boxplus^{\mathcal{B}} := \emptyset$; WAITLIST := $\emptyset$; ◀ The first two are global lists.

1 **if** **PruneCheckUCA6**($\mathbf{B}_0$, $\mathcal{C}_0$, $\{\mathbf{B}_0, \ldots, \mathbf{B}_0\}$, ε, τ, WAITLIST) **then return**; ◀ Page 24.

while WAITLIST $\neq \emptyset$ **do** ▼/* A set $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$ of memorized complementary boxes. */

2 $(\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}\}) :=$ getNext(WAITLIST);

3 $\mathcal{T} \equiv (\text{SPLTR}, (\mathbf{B}_1, \ldots, \mathbf{B}_k), \{\mathbf{CB}_c \mid c \in \mathcal{C}\}, b) :=$ **SplitUCA6**($\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}\}$);

if $\mathcal{T} = \emptyset$ **then continue while**; ◀ All points in $\mathbf{B}$ are solutions.

for $i := 1, \ldots, k$ **do** ◀ Do branching.

$\mathcal{C}_i := \mathcal{C}$;

if SPLTR = BS **and** $i > 1$ **then**

$\mathcal{C}_i := \mathcal{C}_i \setminus \{b\}$; ◀ The constraint b is now redundant in the box $\mathbf{B}_i$.

if $\mathcal{C}_i = \emptyset$ **then** ◀ No running constraints.

$\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}_i\}$; ◀ $\mathbf{B}_i$ is an inner box.

continue for;

4 **PruneCheckUCA6**($\mathbf{B}_i, \mathcal{C}_i, \{\mathbf{CB}_c \mid c \in \mathcal{C}_i\}, \varepsilon, \tau,$ WAITLIST); ◀ On page 24.

Function **PruneCheckUCA6**($\mathbf{B}$, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$, ε, τ, WAITLIST)

5 $\mathbf{B}' :=$ DR($\mathbf{B}, \mathcal{C}$); ◀ Reduce domains.

if $\mathbf{B}' = \emptyset$ **then return** true; ◀ $\mathbf{B}$ is infeasible, the problem has been solved.

6 **if** there is no active variable in $\mathbf{B}'$ w.r.t. $\mathcal{C}$ and ε **then** ◀ Definition 4.2.

CheckEpsilon($\mathbf{B}'$, $\mathcal{C}$, τ); ◀ On page 20.

return true;

7 put(WAITLIST $\leftarrow (\mathbf{B}', \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}\})$); ◀ Put the current problem into the waiting list.

return false; ◀ The problem has not been solved yet.

explicit at Line 6 in Function **PruneCheckUCA6**. It is important (for gaining in performance) to emphasize that checking if a domain (i.e., an interval) is not wider than the predetermined precision ε is only performed on the variables that occur in some running constraint, because some constraints have become redundant. This detail has been omitted in both [Silaghi et al. 2001] and [Silaghi 2002, Section 5.2.3].

In fact, the **UCA5** algorithm is a simplification of the **UCA6** algorithm. Thus, these two algorithms are very much similar and only different at the followings:

—At Line 1 and Line 4: Function **PruneCheckUCA6** takes not only the same arguments as Function **PruneCheckUCA5** does but also a list of complementary boxes computed at the parent of the current search node. Each box in this list is a complementary box w.r.t. a running constraint. At the beginning (Line 1), this list consists of boxes that equal to the initial bounding box.

—At Line 2 and Line 7: Each element in the waiting list WAITLIST of **UCA6** contains not only a domain box and a set of running constraints but also a list of complementary boxes computed at the parent of the current search node.

—At Line 3: Function **SplitUCA6** takes not only a box playing the role of domains and a set of running constraints but also a list of complementary boxes gotten from the waiting list. This function returns not only the used splitting type and

Function SplitUCA6($\mathbf{B}$, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$)

foreach $c \in \mathcal{C}$ **do**
  $\mathbf{CB}_c := \mathtt{CB}(\mathbf{B} \cap \mathbf{CB}_c, c)$;
  if $\mathbf{CB}_c = \emptyset$ **then** $\mathcal{C} := \mathcal{C} \setminus \{c\}$; ◀ c is redundant in $\mathbf{B}$ (Theorem 3.5).
if $\mathcal{C} = \emptyset$ **then** ◀ No running constraints.
  $\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}\}$; ◀ $\mathbf{B}$ is an inner box.
  return $\emptyset$;
SPLITTER := getSplitType(); ◀ Get a splitting mode, heuristics can be used.
if SPLITTER = BS **then** ◀ The splitting mode is box splitting.
  $\mathbf{CB}_b := \mathsf{chooseTheBest}(\{\mathbf{CB}_c \mid c \in \mathcal{C}\})$;
  $(\mathbf{B}_1, \ldots, \mathbf{B}_k) := \mathtt{BS}(\mathbf{B}, \mathbf{CB}_b)$; ◀ If box splitting did not fail, then $\mathbf{B}_1 \supseteq \mathbf{CB}_b$.
  if BS *failed* **then** SPLITTER := DS;
if SPLITTER = DS **then** $(\mathbf{B}_1, \ldots, \mathbf{B}_k) := \mathtt{DS}(\mathbf{B})$; ◀ Bisect $\mathbf{B}$, $k = 2$.
return (SPLITTER, $(\mathbf{B}_1, \ldots, \mathbf{B}_k)$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$, b);

the list of boxes as Function **SplitUCA5** does but also a *new* list of complementary boxes (because some constraints may have been removed).

—From Line 8 to Line 10: This is the main difference between the **UCA6** algorithm and the **UCA5** algorithm. While the **UCA5** algorithm only finds a complementary box that is strictly contained in the bounding box, the **UCA6** algorithm computes complementary boxes for every constraints and then chooses the best.

—From Line 11 to Line 13: Since the **UCA6** algorithm has just computed complementary boxes for every constraints, it chooses the smallest complementary box $\mathbf{CB}_b$ based on the volume which was computed for the constraint b. The constraint b will be used for box splitting operators. In the **UCA5** algorithm, the first-found complementary box which is strictly contained in the bounding box will be used for box splitting operators. Notice that the **UCA5** algorithm does not need an additional amount of memory to remember the computed complementary boxes in the waiting list WAITLIST.

Of course, both the **UCA5** and **UCA6** algorithms are instances of the **BnPSearch** algorithm. They compute inner and boundary union approximations, $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ respectively. These two union approximations are disjoint. Thus, we can obtain an outer union approximation by setting $\boxplus^{\mathcal{O}} := \boxplus^{\mathcal{I}} \cup \boxplus^{\mathcal{B}}$.

THEOREM 4.4. *Given a monotonic inclusion test τ and a positive precision (vector) ε. Both the* **UCA5** *and* **UCA6** *algorithms terminate and provide inner and boundary union approximations, $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ respectively, at the precision ε with respect to the monotonic inclusion test τ (see Definition 2.13).*

PROOF. No solution is lost because of the correctness of DR operators (Definition 3.1). Moreover, all points of any boxes in $\boxplus^{\mathcal{I}}$ are sound solutions. That is due to the complementariness of CB operators (Definition 3.4) and Theorem 3.5. Therefore, $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ are inner and boundary union approximations of the solution set, respectively.

If $\mathbf{B}$ is a box in $\boxplus^{\mathcal{B}}$, then

—The box $\mathbf{B}$ has no active variable w.r.t. ε and the running constraints $\mathcal{C}$ (at

Line 6 of Function **PruneCheckUCA5** and Function **PruneCheckUCA6**);

—Function **CheckEpsilon** (on page 20) returns `unknown` (i.e., $\tau(\mathbf{B}, \mathcal{C}) = \texttt{unknown}$).

If a variable x_i, of which the domain is $\mathbf{B}[x_i]$, is in $\mathsf{vars}(\mathcal{C})$ (the set of variables of $\mathcal{C}$), then the width $\mathrm{w}(\mathbf{B}[x_i])$ is not greater than ε_i. If a variable x_j is not in $\mathsf{vars}(\mathcal{C})$, we then split the interval $\mathbf{B}[x_j]$ of $\mathbf{B}$ into intervals whose widths are not greater than ε_j. Eventually, we obtain a number of ε-bounded boxes whose union is $\mathbf{B}$. It follows from the properties of a monotonic inclusion test (Definition 2.11) that each obtained ε-bounded box $\mathbf{B}'$ satisfies the property $\tau(\mathbf{B}', \mathcal{C}) = \tau(\mathbf{B}, \mathcal{C}) = \texttt{unknown}$, since the projections $\mathbf{B}'[\mathsf{vars}(\mathcal{C})] = \mathbf{B}[\mathsf{vars}(\mathcal{C})]$. □

Roughly speaking, the larger the union $\boxplus^{\mathcal{I}}$ is and the smaller the union $\boxplus^{\mathcal{B}}$ is, the more accurate the solution algorithm is.

5. COMPACTING THE REPRESENTATION OF SOLUTIONS

5.1 Controlling the Reduction of Small Domains

When using the **UCA5** algorithm or the **UCA6** algorithm to solve NCSPs with continuums of solutions, we observe that a better alignment of boxes near the boundary of the solution set can be obtained by finely controlling the application of domain reduction and complementary boxing operators. In particular, if a variable x_i of a subproblem is *inactive* (see Definition 4.2), then the domain reduction or complementary boxing should not be enforced on the variable x_i. This is to obtain better alignments of contiguous boxes and the computational performance.

A domain reduction operator (respectively, a complementary boxing operator) that only reduce the domains of active variables is called a *restricted-dimensional domain reduction operator* (respectively, a *restricted-dimensional complementary boxing operator*). We denote by $\texttt{DR}_{\texttt{rd}}$ (respectively, $\texttt{CB}_{\texttt{rd}}$) the restricted-dimensional domain reduction operator (respectively, the restricted-dimensional complementary boxing operator) obtained from the normal domain reduction operator `DR` (respectively, the normal complementary boxing operator `CB`) by enforcing the reduction only on active variables.

When local consistency techniques are enforced in order to obtain the effect of domain reduction (i.e., they play the role of domain reduction operators), restricted-dimensional operators amount to enforcing the local consistency techniques only for active variables. The recent algorithms for achieving box consistency, hull consistency and kB-consistency can be easily modified to adopt the above idea about restricted-dimensional operators, for example, by ignoring any procedure involving the inactive variables. In case a domain reduction or complementary boxing operator cannot be modified to adopt the idea of reducing only on active variables, we can apply it normally and then restore the domains of inactive variables to the initial domains. In this case, the gain is only in the (better) alignment of contiguous boxes, but not in the performance.

Fortunately, the implementation of box consistency in a well-known product called ILOG Solver [ILOG 2003] supports the above idea about reducing only on active variables. It can be done by simply passing only active variables (`X`) to the function `IloGenerateBounds` when we need to generate narrower bounds on `X`.

An illustration of the difference between the effect of a normal domain reduction

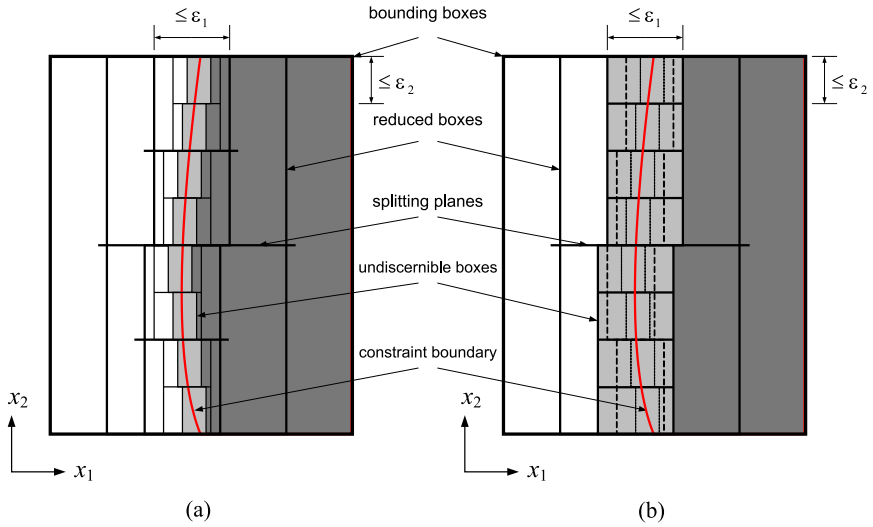

Fig. 7. An example of normal domain reductions and restricted-dimensional domain reductions at different levels: (a) all variables (x_1 and x_2) are considered for the over-reduction; (b) only the active variable (x_2) is considered for the reduction. The dark grey regions are inner boxes, the light grey regions are undiscernible boxes.

(DR) operator and the effect of a restricted-dimensional domain reduction ($\mathtt{DR}_{\mathtt{rd}}$) operator in the solving process is presented in Figure 7. In this example, although the normal DR operator produces more accurate output than the $\mathtt{DR}_{\mathtt{rd}}$ operator does, it has to spend much time on making the boundary region narrower than the allowed tolerance ε_1. In practice, this is unnecessary since real world applications mainly focus on the inner boxes, the boxes near the boundary are often unsafe for the further exploration in the applications, as shown in our experiments (see Section 6). Moreover, the number of boxes (15 inner boxes and 8 undiscernible boxes) resulting from the application of the DR operator is often higher than the number of boxes (3 inner boxes and 8 undiscernible boxes) resulting from the application of the $\mathtt{DR}_{\mathtt{rd}}$ operator. Moreover, the contiguous boxes obtained by using the $\mathtt{DR}_{\mathtt{rd}}$ operator are often aligned; hence, a geometrically compacting technique can work on them efficiently to get a concise representation of the solution set, as shown below.

5.2 Compact Representation of Solutions

Once the effect of better alignments is obtained, the question is how such a set of aligned boxes can be compacted into a smaller set. We propose to use the *extreme vertex representation* (EVR) of orthogonal polyhedra for this purpose. The extreme vertex representation was first proposed by Aguilera and Ayala [1997] for the three-dimensional space, and was later extended to the n-dimensional space in [Bournez et al. 1999; Bournez and Maler 2000]. The basic idea is that the union of disjoint boxes delivered by a box-covering solver defines an *orthogonal polyhedron* for which an improved representation can be generated. An orthogonal polyhedron can be naturally represented as the union of disjoint boxes (by enumerating the

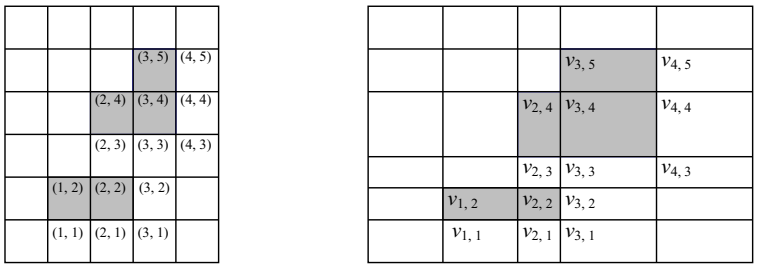

Fig. 8. Examples of a griddy polyhedron and an orthogonal polyhedron: (a) a griddy polyhedron made of the vertex indices of (b) an orthogonal polyhedron.

boxes and their vertices). That representation is called the *disjoint box representation* (DBR) in computational geometry. The EVR is a way of compacting the DBR (see [Aguilera 1998] and [Bournez et al. 1999; Bournez and Maler 2000]). Roughly speaking, in the EVR, the extreme vertices of an orthogonal polyhedron are identified and stored in a compact manner such that no information is lost w.r.t. the representation of the polyhedra. Moreover, when converting from EVR back to DBR, the obtained DBR are often more compact than the initial DBR.

We recall here basic concepts in the theory of extreme vertex representation. The reader can find more details in [Bournez and Maler 2000]. These concepts are sufficient to be presented for *griddy polyhedra*[5] because the results on general orthogonal polyhedra can be easily obtained from the results on griddy polyhedra by mapping the multidimensional array of vertex indices of the orthogonal polyhedra to the multidimensional array of vertices of griddy polyhedra (see Figure 8). In fact, they do not depend on the orthogonality of the underlying basis.

For simplicity, polyhedra are assumed to live in $\mathbf{X} = [0, m[^d \subseteq \mathbb{R}^d$ (the results also hold for $\mathbf{X} = \mathbb{R}^d_+$). Let $\mathcal{G} = (0, 1, \ldots, m-1)^d \subseteq \mathbb{N}^d$ be a grid of integer points. For every point $x \in \mathbf{X}$, $\lfloor x \rfloor$ denotes the grid point corresponding to the (componentwise) integer part of x. The unit box associated with a grid point $x = (x_1, \ldots, x_d)^{\mathrm{T}} \in \mathcal{G}$ is the box $\mathbf{B}(x) = [x_1, x_1 + 1[\times \cdots \times [x_d, x_d + 1[$. The set of all unit boxes is denoted by $\mathcal{B}$. A griddy polyhedron P is the set closure of the union of some unit boxes, or can be viewed as a set of grid points.

Definition 5.1 *Color Function.* Let P be a griddy polyhedron. The *color function* $\mathsf{color} : \mathbf{X} \to \{0, 1\}$ is defined as follows: if x is a grid point then $\mathsf{color}(x) = 1 \Leftrightarrow \mathbf{B}(x) \subseteq P$; otherwise, $\mathsf{color}(x) = \mathsf{color}(\lfloor x \rfloor)$.

We say that a grid point x is black (respectively, white) and that $\mathbf{B}(x)$ is full (respectively, empty) when $\mathsf{color}(x) = 1$ (respectively, $\mathsf{color}(x) = 0$). Figure 9a illustrates the color function for griddy polyhedra. Figure 9b illustrates the concept of a *forward cone* based at $x \in \mathcal{G}$, which is defined as $x^{\lhd} \equiv \{y \in \mathcal{G} \mid x \leq y\}$.

A *canonical representation* scheme for $2^{\mathcal{B}}$ (or $2^{\mathcal{G}}$) is a set $\mathcal{E}$ of syntactic objects

[5]A griddy polyhedron is the union of some unit hypercubes with integer-valued vertices (see [Bournez et al. 1999]).

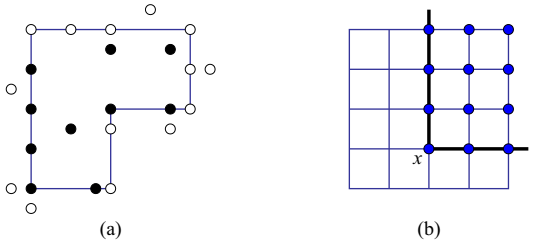

Fig. 9. A griddy polyhedron: (a) sample colors defined by the color function; (b) the forward cone $x^{\triangleleft} \equiv \{y \in \mathcal{G} \mid x \leq y\}$.

such that there is some bijective function $\Psi : \mathcal{E} \rightarrow 2^{\mathcal{B}}$; that is, every polyhedron has a unique representation. The most simple representation scheme is to explicitly enumerate the values of the color function on every grid point; hence, it needs a d-dimensional array of bits with m^d entries. Another simple representation is the vertex representation that consists of the set $\{(x, \mathsf{color}(x)) \mid x \text{ is a vertex}\}$. This is however still verbose. Hence, it is desired to store only important vertices only. The following definition identifies those important vertices.

Definition 5.2 *Extreme Vertex.* A grid point x is called an *extreme vertex* of a griddy polyhedron P if the number of black grid points in $\mathcal{N}(x) = \{x_1 - 1, x_1\} \times \cdots \times \{x_d - 1, x_d\}$ is odd ($\mathcal{N}(x)$ is called the *neighborhood* of x).

Let $\oplus$ denote the exclusive-or (XOR) operation: $p \oplus q = (p \wedge \neg q) \vee (p \wedge \neg q)$. The $\oplus$ operation on sets is defined by $A \oplus B = \{x \mid (x \in A) \oplus (x \in B)\}$. The set of extreme vertices (together with the $\oplus$ operation) makes a canonical representation of griddy polyhedra as follows [Bournez and Maler 2000, Theorem 1 & 2].

Theorem 5.3 Extreme Vertex Representation. *For any griddy polyhedron P, there is a unique set V of grid points in $\mathcal{G}$ such that $P = \bigoplus_{x \in V} x^{\triangleleft}$. Moreover, the set V is the set of all extreme vertices of P and this is a canonical representation.*

Figure 10 illustrates how the application of the concept of extreme vertex representations compacts union approximations. The eight light grey boxes of the boundary union approximation in Figure 10a can be concisely and equivalently represented by the two light grey boxes in Figure 10b. Theorem 5.3 shows that any griddy polyhedron can be canonically represented by the set of its extreme vertices (and their colors).

An effective transformation between DBR and EVR was proposed for low dimensional or small-size (i.e., m is small) polyhedra [Aguilera 1998; Bournez and Maler 2000]. For example, in the three-dimensional space, the average experimental time complexity of converting an EVR to a DBR is far less than quadratic but slightly greater than linear in the number of extreme vertices [Aguilera 1998]. Results in [Bournez and Maler 2000] also imply that, in a fixed dimension, the time complexity of converting a DBR to an EVR by using the XOR operator is linear in the number of boxes in DBR. We then propose to exploit the ideas of these effective transformation schemes to produce a compact representation of contiguous aligned boxes. This process works as follows:

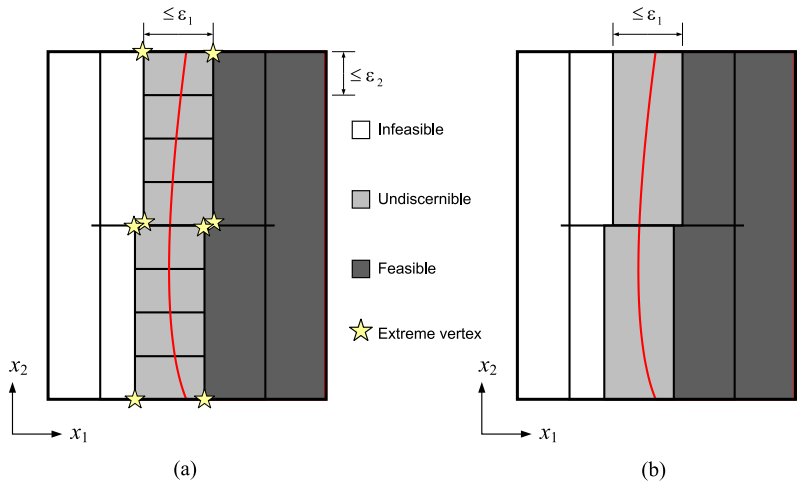

Fig. 10. The use of extreme vertex representations compacts the boundary union approximation in Figure 7b.

(1) Produce a better alignment of the boxes along the boundaries of constraints. This is done by preventing the unnecessary application of reduction operators over inactive variables. Figure 7 shows the effect of better alignment obtained for a set of nearly aligned boxes of an undiscernible approximation. The original set of eight small boxes (Figure 7a) reduces to two groups of four aligned boxes (Figure 7b) without altering the predetermined precision.

(2) The **Combination** function: The set of aligned boxes in each group, $\mathcal{S}_1$, is converted to EVR and then back to DBR to get a set of combined boxes, $\mathcal{S}_2$ (containing only one box in this case). Due to the properties of EVR, $\mathcal{S}_2$ is often more concise than $\mathcal{S}_1$. Figure 10 shows how this conversion procedure reduces eight boxes to two boxes.

The above conversion procedure can theoretically be applied in any dimension. Due to the efficiency of EVR in low dimension, we however restrict its application to very low dimensional small-size regions of the search space in our implementation (see Section 5.3). The running time for this conversion is hence near zero.

5.3 An Improved Branch-and-Prune Search Algorithm

We present in Algorithm 13 an improved version of the **UCA5** and **UCA6** algorithms, which is called **UCA6**$^+$. It also takes as input an NCSP, $\mathcal{P} \equiv (\mathcal{V}, \mathcal{C}_0, \mathbf{B}_0)$, and returns an inner union approximation $\boxplus^{\mathcal{I}}$ and a boundary union approximation $\boxplus^{\mathcal{B}}$ of the solution set of $\mathcal{P}$. Roughly speaking, the **UCA6**$^+$ algorithm uses restricted-dimensional versions of domain reduction and complementary boxing operators instead of normal versions in order to produce the effect of better alignment (and also to gain in performance), and then uses the conversion between the extreme vertex representation and disjoint box representation to get a compact representation of union approximations. The main processes of the **UCA6**$^+$ algorithm are similar to the three main processes of the the **UCA5** and **UCA6** al-

gorithm, but the **UCA6**$^+$ algorithm uses restricted-dimensional domain reduction ($\texttt{DR}_{\texttt{rd}}$) operators in place of domain reduction (DR) operators (see Line 6 in Function **PruneCheckUCA6**$^+$). The **UCA6**$^+$ algorithm also uses restricted-dimensional complementary boxing ($\texttt{CB}_{\texttt{rd}}$) operators in place of complementary boxing (CB) operators (see Line 13 in Function **SplitUCA6**$^+$).

The **UCA6**$^+$ algorithm does not compute complementary boxes for all running constraints as the **UCA6** algorithm does. Instead, it allows users to predefine a policy to choose a subset $\mathcal{C}''$ of $\mathcal{C}$, of which the constraints are enforced with $\texttt{CB}_{\texttt{rd}}$ operators (see Line 12 in Function **SplitUCA6**$^+$). A simple policy is to choose either all the constraints of $\mathcal{C}$ or a fixed number of constraints in $\mathcal{C}$. A more

Algorithm 13: **UCA6**$^+$ – a new branch-and-prune search algorithm

Input: a box $\mathbf{B}_0$, a constraint set $\mathcal{C}_0$, $\varepsilon \in \mathbb{R}^n_+$, a monotonic inclusion test τ, D_{stop}.
Output: an inner union approximation $\boxplus^{\mathcal{I}}$, a boundary union approximation $\boxplus^{\mathcal{B}}$.

$\boxplus^{\mathcal{I}} := \emptyset$; $\boxplus^{\mathcal{B}} := \emptyset$; WAITLIST := $\emptyset$; ◀ The first two are global lists.

1 **if PruneCheckUCA6**$^+(\mathbf{B}_0, \mathcal{C}'_0, \emptyset, \varepsilon, \tau,$ WAITLIST, $D_{\text{stop}})$ **then return**; ◀ On page 31.

while WAITLIST $\neq \emptyset$ **do**

▼/* A set $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$ of complementary boxes that were *optionally* memorized. */

2 $(\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}'\}) :=$ getNext(WAITLIST); ◀ $\mathcal{C}' \subseteq \mathcal{C}$, $\mathbf{CB}_c \subset \mathbf{B}$.

3 $(\text{SPLTR}, (\mathbf{B}_1, \ldots, \mathbf{B}_k), \{\mathbf{CB}_c \mid c \in \mathcal{C}'\}, \mathcal{C}, b) :=$ **SplitUCA6**$^+(\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}'\})$;

if SplitUCA6$^+$ returns $\emptyset$ **then continue while**;

for $i := 1, \ldots, k$ **do** ◀ Do branching.

$\mathcal{C}_i := \mathcal{C}$; $\mathcal{C}'_i := \mathcal{C}'$;

if SPLTR = BS **and** $i > 1$ **then**

$\mathcal{C}_i := \mathcal{C}_i \setminus \{b\}$; $\mathcal{C}'_i := \mathcal{C}'_i \setminus \{b\}$; ◀ b is now redundant in the box $\mathbf{B}_i$ (Theorem 3.5).

if $\mathcal{C}_i = \emptyset$ **then** ◀ No running constraints.

$\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}_i\}$; ◀ $\mathbf{B}_i$ is an inner box.

continue for;

4 **foreach** $c \in \mathcal{C}'_i$ **do if** $\mathbf{B}_i \subseteq \mathbf{CB}_c$ **then** $\mathcal{C}'_i := \mathcal{C}'_i \setminus \{c\}$;

5 **PruneCheckUCA6**$^+(\mathbf{B}_i, \mathcal{C}_i, \{\mathbf{B}_i \cap \mathbf{CB}_c \mid c \in \mathcal{C}'_i\}, \varepsilon, \tau,$ WAITLIST, $D_{\text{stop}})$;

Function PruneCheckUCA6$^+(\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}'\}, \varepsilon, \tau,$ WAITLIST, $D_{\text{stop}})$

6 $\mathbf{B}' := \texttt{DR}_{\texttt{rd}}(\mathbf{B}, \mathcal{C})$; ◀ Restricted-dimensional domain reduction.

if $\mathbf{B}' = \emptyset$ **then return** true; ◀ $\mathbf{B}$ is infeasible, the problem has been solved.

7 **if** there is no active variable in $\mathbf{B}'$ w.r.t. $\mathcal{C}$ and ε **then** ◀ Definition 4.2.

CheckEpsilon$(\mathbf{B}', \mathcal{C}, \tau)$; ◀ On page 20.

return true;

8 **if** there are at most D_{stop} active variables in $\mathbf{B}'$ **then** ▼/* Resort to another technique. */

9 $(\boxplus^{\mathcal{I}}(\mathbf{B}', \mathcal{C}), \boxplus^{\mathcal{B}}(\mathbf{B}', \mathcal{C})) :=$ **DimStopSolver**$(\mathbf{B}', \mathcal{C}, \varepsilon, \tau, \texttt{DR}_{\texttt{rd}}, \texttt{CB}_{\texttt{rd}})$;

▼/* **Combination**(.) does the conversions DBR → EVR → DBR in a D_{stop}-dimensional space. */

$\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup$ **Combination**$(\boxplus^{\mathcal{I}}(\mathbf{B}', \mathcal{C}))$; ◀ Store in the global list of feasible boxes.

$\boxplus^{\mathcal{B}} := \boxplus^{\mathcal{B}} \cup$ **Combination**$(\boxplus^{\mathcal{B}}(\mathbf{B}', \mathcal{C}))$; ◀ Store in the global list of undiscernible boxes.

return true; ◀ The problem has been solved.

10 **foreach** $c \in \mathcal{C}'$ **do if** $\mathbf{B}' \subseteq \mathbf{CB}_c$ **then** $\mathcal{C}' := \mathcal{C}' \setminus \{c\}$;

11 put(WAITLIST $\leftarrow (\mathbf{B}', \mathcal{C}, \{\mathbf{B}' \cap \mathbf{CB}_c \mid c \in \mathcal{C}'\})$); ◀ Put the problem into the waiting list.

return false; ◀ The problem has not been solved yet.

Function SplitUCA6$^+$($\mathbf{B}$, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}'\}$)

12 Choose an arbitrary subset $\mathcal{C}'' \subseteq \mathcal{C}$; ◀ $\mathcal{C}''$ is a set of constraints to be used with the $\texttt{CB}_{\texttt{rd}}$ operator.
13 **foreach** $c \in \mathcal{C}' \cup \mathcal{C}''$ **do**
  if $c \in \mathcal{C}' \cap \mathcal{C}''$ **then**
14 $\mathbf{CB}_c := \texttt{CB}_{\texttt{rd}}(\mathbf{B} \cap \mathbf{CB}_c, c)$;
  else if $c \in \mathcal{C}''$ **then** ◀ $c \notin \mathcal{C}'$.
15 $\mathbf{CB}_c := \texttt{CB}_{\texttt{rd}}(\mathbf{B}, c)$; $\mathcal{C}' := \mathcal{C}' \cup \{c\}$;
  else ◀ $c \in \mathcal{C}'$, $c \notin \mathcal{C}''$.
16 $\mathbf{CB}_c := \mathbf{B} \cap \mathbf{CB}_c$;
  if $\mathbf{CB}_c = \emptyset$ **then** $\mathcal{C} := \mathcal{C} \setminus \{c\}$; ◀ c is now redundant in $\mathbf{B}$ (Theorem 3.5).
  if $\mathbf{CB}_c = \emptyset$ **or** $\mathbf{CB}_c = \mathbf{B}$ **then** $\mathcal{C}' := \mathcal{C}' \setminus \{c\}$;
17 **if** $\mathcal{C} = \emptyset$ **then** ◀ No running constraints.
  $\boxplus^{\mathcal{I}} := \boxplus^{\mathcal{I}} \cup \{\mathbf{B}\}$; ◀ $\mathbf{B}$ is an inner box.
  return $\emptyset$;
18 SPLITTER := getSplitType(); ◀ Get a splitting mode, heuristics can be used.
19 **if** SPLITTER = BS **then** ◀ The splitting mode is box splitting.
20 $\mathbf{CB}_b := \text{chooseTheBest}(\{\mathbf{CB}_c \mid c \in \mathcal{C}'\})$;
  $(\mathbf{B}_1, \ldots, \mathbf{B}_k) := \texttt{BS}(\mathbf{B}, \mathbf{CB}_b)$; ◀ If box splitting did not fail, then $\mathbf{B}_1 \supseteq \mathbf{CB}_b$.
  if $\mathcal{C}' = \emptyset$ **or** BS *failed* **then** SPLITTER := DS;
21 **if** SPLITTER = DS **then** $(\mathbf{B}_1, \ldots, \mathbf{B}_k) := \texttt{DS}(\mathbf{B})$; ◀ Bisect $\mathbf{B}$, $k = 2$.
22 **return** (SPLITTER, $(\mathbf{B}_1, \ldots, \mathbf{B}_k)$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}'\}$, $\mathcal{C}$, b);

complicated and dynamic policy based on the pruning efficiency can be used. The set of constraints to be considered in the computation of complementary boxes – which is done by using $\texttt{CB}_{\texttt{rd}}$ operators (at Line 14 and Line 15) or by intersecting with the memorized complementary boxes (at Line 16) – is thus the union $\mathcal{C}' \cup \mathcal{C}''$, where $\mathcal{C}'$ is the set of constraints associated with the memorized complementary boxes (see Line 2, 3, 5, 11 and 22 in Algorithm 13). Notice that the set $\mathcal{C}'$ is only a subset of $\mathcal{C}$, in general.

The **UCA6$^+$** algorithm uses the same functions getSplitType and chooseTheBest as the **UCA6** algorithm does (see Line 18 and Line 20 in Function **SplitUCA6$^+$**). The computed complementary boxes can be memorized for improving the complementary boxing of subproblems. However, the memorization should be made optional because it may make the computation slow. Unlike the **UCA6** algorithm, the **UCA6$^+$** algorithm only memorizes complementary boxes that do not contain the corresponding bounding box (see Line 4 in Algorithm 13 and Line 10 in Function **PruneCheckUCA6$^+$**).

Function **PruneCheckUCA6$^+$** (on page 31) attempts to apply a $\texttt{DR}_{\texttt{rd}}$ operator to the input subproblem in order to reduce the domains of this subproblem. If it cannot prove that this subproblem is infeasible, it then checks if the subproblem has at most D_{stop} active variables. If the answer is yes, it resorts to a secondary solution technique, called **DimStopSolver**, to solve the current subproblem, provided that **DimStopSolver** provides an output with good alignments. A good candidate for **DimStopSolver** is a search technique with the *uniform cell subdivision*[6] or the

[6] A *uniform cell subdivision* means splitting the domain box into equal ε-bounded boxes, called

uniform bisection on all variables [Sam-Haroud and Faltings 1996; Lottaz 2000]. Variants of the **DMBC**$^+$ or **UCA6** algorithms that use the restricted-dimensional operators can also be candidates.

Given an NCSP, **DimStopSolver** constructs inner and boundary union approximations of the solution set (see Line 9 in Function **PruneCheckUCA6**$^+$). These two union approximations are naturally represented in DBR (or a bounding-box tree). They are converted to EVR and then back to DBR in order to combine each group of contiguous aligned boxes into a bigger equivalent box. This conversion procedure is performed by the **Combination** function.

THEOREM 5.4. *Given a monotonic inclusion test τ and a positive precision (vector) ε. The* **UCA6**$^+$ *algorithm terminates and provides inner and boundary union approximations, $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ respectively, at the precision ε with respect to the monotonic inclusion test τ (see Definition 2.13).*

PROOF. By an argument similar to the proof of Theorem 4.4, we have the following properties:

(1) $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ are inner and boundary union approximations of the solution set, respectively; thus, $\boxplus^{\mathcal{I}} \cup \boxplus^{\mathcal{B}}$ is an outer union approximation of the solution set.

(2) If not applying the **Combination** function in Function **PruneCheckUCA6**$^+$, every box $\mathbf{B}$ in $\boxplus^{\mathcal{B}}$ can be split into ε-bounded boxes such that each resulting box $\mathbf{B}'$ satisfies the property: $\tau(\mathbf{B}', \mathcal{C}) = \texttt{unknown}$, where $\mathcal{C}$ is the set of running constraints in $\mathbf{B}$.

(3) Since the **Combination** function does not alter the union of boxes in $\boxplus^{\mathcal{B}}$, the union of boxes in $\boxplus^{\mathcal{B}}$ equals to the union of the above ε-bounded boxes.

This implies what we have to prove. □

Notice that all the presented algorithms (**DMBC**, **DMBC**$^+$, **UCA5**, **UCA6** and **UCA6**$^+$) are complete if the inclusion test τ is ε-strong for all sufficiently small $\varepsilon > 0$, since they are all of the precision ε w.r.t. τ.

6. EXPERIMENTS

Since all the above-presented search algorithms should work similarly and equally for NCSPs with isolated solutions, in this paper we will only present experiments on NCSPs with continuums of solutions. The set of benchmark problems includes 14 nonlinear problems: **P1**, **P2**, **P3**, **P4**, **FD**, **G12**, **H12**, **F22**, **L01**, **LE1**, **S06**, **S08**, **TD**, **WP**. Their descriptions are given in Appendix A. They are NCSPs with continuums of solutions that have been impartially chosen to show different cases corresponding to different properties of constraints and solution sets and that can be solved efficiently by at least one of the considered search algorithms.

For the purpose of evaluation, we have implemented five search algorithms (**DMBC**, **DMBC**$^+$, **UCA5**, **UCA6** and **UCA6**$^+$) with different options using the same data structures and the same domain reduction operators. Our experiments discarded **DMBC**, which is a point-wise approach, as a reasonable candidate for solving NCSPs with continuums of solutions because it usually produces a huge

cells. When this splitting is used, the solver only need to solve subproblems defined on each cell.

Table I. The running time results for the search algorithms. The first seven problems are three-dimensional while the last seven problems are two-dimensional.

Algorithm ▶ Splitting type ▶ Memorization ▶	ε	**DMBC$^+$** DS No	**UCA6** DS Yes	**UCA6$^+$** DS No	**UCA5** BS + DS No	**UCA6** BS + DS Yes	**UCA6$^+$** BS + DS No	Ratio $\frac{\text{DMBC}^+}{\text{UCA6}^+}$
P1	0.1	> 24h	24.94s	19.34s	172.74s	3.76s	1.03s	> 83883
P2	0.1	> 24h	187.48s	95.70s	6.47s	7.80s	0.79s	> 109367
P3	0.1	37724.72s	61.82s	25.40s	8.86s	14.23s	0.63s	59880
P4	0.1	> 24h	140.25s	92.89s	4.96s	4.51s	0.96s	> 90000
FD	0.1	505.77s	183.62s	101.29s	48.10s	59.13s	31.91s	15.8
G12	0.1	429.82s	172.52s	96.40s	32.72s	33.59s	22.23s	19.3
H12	0.1	2161.17s	889.45s	267.36s	280.81s	273.64s	99.81s	21.7
F22	0.01	5.14s	3.81s	3.98s	3.25s	3.50s	2.70s	1.9
L01	0.01	2073.86s	1082.33s	660.74s	49.30s	51.08s	7.03s	295.0
LE1	0.01	94.04s	39.79s	40.89s	34.35s	22.05s	7.32s	12.8
S06	0.01	58.58s	44.29s	44.84s	29.78s	29.10s	24.34s	2.4
S08	0.01	175.36s	89.25s	41.62s	10.05s	9.90s	5.72s	30.7
TD	0.01	9.82s	5.43s	6.64s	3.46s	3.82s	1.43s	6.9
WP	0.01	296.29s	85.82s	47.50s	26.20s	24.60s	17.21s	17.6

number of boxes, each is ε-bounded, in very long running time. The source codes of the above search algorithms can be found in the BCS 2.5.2 (*box covering solver*) module, which is downloadable at the official web site of the COCONUT project, `http://www.mat.univie.ac.at/coconut-environment/`.

Table II. The numbers of boxes in inner union approximations (on the left) and boundary union approximations (on the right).

Prob. ▼ ▼	ε	**DMBC$^+$** DS Memo = No		**UCA6** DS Memo = Yes		**UCA6$^+$** DS Memo = No		**UCA5** BS + DS Memo = No		**UCA6** BS + DS Memo = Yes		**UCA6$^+$** BS + DS Memo = No	
P1	0.1	>210000	>810000	10402	30601	10219	15716	63124	67824	4065	10854	785	1253
P2	0.1	>280000	>730000	21833	66223	15563	40027	8750	23920	8347	26643	523	1091
P3	0.1	106784	528757	8398	48080	5147	28038	10744	29812	11942	38502	369	932
P4	0.1	>150000	>860000	24230	62405	23901	31972	6643	13988	4979	13423	562	866
FD	0.1	16437	92681	16437	92681	15585	47990	51878	65536	26331	70218	10321	35134
G12	0.1	17440	85062	17440	85062	12426	50878	34470	59440	24524	60526	13404	34590
H12	0.1	27280	144296	27280	144296	18417	88212	75999	127436	55080	127124	29032	74656
F22	0.01	1398	3458	1398	3458	1058	2260	1672	2584	1450	2664	906	1600
L01	0.01	65705	106348	65705	106348	51838	67510	50031	67619	34296	67659	1857	2073
LE1	0.01	14298	19688	14298	19688	9202	15331	13387	13795	8154	21918	1572	1496
S06	0.01	12345	27756	12345	27756	8827	20154	23692	30439	11692	26008	9546	17486
S08	0.01	26722	41208	26722	41208	16836	32478	15852	27384	15717	26624	9287	11716
TD	0.01	2881	3936	2881	3936	1936	2915	2685	4844	3160	4403	565	1091
WP	0.01	22212	38956	22212	38956	14341	29924	24465	36433	17264	33622	11273	18041

In the above algorithms, the domain reduction operators (`DR` and $\texttt{DR}_{\texttt{rd}}$) have been implemented using the function `IloGenerateBounds`, which is a kind of domain reduction and a variant of box consistency, in a well-known commercial product, ILOG Solver 6.0 [ILOG 2003]. The complementary boxing operators (`CB` and $\texttt{CB}_{\texttt{rd}}$) have

Table III. The ratios of the volume of inner approximations to that of outer approximations.

Algorithm ▶		DMBC$^+$	UCA6	UCA6$^+$	UCA5	UCA6	UCA6$^+$
Splitting type ▶	ε	DS	DS	DS	BS + DS	BS + DS	BS + DS
Memorization ▶		No	Yes	No	No	Yes	No
P1	0.1	n/a	0.980	0.979	0.997	0.997	0.990
P2	0.1	n/a	0.972	0.967	0.996	0.996	0.985
P3	0.1	0.710	0.710	0.640	0.956	0.956	0.836
P4	0.1	n/a	0.949	0.948	0.997	0.997	0.974
FD	0.1	0.984	0.984	0.983	0.992	0.992	0.986
G12	0.1	0.874	0.874	0.856	0.924	0.922	0.900
H12	0.1	0.885	0.885	0.868	0.938	0.937	0.918
F22	0.01	0.968	0.968	0.960	0.977	0.978	0.970
L01	0.01	0.999	0.999	0.999	≈1	≈1	0.999
LE1	0.01	0.997	0.997	0.995	0.999	0.999	0.997
S06	0.01	≈1	≈1	≈1	≈1	≈1	≈1
S08	0.01	0.999	0.999	0.999	≈1	≈1	≈1
TD	0.01	0.996	0.996	0.995	0.998	0.999	0.995
WP	0.01	0.999	0.999	0.998	0.999	0.999	0.999

been implemented as in Theorem 3.6. The monotonic inclusion test τ has been constructed as in Theorem 3.9. The inclusion test τ' in Function **PruneCheckDMBC$^+$** (Page 21) has been implemented using a complementary operator, and thus returns either `feasible` or `unknown`. For simplicity, the experiments have been taken with fixed settings for the new algorithms: fragmentation ratio $= 0.25$, $D_{\text{stop}} = 1$. All components of the vector ε are assumed to be the same. The secondary search technique (**DimStopSolver**) in the **UCA6$^+$** algorithm has been implemented as a simple combination of a uniform cell subdivision and a monotonic inclusion test.

The empirical results are presented in Table I, Table II, and Table III. Table I shows the running time results of the algorithms (in seconds and hours). Table II shows the numbers of boxes in inner and boundary union approximations delivered by the algorithms. Table III shows the ratios of the total volume of inner union approximations to that of outer union approximations. The term 'Memorization' (Memo) indicates the memorization of complementary boxes for the next iteration. The terms DS and BS + DS in Table I, Table II, and Table III indicate the splitting policies used in the corresponding search algorithms:

—DS: always dichotomize the largest domain of the domain box.

—BS + DS: attempt to use a box splitting (BS) first; if failed, then proceed with DS.

Our experiments show that the new algorithms (**UCA5**, **UCA6** and **UCA6$^+$**) is better than the classic algorithm (**DMBC$^+$**) in all measures. The best gains of the new algorithms over the classic one are obtained in case the arities of constraints are less than the arity of the problem (e.g., four problems **P1**–**P4**). *This shows how important the reduction of arity of problems is.* In most cases, the **UCA5** and **UCA6** algorithms with the option BS + DS are quite equal in all measures. However, the choice of constraints for splitting in **UCA6** is far better than that in **UCA5** in the solution of **P1**. The **UCA6$^+$** algorithm with the option BS + DS is always better than the others in the running time and the number of boxes, even if it does not need the memorization of complementary boxes (thus, less memory is need).

The best gains of **UCA6**$^+$ over the others are obtained when constraint boundaries contain a large percentage of nearly axis-parallel regions (e.g., **P2** and **P3**).

The **UCA6**$^+$ algorithm with the option BS + DS is slightly less accurate than the **UCA5** and **UCA6** algorithms in the volume measure. However, this situation get better when reducing ε. Moreover, this is hardly a matter for real world applications, because no one could ever use all solutions when a very large percentage of sound solutions has been found and all the considered algorithms are of precision ε w.r.t. τ (see Definition 2.13). The **DMBC**$^+$ algorithm and the **UCA6** algorithm with the option DS produce similar outputs when all constraints in a problem have the same set of variables, as happened for all the problems except four problems, **P1**–**P4**. We observed that *the above gains of the new algorithms over the classic algorithm,* **DMBC**$^+$*, get better when reducing* ε*, especially for hard problems.*

Notice that the arities of constraints in all the above problems, except **P1**–**P4**, equal to the arities of the problems. In fact, only the experiments on **P1**–**P4** may show the full effectiveness of the new algorithms, including the reduction of the arity of problem during the solution (we recall that the time and space complexities of the algorithms are exponential in the arity of problem). The experiments on the other problems do not show the the same improvements as those on **P1**–**P4**. This reveals that the effect of the arity reduction during the solution in new algorithm is an important improvement. Other experiments also show a similar relation among the search algorithms using a variant of hull consistency in [Benhamou et al. 1999].

7. CONCLUSION

In this paper, we presented a uniform view on search strategies of branch-and-prune methods for solving NCSPs. In this view, we started with a generic branch-and-prune algorithm, **BnPSearch**, and then derived from it two classic algorithms, **DMBC** and **DMBC**$^+$, for the point-wise and set-covering approaches, respectively. As the main contribution of the paper, we proposed three new branch-and-prune search algorithms: **UCA5**, **UCA6** and **UCA6**$^+$. Presenting the new algorithms as instances of **BnPSearch** and extensions of **DMBC**$^+$ facilitates the comparison of them. In particular, we clearly presented the differences among the algorithms.

Our experiments show that the **UCA6**$^+$ algorithm (with the option BS + DS) is the most adaptive search, in time and compactness, among the search algorithms for NCSPs with continuums of solutions, while the **UCA5** and **UCA6** algorithms (with the option BS + DS) seem to be able to balance between speed and accuracy in most cases. They are all far better than the classic branch-and-bound search algorithms, **DMBC** and **DMBC**$^+$, especially when the arities of constraints are less than the arity of the problem. Moveover, the new algorithms often provide a large percentage of sound solutions (in the form of a collection or tree of inner boxes) when solving NCSPs with continuums of solutions.

In case the solution set consists of continuums but is highly disconnected, one may wish to cluster its union approximations to get grouping/clustering information on them. In this case, we propose to use the clustering techniques [Vu et al. 2004] to perform a post-processing on the approximations to generate such information.

We predict that the **UCA6**$^+$ algorithm will show more speed up and compactness if we use higher values, $D_{\text{stop}} = 2, 3$ and use the search technique in [Sam-Haroud

and Faltings 1996; Lottaz 2000] in place of **DimStopSolver**. A direction for further research is to explore different combinations of pruning techniques and other tests such as existence, uniqueness, exclusion, and inclusion tests to make new branch-and-prune search methods, especially for addressing different classes of problems. Comparisons with a broader range of search algorithms are also needed.

ACKNOWLEDGMENTS

Support for this research was partially provided by the European Commission and the Swiss Federal Education and Science Office (OFES) through the COCONUT project (IST-2000-26063). We would like to thank ILOG for the software licenses of ILOG Solver used in the COCONUT project, and Professor Arnold Neumaier at the University of Vienna (Austria) for fruitful discussions and very valuable input.

REFERENCES

Agarwal, P. K., de Berg, M. T., Gudmundsson, J. G., Hammar, M., and Haverkort, H. J. 2001. Box-Trees and R-Trees with Near-Optimal Query Time. In *Proceedings of the 17th ACM Symposium on Computational Geometry*. ACM Press, 124–133. 11

Aguilera, A. 1998. Orthogonal Polyhedra: Study and Application. Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona, Spain. 10, 28, 29

Aguilera, A. and Ayala, D. 1997. The Extreme Vertices Model for Orthogonal Polyhedra. Tech. Rep. LSI-97-6-R, LSI-Universitat Politècnica de Catalunya, Barcelona, Spain. 27

Alefeld, G. and Herzberger, J. 1983. *Introduction to Interval Computations*. Academic Press, New York, NY. 5

Apt, K. R. 2003. *Principles of Constraint Programming*. Cambridge University Press, Cambridge, UK. 4

Asarin, E., Bournez, O., Dang, T., Maler, O., and Pnueli, A. 2000. Effective Synthesis of Switching Controllers for Linear Systems. *Proceedings of the IEEE, Special Issue on "Hybrid Systems" 88,* 7 (July), 1011–1025. 2

Benhamou, F. and Goualard, F. 2000. Universally Quantified Interval Constraints. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*. 67–82. 8, 9, 11, 14, 21

Benhamou, F., Goualard, F., Granvilliers, L., and Puget, J.-F. 1999. Revising Hull and Box Consistency. In *Proceedings of the International Conference on Logic Programming (ICLP'99)*. Las Cruces, USA, 230–244. 36

Benhamou, F., Goualard, F., Languenou, E., and Christie, M. 2004. Interval Constraint Solving for Camera Control and Motion Planning. *ACM Transactions on Computational Logic (TOCL) 5,* 4 (Octobor). 8, 9, 11, 14

Benhamou, F., McAllester, D., and Van Hentenryck, P. 1994. CLP(Intervals) Revisited. In *Proceedings of the International Logic Programming Symposium*. 109–123. 13

Benhamou, F. and Older, W. J. 1992. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. Tech. Rep. BNR Technical Report, Bell Northern Research, Ontario, Canada. 8, 13

Benhamou, F. and Older, W. J. 1997. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 32–81. Extension of a technical report of Bell Northern Research, Canada, 1992. 8, 13

Bournez, O. and Maler, O. 2000. On the Representation of Timed Polyhedra. In *Proceedings of International Colloquium on Automata Languages and Programming (ICALP'2000)*. Vol. LNCS 1853. Springer, 793–807. 27, 28, 29

Bournez, O., Maler, O., and Pnueli, A. 1999. Orthogonal Polyhedra: Representation and Computation. In *Hybrid Systems: Computation and Control*. Vol. LNCS 1569. Springer, 46–60. 27, 28

Collavizza, H., Delobel, F., and Rueher, M. 1999. Extending Consistent Domains of Numeric CSP. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*. 8, 11

Garloff, J. and Graf, B. 1999. Solving Strict Polynomial Inequalities by Bernstein Expansion. In *Proceedings of Symbolic Methods in Control System Analysis and Design*. 339–352. 8

Goldberg, D. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys 23,* 1 (March), 5–48. 5

Hansen, E. R. and Walster, G. W. 2004. *Global Optimization Using Interval Analysis*, Second ed. Marcel Dekker. 5

ILOG. 2003. *ILOG Solver 6.0 Reference Manual*. 8, 26, 34

Jaulin, L. 1994. Solution Globale et Guarantie de Problèmes Ensemblistes: Application à l'Estimation Non Linéaire et à la Commande Robuste. Ph.D. thesis, Université Paris-Sud, Orsay, France. 8, 11

Jaulin, L., Kieffer, M., Didrit, O., and Walter, E. 2001. *Applied Interval Analysis*, First ed. Springer. 2, 5, 8, 20

Jaulin, L. and Walter, E. 1993. Set Inversion via Interval Analysis for Nonlinear Bounded-Error Estimation. *Automatica 29,* 4, 1053–1064. 20

Lee, E. and Mavroidis, C. 2002. Solving the Geometric Design Problem of Spatial 3R Robot Manipulators Using Polynomial Homotopy Continuation. *Journal of Mechanical Design, Transaction of the ASME 124,* 4, 652–661. 2

Lee, E. and Mavroidis, C. 2004. Geometric design of 3r robot manipulators for reaching four end-effector spatial poses. *International Journal of Robotics Research 23*, 247–254. 2

Lee, E., Mavroidis, C., and Merlet, J. P. 2002. Five Precision Points Synthesis of Spatial RRR Manipulators Using Interval Analysis. In *Proceedings of the 2002 ASME Mechanisms and Robotics Conference, 2002 ASME Design Technical Conferences*. Montreal, Canada, 1–10. 2

Lhomme, O. 1993. Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*. 232–238. 13

Lottaz, C. 2000. Collaborative Design using Solution Spaces. Ph.D. thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. 2, 4, 8, 9, 11, 33, 37

Moore, R. E. 1966. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ. 5

Moore, R. E. 1979. *Methods and Applications of Interval Analysis*. SIAM Studies in Applied Mathematics. Philadelphia. 5

Neumaier, A. 1990. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge. 5

Neumaier, A. and Merlet, J.-P. 2002. Constraint Satisfaction and Global Optimization in Robotics. `http://www.mat.univie.ac.at/~neum/ms/robslides.pdf`. 2

Sam-Haroud, D. 1995. Constraint Consistency Techniques for Continuous Domains. Ph.D. thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. 4

Sam-Haroud, D. and Faltings, B. 1996. Consistency Techniques for Continuous Constraints. *Constraints 1*, 85–118. 8, 9, 11, 33, 36

Schichl, H. 2003. Habilitation thesis, Faculty of Mathematics, University of Vienna, Autralia. 2

Silaghi, M.-C. 2002. Asynchronously Solving Distributed Problems with Privacy Requirements. Ph.D. thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. 24

Silaghi, M.-C., Sam-Haroud, D., and Faltings, B. 2001. Search Techniques for Non-linear CSPs with Inequalities. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*. 1, 14, 21, 23, 24

Snyder, J. M. 1992. *Generative Modeling for Computer Graphics and CAD*. Academic Press, Inc., London, UK. 2

Van Hentenryck, P. 1998. A Gentle Introduction to NUMERICA. *Artificial Intelligence 103,* 1-2, 209–235. 8

Van Iwaarden, R. J. 1996. An Improved Unconstrainted Global Optimization Algorithm. Ph.D. thesis, University of Colorado at Denver, USA. 17

Vu, X.-H. 2005. Rigorous Solution Techniques for Numerical Constraint Satisfaction Problems. Ph.D. thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. `http://liawww.epfl.ch/Publications/Archive/vxhthesis.pdf`. 2, 4, 6

Vu, X.-H., Sam-Haroud, D., and Faltings, B. 2004. Clustering for Disconnected Solution Sets of Numerical CSPs. In *Recent Advances in Constraints: International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003*. Vol. LNAI 3010. Springer-Verlag, Budapest, Hungary, 25–43. 36

Vu, X.-H., Sam-Haroud, D., and Silaghi, M.-C. 2002. Approximation Techniques for Non-linear Problems with Continuum of Solutions. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002)*. Vol. LNAI 2371. Springer-Verlag, Alberta, Canada, 224–241. 1

Vu, X.-H., Sam-Haroud, D., and Silaghi, M.-C. 2003. Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002*. Vol. LNCS 2861. Springer-Verlag, Valbonne-Sophia Antipolis, France, 194–210. 1

A. NUMERICAL BENCHMARKS

A.1 Problem **TD**

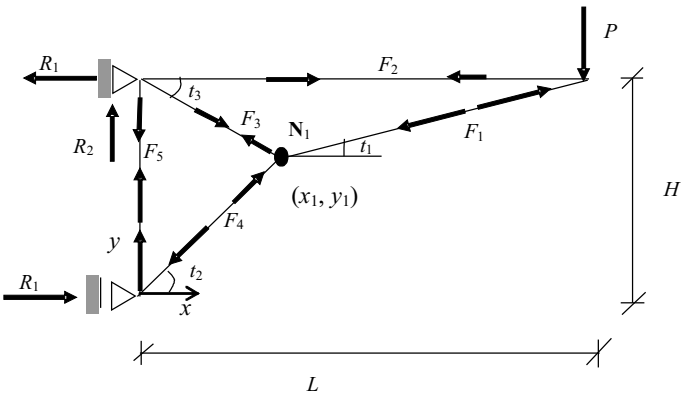

Fig. 11. The geometric design of a truss.

Consider the geometric design problem of a truss depicted in Figure 11. The goal is to find the coordinates of the moveable joint (x_1, y_1) in $[0.01, 10] \times [0.01, 10]$ of the node $\mathbf{N}_1$ of the truss such that all the following constraints are satisfied:

$$\begin{aligned}
F_2 &< TA, \\
F_5 &< TA, \\
F_1 &< C_1 A, \\
F_1 &< TA, \\
F_4 &< C_4 A, \\
F_4 &< TA, \\
|F_3| &< TA, \\
F_3 \leq 0 &\Rightarrow -F_3 < C_3 A, \\
x_1 &< L, \\
y_1 &< H,
\end{aligned}$$

where

E	$= 210 * 10^6$	Young's modulus of steel, unit = kN/m^2;
T	$= 235 * 10^3$	The yield stress of steel, unit = kN/m^2;
A	$= 0.25$	The area of cross section of truss members;
r	$= 0.5$	The radius of gyration of the cross section of truss members;
P	$= 400$	The loading capacity;
H	$= 6$	The height of truss;
L	$= 10$	The length of truss;

and the auxiliary variables are defined as follows:

$$\tan t_1 = (H - y_1)/(L - x_1),$$

$$
\begin{aligned}
\tan t_2 &= y_1/x_1,\\
\tan t_3 &= (H-y_1)/x_1,\\
R_1 &= PL/H,\\
F_1 &= P/\sin t_1,\\
F_2 &= P/\tan t_1,\\
F_3 &= (R_1-F_2)/\cos t_3,\\
F_4 &= R_1/\cos t_2,\\
F_5 &= R_1\tan t_2,\\
L_1 &= \sqrt{(L-x_1)^2+(H-y_1)^2},\\
L_3 &= \sqrt{x_1^2+(H-y_1)^2},\\
L_4 &= \sqrt{x_1^2+y_1^2},\\
C_1 &= \pi^2 E/(L_1/r)^2,\\
C_3 &= \pi^2 E/(L_3/r)^2,\\
C_4 &= \pi^2 E/(L_4/r)^2.
\end{aligned}
$$

In fact, this is a two-dimensional problem: the variables are x_1 and y_1. All the other variables can be easily eliminated in a preprocessing phase. The reduced constraints are however too complicated (in the number of elementary operations) to be read, and is hence not listed here.

A.2 Problem **FD**

Consider the design problem of the beam of a railway bridge under cyclic stress. The goal is to find $(L, q_f, Z) \in [10,30]\times[70,90]\times[0.1,10]$ such that the following yield stress and fatigue stress are satisfied:[7]

$$
\begin{aligned}
\sigma &< f_y,\\
\sigma_e &< \text{resistance},
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma_c &= 115000 && \text{Yield stress of steel, unit} = \mathrm{kN/m^2};\\
\gamma &= 1.1 && \text{The safety factor;}\\
f_y &= 460000 && \text{Unit} = \mathrm{kN/m^2};\\
\text{years} &= 200 && \text{The number of years to fatigue failure;}
\end{aligned}
$$

and the auxiliary variables are defined as follows:

$$
\alpha = \begin{cases}
1.3 & \text{if } L \le 4,\\
1.3-0.1(L-4) & \text{if } 4 < L \le 7.5,\\
0.95-0.008(L-7.5) & \text{if } 7.5 < L \le 20,\\
0.85-(L-20)/300 & \text{if } 20 < L \le 50,\\
0.75 & \text{if } L > 50,
\end{cases}
$$

$$
\phi = 0.82 + 1.44/(\sqrt{L}-0.2),
$$

[7] The variable Z is scaled up 100 times in unit in comparison to the original version.

$$\begin{aligned} q_r &= q_f\phi, \\ \sigma &= q_r L^2/8/(Z/100), \\ \sigma_e &= \alpha\sigma, \\ \text{cycles} &= 0.05 \text{ years}, \\ \sigma_r &= \sigma_c(\min\{2.5, \text{cycles}/2\})^{-1/3}, \\ \text{resistance} &= \sigma_r/\gamma, \end{aligned}$$

A.3 Problem **WP**

This is a two-dimensional simplification of the design model for a kinematic pair consisting of a wheel and a pawl. The constraints determine the regions where the pawl can touch the wheel without blocking its motion.

$$\begin{cases} 20 < \sqrt{x^2+y^2} < 50; \\ 12y/\sqrt{(x-12)^2+y^2} < 10; \\ x \in [-50, 50],\ y \in [0, 50]. \end{cases}$$

A.4 Problem **P1**

Three dimensions; the arities of constraints are less than the arity of problem:

$$\begin{cases} 2x^2 \le 3y - (y+1)^{0.2} + 5; \\ \ln(y^{3/2} + 2y + 1) + 5 \le z + (z+1/2)^{0.1}; \\ (x+1)^{1.5} \ge 2\sqrt{x}/(3+\sqrt{z^2+1}); \\ x \in [0, 50],\ y \in [0, 100],\ z \in [0, 50]. \end{cases}$$

A.5 Problem **P2**

Three dimensions; the arities of constraints are less than the arity of problem:

$$\begin{cases} x^2 \le y; \\ \ln y + 1 \ge z; \\ xz \le 1; \\ x \in [0, 15],\ y \in [1, 200],\ z \in [-10, 10]. \end{cases}$$

A.6 Problem **P3**

P2 added with the fourth constraint whose arity equals to the problem's arity:

$$\begin{cases} x^2 \le y; \\ \ln y + 1 \ge z; \\ xz \le 1; \\ x^{3/2} + \ln(1.5z + 1) \le y + 1; \\ x \in [0, 15],\ y \in [1, 200],\ z \in [0, 10]. \end{cases}$$

A.7 Problem **P4**

Three dimensions; the arities of constraints are less than the arity of problem:

$$\begin{cases} x^{1.5} + 1.9 \le \ln(y^3 + y + 1.5); \\ \ln(y^2 + z + 1) \le z + 2; \\ \sqrt{x^2 + z^2 + 12x + 5} \le 3 + (2x+3)^3; \\ x \in [0, 50],\ y \in [0, 100],\ z \in [0, 50]. \end{cases}$$

A.8 Problem **G12**

Three dimensions; the arities of constraints are equal to the arity of problem:

$$\begin{cases} x_1^2 + 0.5x_2 + 2(x_3 - 3) \geq 0; \\ x_1^2 + x_2^2 + x_3^2 \leq 25; \\ x_1, x_2, x_3 \in [-8, 8]. \end{cases}$$

A.9 Problem **H12**

Three dimensions; the arities of constraints are equal to the arity of problem:

$$\begin{cases} x_1^2 + x_2^2 + x_3^2 \leq 36; \\ (x_1 - 1)^2 + (x_2 - 2)^2 + x_3^2 \geq 16; \\ x_1^2 + (x_2 - 0.4)^2 \geq 2x_3; \\ x_1, x_2, x_3 \in [-10, 10]. \end{cases}$$

A.10 Problem **F22**

Two dimensions; the intersection of a tricuspoid and a circle:

$$\begin{cases} (x^2 + y^2 + 24x + 36)^2 \leq 64(x + 3)^3; \\ x^2 + y^2 \geq 8; \\ x, y \in [-4, 4]. \end{cases}$$

A.11 Problem **L01**

Two dimensions; a problem with logarithm and power operations:

$$\begin{cases} (x + 0.1)\sqrt{y} \geq 20 + \sqrt{x}; \\ \ln(\sqrt{y + 1} + 13) + 50 \geq (x + 0.5)^{1.2}; \\ x \in [0, 50],\ y \in [0, 200]. \end{cases}$$

A.12 Problem **LE1**

Two dimensions; a problem with logarithm, square root and exponent operations:

$$\begin{cases} e^{x+1}/e^{\sqrt{y+1}} \leq 100\sqrt{xy + 7} + 30; \\ (x^2 - 3x + 1)\sqrt{y + 2} \geq x \ln(10y + 3) + 50; \\ x, y \in [0, 50]. \end{cases}$$

A.13 Problem **S06**

Two dimensions; a single constraint whose solution set consists of disconnected subsets:

$$\begin{cases} 12y/\sqrt{(x - 12)^2 + y^2} \leq 10; \\ x \in [-50, 50],\ y \in [0, 50]. \end{cases}$$

A.14 Problem **S08**

Two dimensions; the difference between two circles with interior:

$$\begin{cases} 20 \leq \sqrt{x^2 + y^2} \leq 50; \\ x \in [-50, 50],\ y \in [0, 50]. \end{cases}$$