Ideas Investigación

Nuevas

▼ Uso de red transformer para generación de bloques en packing 3D

Colab

Idea: Entrenar red transformer para que dado un input (secuencia de cajas), sea capaz de dar las indicaciones para construir un bloque lo más grandes posibles que quepan en un contenedor de 100x100x100

Por ejemplo:

Input: (10,2,20),(8,7,45),(30,40,70)…

Output: x A (z (y A B) C) lo que quiere decir que la caja A la junto en la dimensión x, con la unión de las cajas AB (dim y) y C en la dimensión z

Los tokens necesarios serían 100+#cajas+#dim + SOS + EOS

TODO

- Generar data usando generador de bloques
- Generador de bloques a partir de output

▼ **Configuración parámetros BSG-CLP en preprocesamiento**

TODO

- [x] ~~ParamILS ($\alpha, \beta, \gamma, \delta$)~~
- [x] ~~Opción para que la salida sea la mejor parametrización encontrada, parametrizar segundos por run, rango de parámetros y cantidad de valores (archivo)~~
- [ ] Correr versión preprocesamiento con instancias. → **CORRIENDO**

Format of file

```
ordersize o1 o2 o3 o4
min1 max1 n
min2 max2 n
...
minN maxN n

#Example:
4 4 5 6 3
0 0 0
0 0 0
0 0 0
0.00 0.08 5
2.0 6.0 5
2.0 6.0 5
0.0 0.4 5
```

La idea consiste en configurar los parámetros del algoritmo (i.e., α, β, γ, p, min_{fr}, n_{bl}) utilizando greedies [aletarotios], previo a la resolución con BSG-CLP.

Los pasos del preprocesamiento serían:

1. Generación de bloques (20,000) con min_fr<0.98
2. Configurar parámetros usando algoritmo similar a param-ILS, es decir,
    a. seleccionar un parámetro
    b. probar distintos valores en el rango usando greedy o greedy aleatorio
    c. Aproximar resultados usando gaussiana, y quedarse con la media.

d. Repetir

La métrica de evaluación puede considerar la cantidad de pasos y bloques además del % de ocupación alcanzado. Ojo que el tiempo puede que sea dominado por la cantidad de evaluaciones de bloques, algo así: $O(steps * n_{bl})$

El greedy puede detenerse si la cantidad de pasos es grande (por ejemplo, más del doble que utilizando la estrategia de base).

▼ ChatGPT para revisión de controles con preguntas

▼ Making no-cycle matrices (OBBT)

▼ **TODO**

- Revisar concepto de ciclos, heurística propuesta elimina ciclos directos entre dos restricciones, pero hay otros ciclos.
- Cómo encontrar ciclos?
- Cuál es la mejor casilla para hacer cero?

▼ **Algoritmo que elimina ciclos entre 2 restricciones**

Formulación del problema

Problema: Dada una matriz A de dimensiones nxm (m > n) con elementos enteros, se desea transformar la matriz A en una matriz B sin ciclos realizando operaciones de fila y/o columna. Una matriz "sin ciclos" se define como una matriz en la que no existe ninguna submatriz 2x2 con 4 elementos distintos de cero.

Objetivo: Transformar la matriz A en una matriz B sin ciclos realizando operaciones filas/columna. Salida: secuencia de operaciones fila/columna.

Variables:

A: matriz de entrada de dimensiones nxm.
B: matriz resultante sin ciclos.

Restricciones:

Las dimensiones de la matriz B deben ser iguales a las dimensiones de la matriz A (nxm).
En cada submatriz 2x2 generada por 2 filas y 2 columnas en la matriz B, debe haber al menos 1 cero.
La matriz B se obtiene realizando un mínimo número de operaciones de fila y/o columna en la matriz A.

Algoritmo

El algoritmo tiene como objetivo encontrar una matriz sin ciclos. Para ello, recorre todas las submatrices de 2x2 de la matriz y cuenta el número total de ciclos. Luego, busca un elemento que forme parte de un ciclo y lo elimina mediante operaciones de fila. Este proceso se repite varias veces y se guarda la matriz sin ciclos que tiene el menor número total de ciclos.

En resumen, el algoritmo utiliza la eliminación de elementos que forman ciclos en la matriz para reducir gradualmente el número de ciclos hasta obtener una matriz sin ciclos.

Colab con acercamiento heurístico

$$\left.\begin{array}{ll} \min & \pm x_k \\ \text{s.t.} & \vec{g}_{\text{lin}}(\vec{x}) \leq 0 \\ & \vec{x}^l \leq \vec{x} \leq \vec{x}^u \\ & \vec{x} \in \mathbb{R}^n \end{array}\right\}$$

Precondicionar para obtener PA "pseudodiagonal" sin ciclos de la forma:

```
([[1, 1, 1, 0, 0, 0],
  [0, 0, 1, 1, 1, 0],
  [0, 0, 0, 0, 1, 1]])
```

PAx = Pb

En inecuaciones una matrix sin loops se puede ver así:

$$\begin{pmatrix} x & x & 0 \\ -x & -x & -x \\ 0 & x & -x \end{pmatrix}$$

En las columnas, los positivos se propagan entre ellos y los negativos también.

A. CPMP + RL (Lucas Agullo)

Paper CPMP_RL - overleaf

- Diseño de experimentos

B. Precondicionador óptimo aplicando inversa

El precondicionador se podría obtener de una linearización del sistema. Como n=m, se puede calcular la inversa directamente, lo que es mucho más económico que 2n-simplex.

Para sistemas con n>m, se podría **forzar un sistema cuadrado similar**. Por ejemplo, se pueden fusionar filas similares eliminando las variables de holgura $w_i \geq 0$:

$(x + y + w1 = 0; 2x + y - w2 = 0) \rightarrow 1.5x+y=0$
Cada fusión reduce m en 1 y n en 2.

Luego, si el sistema resultante es n>m, se pueden escoger m variables con mayor impacto para precondicionamiento.

Observación:
La mejora de cada bound depende de un conjunto de *restricciones activas*, estas restricciones pueden ser distinta para cada bound, por lo que hay que resolver 2n problemas.

Fusión de filas para reducir restricciones.
Descartar variables con menor impacto: x=mid(x), y=mid(y)..., para reducir variables

C. Kernel de sistemas m>n

A partir de un sistema lineal con m>n se pueden obtener restricciones del tipo:

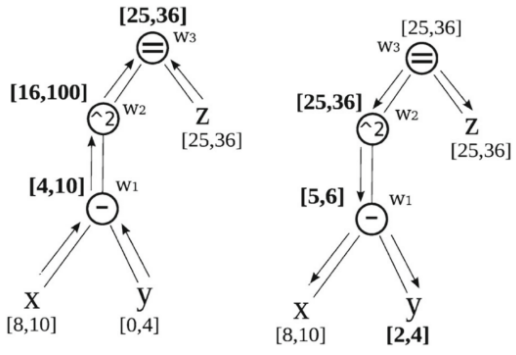
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = w_1 \begin{bmatrix} c_{11} \\ c_{12} \\ c_{13} \end{bmatrix} + w_2 \begin{bmatrix} c_{21} \\ c_{22} \\ c_{23} \end{bmatrix}$$

Las que agregan variables auxiliares w_j.

D. Eliminación de expresiones comunes (o extracción de subsistemas lineales)

When interval branch and bound solvers are used for solving numerical constraint satisfaction problems, constraint propagation algorithms are commonly used for filtering/contracting the variable domains. However, these algorithms suffer from the **locality problem** which is
related to the reduced scope of local consistencies.

For filtering, basic techniques like HC4 uses an **expression tree** related to the constraints of the system. Variables are replaced by its domains and a two-phase procedure is applied on each constraint. Something like this:



It is known that we can reduce the locality problem by constructing a directed acyclic graph (DAG) by **merging equivalent nodes** (or common subexpressions) and **identifying subsystems** of n-ary sums in the DAG. Contracting these subsystems and propagating the changes to the general system may improve the whole strategy.

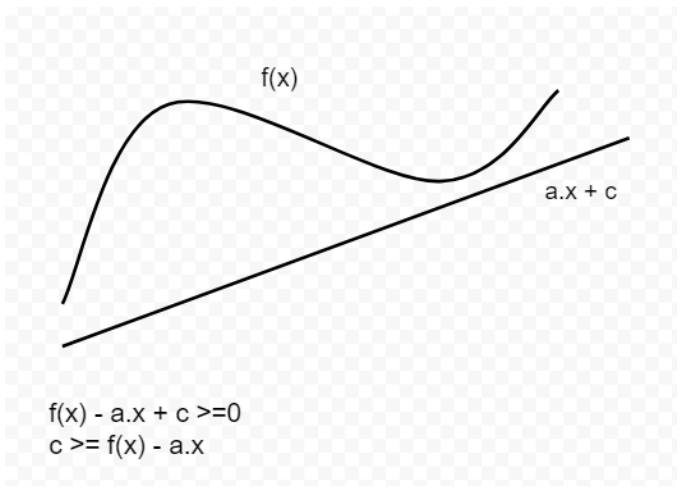
La idea de tu proyecto sería implementar un algoritmo en Python, que permita transformar un sistema de ecuaciones no lineal extrayendo los sistemas lineales relacionados con las expresiones comunes. En otras palabras, habría que reimplementar el algoritmo explicado en este paper para que pueda ser usado fuera del solver.

Pasos

1. Generar árbol de expresión a partir de ecuaciones en formato string
2. Aplicar algoritmo del paper para construir DAG
3. Transformar DAG en sistema de ecuaciones (formato string)

H. Linearization with image computation.

Para encontrar $fl(x) \leq f(x)$, con $fl(x) = a.x + c$.
Podemos obtener *a* derivando, e.g., $a = f'(x_m)$.
Luego, para obtener *c*, podemos calcular un lower bound para $f(x) - a.x$, ya que:
$c \leq f(x) - a.x$



I. Evaluación por mínimo/máximo

Encontrar intervalos mínimos y máximos para cada variable de la función. Usar $\frac{\partial f}{\partial x_i}(x) = 0$ en caso de que x_i no sea monótona.

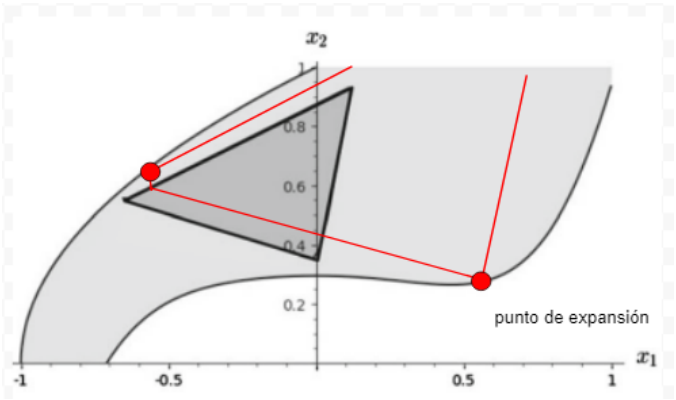
Evaluar en cajas mínimas y máximas.

Equivalente a evaluación por monotonía si la función es monótona.

f'(x) <= 0 -> hc4Revise -> x' = [0,2}
f'(x) >= 0 -> hc4Revise -> x'' = [1,3}
x' inter x'' = [1,2] -> mínimo

J. AbsTaylor2

Es posible mejorar abstaylor si seleccionamos con más cuidado los puntos de expansión?



TODO:

- Probar con ejemplos en Python. (Usar ejemplo del paper)

Online MD explorer

Explorador de soluciones online para ibexMop

LocalSearch+RL Greedy-CLP

Idea: Implementar A2C que aprenda a **seleccionar parámetros de función de evaluación VCS** para problema CLP.

Estado: +/- valores de parámetros + %utilización alcanzada por greedy

Acciones: +/- valores de parámetros

Recompensa: +/- %utilización alcanzada por greedy

Stack de estados para el agente.

LocalSearch+RL

Idea: Implementar DeepRL que aprenda optimizar función caja negra f(x) probando valores de x.

Estado: $x_t, f(x_t)$

Acciones: Δx_t

Recompensa: $f(x + \Delta x_t) - f(x_t)$

Entrenar en base a **stack de estados.**

Eliminación de expresiones comunes

Agentes HL y LL

Replanteamiento

J Glob Optim
DOI 10.1007/s10898-015-0390-4

Interval Branch-and-Bound algorithms for optimization and constraint satisfaction: a survey and prospects

Ignacio Araya[1] · **Victor Reyes**[2]

Received: 28 April 2014 / Accepted: 6 December 2015

Abstract Interval Branch and Bound algorithms are used to solve rigorously continuous constraint satisfaction and constrained global optimization problems. In this paper, we explain the basic principles behind interval Branch and Bound algorithms. We detail the main components and describe issues that should be considered to improve the efficiency of the algorithms.

Keywords Interval arithmetic · Constraint propagation · Numerical constrained optimization · Numerical constraint satisfaction · Interval-based solver · Branch and Bound

1 Introduction

Interval-based methods have gained considerable interest over the last decade. Two main features distinguish interval-based methods from other methods:

- They account for uncertainties in the parameter values and computation errors over the floating-point numbers.
- They are able to treat the complete search space, thus offering proofs of infeasibility and/or certification of solutions.

Because of these features, these methods are being used in several research areas, including robotics, localization, model qualification, and the design of control systems. In [53,68,81], robots and vehicles obtain useful information from a set of sensors (e.g., odometers, GPS or gyrocompass). Then, this information is transformed into a set of non-linear systems of

✉ Ignacio Araya
ignacio.araya@pucv.cl

Victor Reyes
vreyes@inf.utfsm.cl

[1] Escuela Informática, Pontificia Universidad Católica de Valparaíso, Avenida Brasil 2950, V Región, Chile

[2] Depto. Informática, Universidad Técnica Federico Santa María, Avenida España, 1680, V Región, Chile

Published online: 29 December 2015

constraints. Solving these systems means finding the position, displacement, or required movement for the vehicle/robot. Uncertainties of sensor measurement errors, mechanical parts or floating-point number computation errors are considered by interval-based methods. Interval-based techniques have also been used successfully for localizing nodes in mobile networks and tracking acoustical sources [64,65,91,104]. In designing robust control systems, the quantitative feedback theory (QFT) method has been studied by interval-based techniques in several works (e.g., [92,93,98]). QFT uses the feedback of measurable plant/machinery outputs to generate an acceptable response. Usually, the general problem in QFT is related to how to design the feedback controller and prefilter such that the desired specifications are satisfied in the region of uncertainty. In [98], the authors propose posing the prefilter problem as a constraint satisfaction problem following a set of rules. The problem is then solved using interval-based techniques. The approach is validated using a magnetic levitation system. In [99,100], the authors propose finding reachable sets using interval-based methods, i.e., finding all the trajectories on a hybrid system (involving discrete and continuous dynamics), starting from a possible initial set of states under disturbances and uncertainties in parameter values.

Interval Branch and Bound (B&B) strategies are primarily used to solve continuous constraint systems and to handle constrained global optimization problems. These strategies are *mathematically rigorous*, i.e., they account for the rounding errors that are implied by floating-point operations in their implementations. The story of interval B&B started with interval analysis [90]. Numerous interval-based techniques for numerical unconstrained optimization appear in Ratz's dissertation [102] and are included in the Pascal-XSC solver described in [45]. In [47], Hansen provides an informal description of many techniques and heuristics for use in unconstrained optimization algorithms. In [57], Kearfott proposes a solver containing techniques for the monotonicity test and a simple technique for computing upper bounds. In [101], Ratschek explains the fundamentals for handling inequality constraints, and in [32], Hansen discusses many interval techniques for both inequality and equality constraints. In the mid-1990s, Kearfott designed GlobSol [59], a solver for constrained global optimization problems. Researchers from the constraint programming community designed the constrained global optimization solvers Numerica [120], introducing interval constraint propagation algorithms, Icos [70,71] and IbexOpt [2,118], introducing safe linear relaxations. The optimizer IBBA [83–85,96], integrated constraint propagation and affine arithmetic.

Despite all of the advances made to date in interval-based methods, interval B&B algorithms generally remain less efficient than global optimizers (non-linear programming solvers) from the mathematical programming community, such as BARON [107], Couenne [9] and the recent ANTIGONE [89]. The latter optimizers, however, are non-rigorous, i.e., they cannot offer any guarantee and occasionally miss the best solution due to a lack of rigorous computations over the floating-point numbers. In a recent work [5], a set of rigorous and a set of non-rigorous global optimizers were compared (BARON, Couenne, IbexOpt, IBBA, Icos and GlobSol), considering a set of 74 instances. IbexOpt and BARON offered the best results. Although BARON and Couenne were shown to be better than IbexOpt in simple instances, IbexOpt outperformed Couenne and reached the performance of BARON in the most difficult ones.

In this paper, we explain the basic principles behind interval B&B algorithms and offer a review of their main components. Then, we present a set of issues to be considered to improve the performance of current interval B&B algorithms.

Section 2 provides basic notions about interval arithmetic. In Sect. 3, the constraint satisfaction and constrained global optimization problems are defined. Section 4 describes the general structure of interval-based solvers. Sections 5, 6, 7 and 8 detail the main components

of interval-based solvers, describing the most used methods and discussing related issues. Conclusions are given in Sect. 9.

2 Interval arithmetic: notation and basic definitions

An **interval** $\boldsymbol{x}_i = [\underline{x_i}, \overline{x_i}]$ is the set of real numbers between $\underline{x_i}$ and $\overline{x_i}$. $\underline{x_i}$ denotes the minimum of $\boldsymbol{x}_i$, and $\overline{x_i}$ denotes the maximum of $\boldsymbol{x}_i$. The width of $\boldsymbol{x}_i$ is $\text{wid}(\boldsymbol{x}_i) = \overline{x_i} - \underline{x_i}$. A **degenerated interval** is an interval with width 0. $\text{mid}(\boldsymbol{x}_i)$ denotes the midpoint of $\boldsymbol{x}_i$. A **box** $\boldsymbol{x}$ is the Cartesian product of intervals $\boldsymbol{x}_1 \times \cdots \times \boldsymbol{x}_n$. Depending on the case, $\boldsymbol{x}$ may also denote a vector of intervals $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. $\mathbb{IR}$ denotes the set of all of the intervals.

A function $f{:}\mathbb{R}^n \to \mathbb{R}$ is **factorable** if it can be computed in a finite number of simple steps, starting with model variables and real constants, using elementary unary and binary operators. Factorable functions are extended to intervals. $\boldsymbol{f}{:}\mathbb{IR}^n \to \mathbb{IR}$ is said to be an **extension** of the factorable function f to intervals. $\boldsymbol{f}(\boldsymbol{x})$ contains the image of f over $\boldsymbol{x}$, i.e., $\boldsymbol{f}(\boldsymbol{x}) \supset \{f(x), x \in \boldsymbol{x}\}$. The **optimal image** ($\boldsymbol{f}_{opt}$) is the sharpest interval containing $\{f(x), x \in \boldsymbol{x}\}$.

Interval arithmetic defines the extension of the classic elementary operators (e.g., $+$, $-$, $*$, $/$, $power$, exp, log, sin, cos,...). For instance, $[a, b] + [c, d] = [a + b, c + d]$, $[log]([a, b]) = [log(a), log(b)]$, etc. The **natural extension** $\boldsymbol{f}_N$ of a factorable function f corresponds to the mapping of f to intervals using the corresponding interval operators. For instance, consider the function $f(x_1, x_2, x_3) = x_1 + (x_2 \cdot x_3)^2 + 4$. The natural extension of f is given by:

$$\boldsymbol{f}_N(\boldsymbol{x}) = \boldsymbol{x}_1 + (\boldsymbol{x}_2 \cdot \boldsymbol{x}_3)^2 + 4$$

If we consider the following domains of variables, $\boldsymbol{x}_1 = [1, 2], \boldsymbol{x}_2 = [2, 4]$ and $\boldsymbol{x}_3 = [-1, 1]$, we can compute the image of the natural extension or **natural evaluation**; that is, $[1, 2] + ([2, 4] * [-1, 1])^2 + 4 = [5, 22]$. In this case, the image is optimal, i.e., $\boldsymbol{f}_N(\boldsymbol{x}) = \boldsymbol{f}_{opt}(\boldsymbol{x})$; however, this is not the general case.

A common way to compute the image of an interval function consists of representing the factorable function as an **expression Direct Acyclic Graph** (expression DAG). Variables and constraints are represented by leaves in the DAG, and the operators are represented by vertices or nodes. Thus, the function or expression corresponds to the root node (see Fig. 1). The computation of the image of $\boldsymbol{f}$ is performed by replacing the variables with their related intervals/domains and applying, recursively, interval arithmetic in each node.

The main reason that the computation of optimal image approximations is difficult is known as the **dependency problem**. The dependency problem occurs in expressions with multiple occurrences of variables. This problem explains, for instance, why the extension to intervals of the expression $x_1 - x_1$ cannot be computed optimally, i.e., $\boldsymbol{x}_1 - \boldsymbol{x}_1 \neq [0, 0]$. The reason is that the interval operators consider each occurrence of a variable as an independent occurrence. Thus, $\boldsymbol{x}_1 - \boldsymbol{x}_1$ is equivalent to $\boldsymbol{x}_1 - \boldsymbol{x}_2$ when the intervals related to x_1 and x_2 are equivalent. If f is continuous in a box $\boldsymbol{x}$ and has *no multiple occurrences of variables*, then the natural extension computes an optimal image approximation of f over $\boldsymbol{x}$.

Several extensions, focused on reducing the dependency problem, have been proposed thus far. Among the most well-known extensions, we find the monotonicity-based extension and the Taylor-based extension. The **monotonicity-based extension** (ME) first computes a lower and upper bound of the gradient of f to identify the *monotonic variables*. **Monotonic variables** are all of the variables x_i such that f is monotonically increasing (or decreasing)

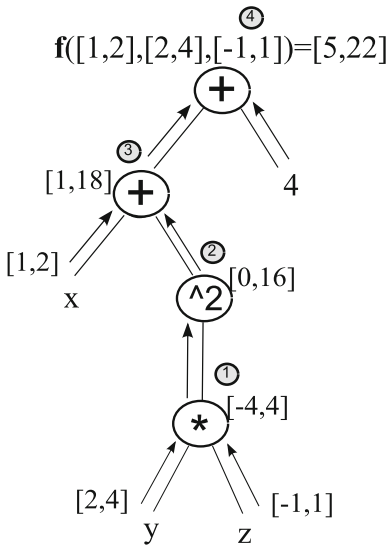


Fig. 1 The evaluation of $[1, 2] + ([2, 4] * [-1, 1])^2 + 4$ performed in the corresponding DAG

w.r.t. x_i. These variables are fixed to the value that maximizes (resp. minimizes) the evaluation of f. Then, an upper bound (resp. lower bound) of the image of f is computed.

$$\boldsymbol{f}_M(\boldsymbol{x}) = \left[\underline{\boldsymbol{f}_N(x_1^-, \ldots, x_n^-, \boldsymbol{w})}, \overline{\boldsymbol{f}_N(x_1^+, \ldots, x_n^+, \boldsymbol{w})}\right]$$

x_i^+ (resp. x_i^-) is the value of the monotonic variable x_i that maximizes (resp. minimizes) f. $\boldsymbol{w}$ is a vector of non-detected monotonic variables. Note that the bounds of the image of f are computed using the natural extension. Any other extension could be used instead.

The *occurrence grouping extension* OG [2] is a variant of ME. It generates a new function f^{og} that increments the number of monotonic variables of $f \cdot f^{og}$ is then evaluated using ME. The increment of monotonic variables in f^{og} implies that OG always computes a better (or equal) image approximation than ME.

Consider, for example, the function $f(x, w) = x^2 - 5x - w^3 + 2w^2 + 14w$ and the variable domains $\boldsymbol{x} = [0, 2]$ and $\boldsymbol{w} = [-2, 1]$. The natural evaluation of f in the domains is $[-39, 53]$. By using an interval gradient, the function is detected to be monotonically decreasing w.r.t. x. Thus, ME computes a sharper interval image: $[-35, 39]$. OG generates a new function, f^{og}, replacing each occurrence of the Non-detected monotonic variable w by a convex linear combination $r_a w_a + r_b w_b + r_c w_c$ such that f^{og} is monotonically increasing w.r.t. w_a and monotonically decreasing w.r.t. w_b:

$$f^{og}(x, w_a, w_b, w_c) = x^2 - 5x - w_a^3 + 2(0.25w_a + 0.75w_c)^2 + 14w_a$$

Finally, by evaluating f^{og} using ME, OG computes a better image approximation of f: $[-26, 16.25]$.

In general, Taylor extensions [63,75] are of the form: $\boldsymbol{f}_{T_d}(\boldsymbol{x}) = \boldsymbol{P}_d(\boldsymbol{x} - \tilde{x}) + \boldsymbol{e}_d$, where $P_d(x)$ is a degree-d polynomial approximation of f over $\boldsymbol{x}$, $\tilde{x}$ is a base point (often the midpoint of $\boldsymbol{x}$), and the interval $\boldsymbol{e}_d$ encompasses the truncation error of the polynomial over $\boldsymbol{x}$. In particular, **the first-order Taylor extension** follows a derivation of the mean value theorem, and it is given by the following definition [90]:

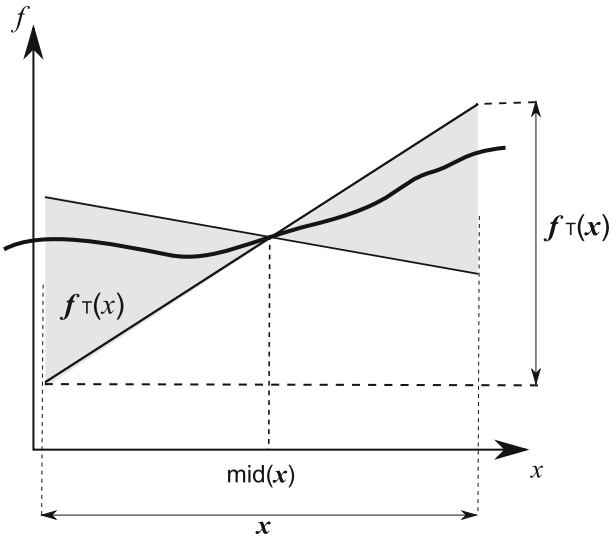


Fig. 2 The linearization of f by using the Taylor extension. $\boldsymbol{f}_T(\boldsymbol{x})$ computes the image of the linearization

$$\boldsymbol{f}_T(\boldsymbol{x}) = f(\text{mid}(\boldsymbol{x})) + \sum_{i=1}^{n} \boldsymbol{a}_i(\boldsymbol{x}_i - \text{mid}(\boldsymbol{x}_i))$$

where $\boldsymbol{a}_i$ is an interval overestimation of the image of $\frac{\partial f}{\partial x_i}$ over $\boldsymbol{x}$. Basically, the extension computes the image of a non-convex interval linearization of f. The linearization is obtained by using the interval gradient of f in the box (see Fig. 2). A better approximation is made by **Hansen's extension** $\boldsymbol{f}_H$ [50]. Hansen's extension can be observed as a recursive variant of the first-order Taylor extension.

$$\boldsymbol{f}_H(\boldsymbol{x}) = f(\text{mid}(\boldsymbol{x})) + \sum_{i=1}^{n} \boldsymbol{a}'_i(\boldsymbol{x}_i - \text{mid}(\boldsymbol{x}_i))$$

where $\boldsymbol{a}_i$ is an interval overestimation of the image of $\frac{\partial f}{\partial x_i}$ over $\boldsymbol{x}^{(i)}$, and the box $\boldsymbol{x}^{(i)}$ is equivalent to $\boldsymbol{x}$ with $n - i$ intervals of zero width:

$$\boldsymbol{x}^{(i)} = \boldsymbol{x}_1 \times \cdots \times \boldsymbol{x}_i \times \text{mid}(\boldsymbol{x}_{i+1}) \times \cdots \times \text{mid}(\boldsymbol{x}_n)$$

Another type of Taylor extension is based on *interval slopes* [66]. Given two points x and c in box $\boldsymbol{x}$, the vector $f[x, c] \in \mathbb{R}^n$ is called a slope of f between x and c if $f(x) = f(c) + f[x, c](x - c)$ holds. Consider a point $c \in \boldsymbol{x}$ and $\boldsymbol{f}[\boldsymbol{x}, c] \in \mathbb{IR}^n$ an interval slope that bounds $f[x, c]$ over all $x \in \boldsymbol{x}$. The extension based on interval slopes is given by:

$$\boldsymbol{f}_S(\boldsymbol{x}) = f(c) + \boldsymbol{f}[\boldsymbol{x}, c](\boldsymbol{x} - c)$$

Slopes can be calculated in a similar fashion as the interval derivatives [13,110]. Extensions based on interval slopes usually provide better image approximations than the first-order Taylor extension, particularly if the center of the slope is carefully chosen [78].

3 Problem formulations

In this section, we describe the two problems that will be addressed in this work.

Definition 1 (*Numerical constraint satisfaction problem* (*NCSP*)) A **numerical CSP**, or constraint system $S = (C, x, \boldsymbol{x})$, consists of a vector $x = (x_1, \ldots, x_n)$ of variables varying in a box $\boldsymbol{x} \in \mathbb{IR}^n$ and a set of constraints C. C includes a set of equality constraints $h_1(x) = 0, \ldots, h_p(x) = 0$, where $h_i: \mathbb{R}^n \to \mathbb{R}, \forall i$, and a set of inequality constraints $g_1(x) \leq 0, \ldots, g_q(x) \leq 0$, where $g_j: \mathbb{R}^n \to \mathbb{R}, \forall j$. A **solution** $x' \in \boldsymbol{x}$ of S satisfies all constraints.

When used to compute the set of all of the solutions of an NCSP, interval-based methods generally find a set of *atomic* boxes B_ϵ (i.e., boxes smaller than the required precision), such that all of the solutions to the problem belong to at least one of these boxes. It is possible that there are some boxes in B_ϵ that do not contain any solution to the problem.

Definition 2 (*Numerical Constrained global Optimization Problem* (*NCOP*)) Consider the constraint system $S = (C, x, \boldsymbol{x})$, where x is a vector of n variables. Consider a real function $f_{obj}: \mathbb{R}^n \to \mathbb{R}$. The **numerical constrained global optimization problem**, related to the **objective function** f_{obj} and the system of constraints S, consists of finding:

$$x^* = \min_{x \in \boldsymbol{x}} f_{obj}(x) \text{ s.t. } x \text{ satisfies all of the constraints in } C$$

If $x' \in \boldsymbol{x}$ is the solution to S, then it is said to be **feasible**.

In this case, the objective of the interval-based optimizer is generally to find one ϵ-**optimal** solution, i.e., a feasible point x'[1] such that $f_{obj}(x') - f_{obj}(x^*) \leq \epsilon_{obj}$, where ϵ_{obj} corresponds to the desired precision of the method. Some variants try to find a set of atomic boxes containing all of the ϵ-optimal solutions. Although we refer primarily to the former optimizers in the following, most of the presented methods are applicable to both of them.

4 Interval-based algorithm

In this section, we describe the basic structure of interval-based algorithms for NCOPs and NCSPs. Both types of algorithms are typically based on the B&B strategy. The main procedures and methods, which are covered in the next sections, have been highlighted in bold.

4.1 NCSP algorithm

Consider the NCSP $S = (C, x, \boldsymbol{x})$. Algorithm 1 shows the basic skeleton of an NCSP algorithm. The search tree is implemented using a list of leaves, L. It achieves an exhaustive exploration of the search space with the objective of finding a set of atomic boxes containing all of the solutions. The procedure is initialized with a box, $\boldsymbol{x}$, that contains the initial domain of each variable in the system. The algorithm returns a set of atomic boxes, B_ϵ, that contain all of the solutions to the problem.

In each iteration, a node is selected from L. Usually, the algorithm follows a *depth-first search strategy*. This can be implemented by treating L as a stack; i.e., nodes are removed from and inserted at the front of L. Each node is treated by two methods: branching/bisection and pruning. The branching, or **bisection**, consists of splitting the current box into several

[1] or an atomic box containing a feasible point.

Algorithm 1 `NCSPalgo`$(C, x, \boldsymbol{x}, \epsilon)$; **out:** B_ϵ

```
L ← {x}
while L ≠ ∅ do
  x ←select-and-remove(L)
  (x^r, x^l) ← bisect(C, x, x)
  for all x ∈ {x^r, x^l} do
    x ← pruning(C, x, x)
    if x ≠ ∅ then
      if is-atomic-box?(x, ε) then
        B_ε ← B_ε ∪ {x}
      else
        insert(L, x)
      end if
    end if
  end for
end while
```

sub-boxes (generally two). The new boxes are then treated by the pruning method. Two decisions are made by the bisection method: on which variable/interval to bisect the box and on which value of the domain of the variable to do so. More details about branching/bisection methods are given in Sect. 6.

After branching, the `pruning` procedure treats the new pair of boxes by turn. This procedure is compounded by one or several contraction methods or by **contractors**. Contractors attempt to filter the boxes by eliminating inconsistent values from their bounds without a loss of solutions. If all values in a box are filtered, an empty box is returned. Contractors are studied in more detail in Sect. 5.

After pruning, empty boxes are discarded from L. Non-empty boxes are put in L only if they are not *atomic boxes*; otherwise, they are put in B_ϵ. At the end of the search, B_ϵ contains the set of all solutions to the problem. There are some techniques to show the existence of a unique solution in a box (e.g., interval Newton [90]). They are used to detect atomic boxes in B_ϵ that certainly contain solutions. These techniques work only when certain conditions are satisfied in the box.

Observe the example in Fig. 3. The node/box 1 represents the initial domain of the variables x and y. Each curve in the box represents an equality constraint; Thus, each point of a curve satisfies the corresponding constraint. The problem has two solutions: the two intersections of the curves.

In the first iteration of the algorithm, box 1 is selected. It is bisected on x, and two new boxes are created. Each of them is contracted by using the pruning method. The contraction reduces each box to a smaller white box. These boxes are not atomic; therefore, they are added into the stack L. In the second iteration, box 2 is taken from L. (It is the first box in the stack.) The procedure is repeated for this node; the box is bisected on y, and each of the sub-boxes is pruned. Note that the algorithm has detected that the right box does not contain any solution; therefore, it is discarded. However, the left box is just contracted and put in L. The search continues with box 3 and its entire corresponding subtree before passing on to node 4. When node 4 is treated, two sub-boxes are again generated by the bisection. In this case, the pruning reduces the left box to an atomic one, which is added to B_ϵ. The right box is discarded by the pruning.

Although the depth-first search strategy is usually used due to its constant complexity in memory, some variants have been proposed to quickly discover potential solutions in different areas of the search space [24].

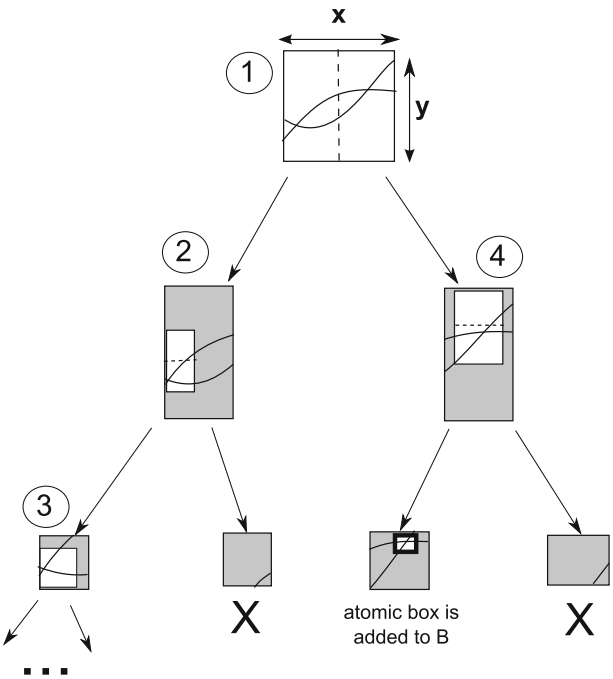


Fig. 3 Example of a search tree for an NCSP problem consisting of two variables and two constraints

A well-known problem related to interval B&B algorithms (NCSP and NCOP algorithms) is known as the *cluster problem* [30,109]. The cluster problem (or cluster effect) consists of the excessive splitting of boxes close to a solution and the failure to remove many boxes not containing the solution. As a consequence, these algorithms slow down considerably once they reach regions close to the solutions. In [109], Schichl and Neumaier discuss the reasons why this problem occurs. They also propose methods for defining exclusion regions (boxes and polytopes) around each solution found. These regions are guaranteed to contain no other solution and thus can be safely discarded. Exclusion regions can also be defined around arbitrary points guaranteed to contain no solution.

Another issue of NCSP algorithms is related to finding regions of solutions when the system is under-constrained. Only a few works [33,38,40,105,108,125] address this problem. In general, if the precision of the solver is set to be too small, the number of atomic boxes containing solutions may overflow the memory of the machine. Furthermore, from such a large number of atomic boxes, it may be very difficult to obtain useful information. When the system becomes highly dimensional, the solution set may be better visualized in other ways (e.g., projection of the solution set over certain parameters/variables [51]). In [122], the authors propose a method for constructing compact approximations of the complete set of solutions. They combine an efficient representation of orthogonal polyhedra [14] with adapted splitting strategies to construct the approximation as unions of boxes. In [40], the authors propose representing the search space using a parallelepiped instead of boxes to extend the power of the interval Newton to under-constrained systems.

4.2 NCOP algorithm

Algorithm 2 details a basic implementation of an NCOP solver. As in the NCSP solver, starting from an initial node, the NCOP solver builds a search tree using a B&B schema. In this case, each node is represented by a pair of variables, $(\boldsymbol{x}, lb)$. $\boldsymbol{x}$ corresponds to the domain space of the variables, and lb is a (generally infeasible) lower bound for the objective function, f_{obj}, in the box $\boldsymbol{x}$ and with the constraints C. In the root node, lb takes the value $-\infty$. The value is improved by the pruning method. LB corresponds to the minimum value of the lower bounds for all leaf nodes. Additionally, UB is the cost of the current best feasible point found during the search. A termination occurs when $UB - LB$ reaches a precision ϵ_{obj}, and a *quasi-optimal solution* x_{UB} with cost $UB = \overline{f_{obj}(x_{UB})}$ is returned.

The same **branching** and **contractors** of the NCSP solver may be used here. An additional constraint related to the objective function may be added to increase the pruning: $f_{obj}(x) \leq UB$. Instead of just adding the constraint, we can add a new variable y with initial domains $\boldsymbol{y} = [lb, UB]$ and the constraint $y = f_{obj}(x)$. In this way, an eventual contraction of the left bound of $\boldsymbol{y}$ implies an improvement of the lower bound corresponding to the node: $lb \leftarrow \underline{y}$. To improve the lower bound, some solvers also compute an approximation of the image using different interval extensions (e.g., [78]).

In unconstrained problems (i.e., $C = \emptyset$), a monotonicity test may be applied to box $\boldsymbol{x}$. If f_{obj} is monotone w.r.t. some coordinates of $\boldsymbol{x}$, then each related interval can be reduced to its respective lower or upper bound. In these problems, it is also possible to add the system of equations $\bigtriangledown f_{obj}(x) = 0$ to restrict the search to the stationary points of f_{obj} in $\boldsymbol{x}$. These additional equations can be used only in boxes strictly contained in the initial box [78].

Algorithm 2 `NCOPalgo`(**in:** $f_{obj}, C, x, \boldsymbol{x}, \epsilon, \epsilon_{obj}$); **out:** x_{UB}, UB, B

```
L ← {(x, −∞)}; UB ← +∞; LB ← −∞
while L ≠ ∅ and UB − LB > ε_obj do
  (x, lb) ← select-and-remove(x, lb)
  (x^l, x^r) ← bisect(C, x, x)

  for all x ∈ {x^l, x^r} do
    (x, lb') ← pruning(f_obj, C, x, x, UB)
    if x ≠ ∅ and lb < UB then
      (x', new_UB) ←upper-bounding(f_obj, C, x, x, UB)
      if new_UB < UB then
        (x*, UB) ← (x', new_UB)
        remove from L and B_ε any pair (x, lb) such that lb > UB
      end if
      if is-atomic-box?(x, ε) then
        B_ε ← B_ε ∪ {(x, lb')}
      else
        insert(L, (x, lb'))
      end if
    end if
  end for
  LB ← min_{(x,lb)∈L∪B_ε} lb
end while
```

The **upper bounding** consists of finding feasible solutions in $\boldsymbol{x}$ with costs lower than the current UB. The procedure returns a feasible point or a *certified box* (an

atomic box that certainly contains a feasible solution) plus an upper bound of the image of the objective function over this point/box (new_UB). If $new\_UB < UB$, then UB is updated. Updating UB implies a *global* reduction of the feasible space through the constraint $f_{obj}(x) \leq UB$. This potential reduction in the feasible space makes the upper bounding a crucial component in global optimizers. Cheap local search methods are generally used to find feasible points (e.g., selecting the midpoint of the box [96], applying the Newton method [41,70], extracting convex inner regions [5]), plus some feasibility tests (e.g., the natural evaluation of the candidate point [5,96], the Borsuk proof [70]). More details on the upper bounding techniques are given in Sect. 7.

For the same reason as upper bounding, the **selection of the next box** (procedure `select-and-remove`) is crucial in global optimizers. In each iteration, the algorithm should select the next box that is most likely to improve the UB the most to reduce the feasible space. The most used strategy is the best-first using as a heuristic the box with the lowest lb [5,70,96]. The algorithm hopes to find in this box a feasible point with cost lb, in which case, the search would end. In [18,77], the authors propose a set of original heuristics that also account for the size and feasibility of the box. More details about box-selection techniques are given in Sect. 8.

Nodes resulting in empty boxes after pruning and nodes that do not contain the global optimum (i.e., $lb > UB$) are discarded. If $\boldsymbol{x}$ becomes atomic, then it is added to a list B_ϵ with its associated lower bound. These boxes are not further contracted, though their lower bounds are considered for computing LB. If $\boldsymbol{x}$ is not atomic, then the pair/node is inserted on the list L. At the end of the iteration, LB is updated using the lowest lower bound in L and B_ϵ.

At the end, the solver returns one of the following results depending on the reached termination criteria:

- If $UB - LB \leq \epsilon_{obj}$, the solver returns an ϵ-optimal solution of the problem x_{UB} and its cost UB.
- Otherwise, the solver has most likely not found a solution with a cost that is sufficiently close to the optimal cost. Regardless, it returns a set of atomic boxes, B_ϵ, and the best found solution, x_{UB}. Either B_ϵ contains the optimum of the problem, or x_{UB} is an ϵ-optimal solution with cost UB.
- If neither boxes nor a feasible solution is returned, then the problem is unfeasible.

A simple way to modify the algorithm to find all of the optimal solutions consists of removing the termination criteria $UB - LB > \epsilon_{obj}$. In this way, the algorithm would return a set of atomic boxes B_ϵ containing all of the optimal solutions of the problem plus a feasible solution x_{UB}.

Figure 4 shows an example of the tree search. The node/box 1 represents the initial domain of the variables. The area inside the curve in the box represents the feasible solution space. The small star indicates the location of the optimal solution with cost of -2.5 (this value is not known by the algorithm at the beginning). In the first iteration, node 1 is selected and bisected. Its two children are contracted, and the contractors compute new lower bounds, (-7 and -6), for each child. Then, the `upper-bounding` procedure attempts to find feasible solutions to improve the current upper bound. The method was successful in the left child; it found a new upper bound equal to 10. Both children are added to L. In the second iteration, the node with the lowest upper bound is selected (node 2) and bisected, and its children are pruned. Note that the contractors may remove feasible solutions from the boxes. This is because these methods use the additional constraint related to the current

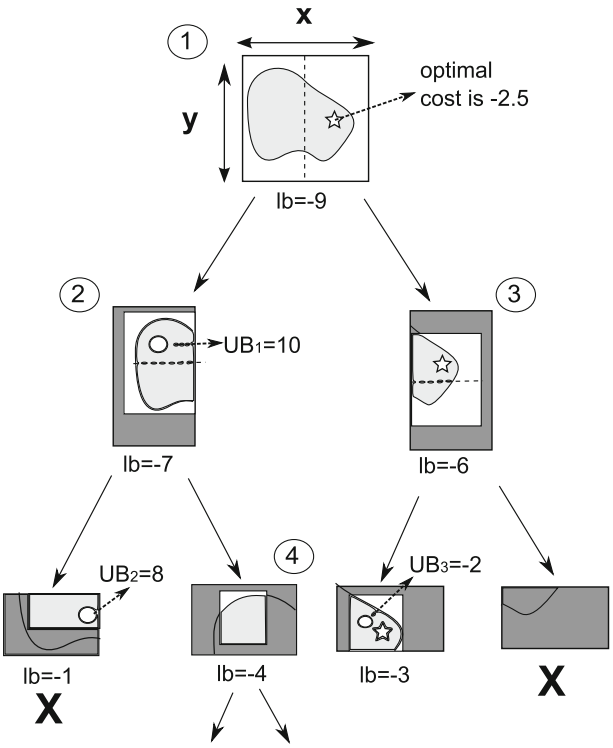


Fig. 4 Example of a search tree for an NCOP problem consisting of two variables

upper bound, i.e., $f_{obj}(x) < 10$. A new upper bound is found in the left child of node 2 ($UB = 8$). In the third iteration, node 3 is selected. After pruning, a new upper bound equal to -2 is found in its left child. Then, the box with $lb = -1$ is removed from L because its lower bound is greater than the current upper bound. The search continues with box 4…

4.3 Advanced features: use of optimality conditions

A way to reduce the cluster effect in NCOP problems is to use *optimality conditions*. Optimality conditions, such as Karush–John or Karush–Kuhn–Tucker conditions [54,55], generally state that for any local optima, a set of constraints must be satisfied; therefore, they can be used to filter the domains of variables. Currently, only a few interval-based optimizers incorporate these constraints to improve the filtering. In [35,50,57], the constraint system is solved using interval Newton, whereas techniques from constraint programming are applied in [44]. In [60], Kearfott states that interval Newton often fails in contracting the box using optimality conditions because the Jacobian matrix is commonly singular. Constraint propagation and relaxation-based contractors have nothing to do with the Jacobian matrix; furthermore, the nature of the optimality conditions is completely different from the nature of the system of constraints. For these reasons, we believe that contractors, distinct from interval Newton, may help to improve the efficiency of NCOP solvers when the optimality conditions are considered.

5 Contractors

Note that one of the main components shared by the NCSP and NCOP solvers is the contractors. These methods attempt to eliminate values from the domains of variables (search spaces) that do not satisfy one or more constraints in the constraint system. Contractors do not lose solutions, i.e., they are mathematically rigorously implemented and do not involve heuristics that could ignore solutions. We distinguish three types of contraction algorithms: contractors from interval analysis, contractors from constraint programming (or CP contractors) and linear relaxation-based contractors.

5.1 Contractors from interval analysis

One of the most effective filtering algorithms used in interval analysis is the extension of the Newton method to intervals (the *interval* Newton method). The objective of the classical Newton method is to find successively better approximations to the zeroes of a real function. Consider a continuous and differentiable univariate function $f(x)$ and its derivative $f'(x)$. If $x^{(l)}$ is an approximation of a solution of the equation $f(x) = 0$, then a better approximation is obtained by:

$$x^{(l+1)} = x^{(l)} - \frac{f(x^{(l)})}{f'(x^{(l)})}$$

The interval Newton method generalizes the procedure to intervals. Consider a univariate function $f(x)$ and an initial interval $\boldsymbol{x}$. The procedure applies successively the contraction step:

$$\boldsymbol{x} \leftarrow \boldsymbol{x} \cap \left(\text{mid}(\boldsymbol{x}) - \frac{f(\text{mid}(\boldsymbol{x}))}{[f'](\boldsymbol{x})}\right)$$

for contracting the interval $\boldsymbol{x}$. If the procedure converges to a fixed point, then it will return an atomic interval containing the only solution of the equation $f(x) = 0$ in $\boldsymbol{x}$. If an iteration of the contraction step returns an empty interval, then there is no solution in $\boldsymbol{x}$. When the iterations do not converge to a fixed point, we cannot predict the presence or absence of solutions in $\boldsymbol{x}$.

The generalization of the method to n variables and n equations is obtained by using the mean value theorem. (further details may be found in [50,90]) to linearize the system. Consider a vector $x = (x_1, \ldots, x_n)$ of variables and a function vector $f = (f_1, \ldots, f_n)$. Using the mean value theorem, it is easy to show that the roots of f in a box $\boldsymbol{x}$ verify the following expression:

$$\exists A \in \boldsymbol{J}(\boldsymbol{x}) \;\text{ such that }\; A.y = -f(\text{mid}(\boldsymbol{x})) \tag{1}$$

where $\boldsymbol{J}$ is an interval extension of the Jacobian matrix of f in box $\boldsymbol{x}$. In other words, each element (i, j) of the matrix $\boldsymbol{J}(\boldsymbol{x})$ contains an image approximation of $\frac{\partial f_i}{\partial x_j}(x)$ over box $\boldsymbol{x}$. y is a vector of auxiliary variables: $y = x - \text{mid}(\boldsymbol{x})$. $\text{mid}(\boldsymbol{x})$ is the midpoint of the vector $\boldsymbol{x}$. Consider $\boldsymbol{y}^s$ to be a box that contains all points that satisfy the relation (1). Considering that $x = y + \text{mid}(\boldsymbol{x})$, the contraction step performed by the *multivariate interval Newton* is

$$\boldsymbol{x} \leftarrow \boldsymbol{y}^s + \text{mid}(\boldsymbol{x})$$

There are many variants of the interval Newton method, and they differ primarily in regard to the algorithm that is used to find $\boldsymbol{y}^s$. Some solvers (e.g., Ibex [23]) implement Hansen's

matrix [49] instead of the Jacobian matrix. Hansen's matrix contains sharper coefficients, allowing one to obtain a better linearization of the system. In [80], the author symbolically manipulates the elements of the matrix. Reducing multiple occurrences of variables leads to a better image approximation of the partial derivatives. Soares [113] proposes using affine arithmetic instead of interval arithmetic to maintain the linear dependence among variables. There are several different ways to linearize the system, e.g., the methods of Krawczyk [67], Borsuk [37], and Kantorovich [28,97,116]. Several methods are also used to solve the linear system, e.g., matrix inversion, Gauss–Seidel, Hansen–Bliek [50], Gaussian elimination, and LU. *Preconditioning* [46] is a technique commonly used to improve the precision of the box $\boldsymbol{y}^s$. The preconditioning performs a type of reparation of an ill-conditioned matrix $\boldsymbol{J}(\boldsymbol{x})$ to obtain more precise solutions.

The main strength of the interval Newton contractor is that it is Global, i.e., it considers all of the system to be only one constraint, thereby reducing all of the variable domains simultaneously. However, the main limitations of the interval Newton contractor are (1) it works only with equality constraints, (2) it generally works with *well-constrained* systems (i.e., square systems of independent equations with a finite number of solutions), and (3) it generally converges only when boxes are small enough [48]. In large boxes, due to interval arithmetic operations, the size of the solution $\boldsymbol{y}^s$ may increase enormously, preventing the convergence of the contraction step.[2]

5.2 Constraint-propagation algorithms

The constraint-propagation algorithms are issued from constraint programming over intervals. These algorithms focus on the domain reduction w.r.t. *only one* equation/constraint and are performed by a *revision* procedure. A propagation algorithm, similar to the well-known `AC3` algorithm [74], is then used for propagating the reductions to all of the system until a fixed point is reached.

All propagation algorithms used by interval-based solvers have similar `AC3`-like propagation procedures, which differ primarily in the revision procedure. Generally, revision procedures attempt to enforce partial consistencies over the elements of the domains. Partial consistencies arising from finite-domain CSPs do not seem reasonable when we address constraints on the real numbers (in fact, on the floating-point numbers) due to the very large domains. Hence, new partial consistencies have been defined by the interval community.

Hull consistency [11] considers only the bounds of the intervals. Revision methods enforcing hull consistency should find the minimal interval for a variable x_i containing all consistent values related to a constraint C_j and a box $\boldsymbol{x}$. However, primarily due to the dependency problem, it is generally not possible to implement such methods efficiently.

An intuitive way to filter the domains of variables is to isolate the variable that we want to filter and compute the image of the generated *projection function*. For instance, consider the constraint $x_1 + 2x_2 = 10$, with domains $\boldsymbol{x}_1 = [0, 5]$ and $\boldsymbol{x}_2 = [1, 3]$. If we want to reduce the domain of $\boldsymbol{x}_1$, we can isolate x_1 in this way: $x_1 = 10 - 2x_2$. Then, by using some interval extension (e.g., the natural extension), we can compute an image approximation of the left side of the expression or **projection function** of x_1. The result should then be intersected with the initial domain of $[x_1]$:

$$\boldsymbol{x}_1 \leftarrow \boldsymbol{x}_1 \cap (10 - 2\boldsymbol{x}_2) = [0, 5] \cap [4, 8] = [4, 5]$$

[2] By using appropriate preconditioners, the contractor can be used for contracting *certain coordinates*, even for some large boxes and in some singular cases [56].

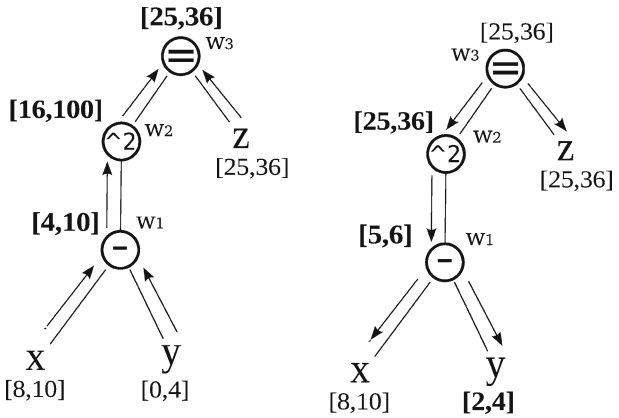


Fig. 5 Evaluation and projection phases of the `HC4-Revise` procedure in the constraint $(x - y)^2 = z$, with domains $\boldsymbol{x} = [8, 10]$, $\boldsymbol{y} = [0, 4]$ and $\boldsymbol{z} = [25, 36]$

Note that as the image of the projection function is computed using interval extensions, this method also suffers from the dependency problem. Thus, it generally cannot compute the hull consistency. A second problem occurs when a variable appears several times in a constraint, and the isolation is generally impossible. In this case, each occurrence can be isolated independently. For instance, consider the constraint $x^3 + 3x = 14$, with domain $\boldsymbol{x} = [0, 3]$. To reduce the domain of the variable, we could consider the projection functions of each occurrence and intersect their image approximations with the interval $\boldsymbol{x}$:

$$[x] \leftarrow \boldsymbol{x} \cap (-3\boldsymbol{x} + 14)^{1/3}$$
$$[x] \leftarrow \boldsymbol{x} \cap \frac{-\boldsymbol{x}^3 + 14}{3}$$

In the example, we obtain $x = [1.70, 2.42]$. After several iterations of the procedure, $\boldsymbol{x}$ converges to the solution ([2, 2]). However, if the projection functions have several occurrences of the variable to project (i.e., the variable appears more than twice in the constraint), due to the dependency problem, the projection will converge to sub-optimal domains. We refer to this particular problem as the **isolation problem**.

The algorithm `HC4-Revise` [11] filters the domains of variables without explicitly generating the projection functions. `HC4-Revise` requires traversing the expression DAG only twice to perform the projection over each occurrence of a variable. The algorithm works in two phases. The *forward* phase is recursively performed from the leaves to the root (see Fig. 5-left). This phase computes, using interval arithmetic, the natural evaluation of the subexpressions represented by the DAG nodes.

Consider, for example, the node $w_2 = w_1^2$. The related interval evaluates $\boldsymbol{w}_1^2 = [4, 10]^2 = [16, 100]$. The *backward* phase traverses the DAG top down, applying in every node a related *projection operator* (see Fig. 5-right). The projection operator reduces the intervals of the nodes, eliminating inconsistent values w.r.t. the corresponding unary or binary basic operator. Consider, for example, the node w_1, corresponding to the sub-expression $w_1 = x - y$ in Fig. 5-right. The projection operator performs the following contraction over y

$$[y] \leftarrow [0, 4] \cap ([8, 10] - [5, 6]) = [2, 4]$$

If an empty interval is obtained during the backward phase, the initial domains do not satisfy the constraint, and an empty box is returned.

Note that `HC4-Revise` performs an automatic projection over the intervals, using the *natural extension*. A recent work [6] describes a variant of `HC4-Revise` called `MinMax-Revise`. It uses a monotonicity-based extension to perform the filtering (more precisely, the OG extension). Just as `HC4-Revise` does, `MinMax-Revise` suffers from the isolation problem.

In the global optimization community, procedures similar to `HC4` (the propagation algorithm that uses `HC4-Revise`) have been proposed. These procedures are known as feasibility-based bounds tightening (`FBBT`) [17,73,83,111]. Commonly, `FBBT` represents the whole constraint system as one large DAG with several roots. In this DAG, some common subexpressions appearing in several constraints are represented by single subgraphs with several parents.

`BC3-Revise` [11] and its variants address the isolation problem. To filter an interval related to a variable x_i in a box $\boldsymbol{x}$, `BC3-Revise` selects a sub-interval/slice $\boldsymbol{s}$ on a bound of $\boldsymbol{x}_i$. Then, through the use of an interval-based extension $\boldsymbol{f}$, it tests whether $\boldsymbol{f}(\boldsymbol{x}')$ satisfies the related constraint. $\boldsymbol{x}'$ is a sub-box of box $\boldsymbol{x}$, where the interval $\boldsymbol{x}_i$ has been replaced by $\boldsymbol{s}$. If the test fails, then $\boldsymbol{s}$ is removed from $\boldsymbol{x}_i$, and another bound slice is tested; otherwise, a smaller slice may be tested. Usually, `BC3-Revise` performs a dichotomic process to reach the *maximal reduction* on each interval bound. We call left-narrow (resp. right-narrow) the dichotomic process that improves the left (resp. right) bound of the interval. If a precision ϵ is given, then the complexity of the dichotomic process is time $O(1/\epsilon)$. Usually, to accelerate the convergence of `BC3-Revise`, the univariate interval Newton algorithm is called in each slice after performing the test.

To reduce the dependency problem, constraint-propagation-based algorithms may use sophisticated (but more expensive) interval extensions. For instance, `Mohc` [6] uses the occurrence-grouping extension, whereas the propagation algorithm proposed in [121] uses several interval extensions, including an affine arithmetic-based one.

5.2.1 The locality problem

In addition to the dependency problem, the constraint-propagation algorithms suffer from the **locality problem**. It is caused by the reduced scope of local consistencies (in discrete and continuous domains). Hull consistency, for instance, guarantees that for each constraint $f(x) \leq 0$ and for each bound of a variable, there exists a vector x_c such that $f(x_c) \leq 0$ is consistent. However, there is no guarantee that x_c is consistent in the other constraints.

Consider, for example, the system $\{x+y = 10; x-y = 0\}$, with domains $\boldsymbol{x} = [0, 10]$; $\boldsymbol{y} = [0, 10]$. The domains are hul consistent: For the value $x = 0$, the vector $(0, 10)$ is consistent in the first constraint, and the vector $(0, 0)$ is consistent in the second one. However, $x = 0$ is not globally consistent because $(0, 10)$ is not consistent in the second constraint, nor is $(0, 0)$ in the first one (the same analysis may be performed for each of the other bounds). Hull consistency may become global when the graph of the constraints has no cycles. However, this is not the general case.

Consider a constraint C and the system S, consisting of the primitive constraints of C. In particular, the filtering reached by `HC4-Revise` in C is equivalent to the filtering reached by `HC4` in S, assuming that both algorithms are carried out until the fixed point [11]. Note that constraint decomposition transforms a dependence problem into a locality problem (and vice versa). Thus, when revision algorithms stronger than `HC4-Revise` are used (e.g.,

`BC3-Revise` [21] or `Mohc-Revise`), it is not advisable to decompose the constraints. These algorithms face and reduce the dependency problem but not the locality problem.

Stronger consistency techniques (particularly kB consistencies) may partially reduce the locality problem (e.g., `3B` [72], `3BCID` [119]). However, enforcing kB consistencies with $k > 3$ is computationally expensive. Consider the $2n$ subproblems generated by replacing one interval (n in total) with one of its bounds. Thus, a system is **3B-consistent** [72] if each of these $2n$ subproblems is hull consistent. The 3B consistency is similar to SAC for CSP, but it is limited to the bounds of the domains. Algorithms enforcing the 3B consistency (e.g., `3B`, `ABT` [10], `3BCID`[3]) generally split the intervals into several sub-intervals (slices) of width w. Then, the *3B(w)-consistency* is enforced, discarding the slices at the bounds by using the underlying revised hull consistency algorithm (e.g., `HC4`, `Mohc`, `Box`).

5.2.2 Issues and perspectives related to constraint-propagation algorithms

In this section, we refer to advanced features and promising ideas that, we think, merit further research. They primarily address the locality and dependence problems.

5.2.3 Common subexpression elimination

Common subexpression elimination (CSE) is an alternative for addressing the locality problem. It consists of searching for identical or common subexpressions and analyzing whether it is worthwhile to replace them with a single auxiliary variable. In [1], it was shown that transforming the original system through use of CSE improves the contraction power of the classical contractor `HC4`. This behavior may be explained by the reduction performed on the domains related to auxiliary variables (projection information). In the original system, the projection information is lost; however, in the new system, auxiliary variables not only store the projection information but also share it with other constraints. Algorithms such as `FBBT` and `FBPD` (forward-backward propagation on DAG [123]) detect *some* common subexpressions on the system and represent them by nodes with several parents in a DAG representation of the whole constraint system. In the work of Vu et al. [123], it is noticeable that merging equivalent nodes of the DAG helps to reduce the solver time by several orders of magnitude. This is due to CSE.

On this particular subject, there remains much to explore. Currently, CSE- and DAG-oriented techniques only replace expressions that are exactly equal. For example, the current techniques do not detect common subexpressions among the expressions $x + y$ and $2x + 2y$; however, according to [1], it is worthwhile to replace $x + y$ with an auxiliary variable v, thus generating the equivalent expressions v and $2v$.

5.2.4 Linear combination of constraints

Combining linearly the constraints to obtain an easier (or better conditioned) problem to solve seems to be a promising but unexploited idea to address the locality problem. The interval Newton, for instance, performs a preconditioning of the system using a preconditioning matrix. This approach, however, works on a linear relaxation of the original system, thus losing important nonlinear information. A few works [21,124] propose applying linear combination techniques to improve constraint-propagation algorithms. Although the results are promising, the application of these methods is restricted to a few types of instances. A

[3] Actually, `3BCID` enforces CID consistency [119], a slightly stronger one.

drawback of these methods is that if special care is not taken, the constraints of the new system may involve many more terms than the original set of constraints, thus increasing the dependency problem. We expect that improvements of current techniques for approximating the image of functions will offer promising new opportunities for techniques based on the combination of constraints.

5.2.5 Symbolic preprocessing for reducing the dependency problem

Related to the dependency problem, we believe that automatic symbolic computation mechanisms are missing in most of the currently competitive interval algorithms. The sub-distributivity property of intervals states that $\boldsymbol{x}_1(\boldsymbol{x}_2 + \boldsymbol{x}_3) \subset \boldsymbol{x}_1\boldsymbol{x}_2 + \boldsymbol{x}_1\boldsymbol{x}_3$. In other words, if we have a polynomial with a common factor, then we obtain a narrower image (computed using the natural extension) if this factor is factored out. This is generally true not only for the natural extension but also for most of the interval extensions. Thus, what we should do is relatively obvious: we should factor out the common factors of polynomials to reduce the number of occurrences of variables. Of course, there is no optimal factorized form, but using simple greedy algorithms appears to be sufficient [20,22].

5.3 Linear relaxation-based contractors

There are many approaches that use linear-relaxation methods to solve optimization problems. However, most of them (e.g., the famous Baron [117]), though fast and complete, are not rigorous; i.e., they cannot guarantee their results.

Related to interval-based solvers, several linear relaxation-based contractors have emerged in recent years. They face important drawbacks compared to other interval-based approaches: the locality problem, related to constraint-propagation techniques, and the restrictive application of the interval Newton method and its variants. These methods usually work on a linear relaxation of the entire system or a part of it. The simplex algorithm is then used to narrow the domain of each variable by using the relaxed system i.e., the problem of minimizing x_i (resp. maximizing x_i) is solved to find a new lower bound (resp. upper bound) for each interval $\boldsymbol{x}_i$. Algorithm 3 shows the structure of a typical linear relaxation-based contractor [5,7,71].

Algorithm 3 `PolytopeHull`(**in:** C, x); **out:** $\boldsymbol{x}$

```
for all C_j ∈ C do
  polytope ← polytope Ulinear-relaxation(C_j)
end for
for all x_i ∈ x do
  x_i ← min x_i subject to polytope
  x̄_i ← max x_i subject to polytope
end for
```

The first loop on the constraints builds a *polytope* (i.e., a set of linear constraints), whereas the second loop on the variables contracts the domains, without a loss of solutions, by calling the simplex algorithm twice per variable.

The `linear-relaxation` method transforms a non-linear constraint C_j into a set of linear constraints Cl_j, such that any point satisfying C_j in $\boldsymbol{x}$ also satisfies the set of constraints Cl_j. The heuristics mentioned in [7] indicate in which order the variables should be handled, thus avoiding, in practice, calling the simplex algorithm $2n$ times. The application of simplex

must be mathematically rigorous. A cheap post-processing proposed by [94], which uses interval arithmetic, is commonly used to certify the solution returned by the algorithm.

There are only a few methods, based on linear-relaxation, currently used by interval-based solvers. These methods differ primarily in the linear relaxation technique. Figure 6 shows examples of the three linear-relaxation approaches explained below.

In [61], the authors propose a symbolic preprocessing step to relax nonconvex terms using linear underestimators and/or linear overestimators. The preprocessing first expands the original constraint system and objective function by decomposing the expressions. Intermediate variables are generated to represent intermediate operations.

For instance, consider the NCOP[4]: Minimize $f(x) = (x_1 + x_2 - 1)^2 - (x_1^2 + x_2^2 - 1)^2$, for $x_1 \in [-1, 1]$ and $x_2 \in [-1, 1]$. The preprocessing decomposes the objective function by adding intermediate variables, e.g., $v_3 = x_1 + x_2$, $v_4 = v_3 + 1$, and $v_5 = v_4^2$. The interval related to the variables may be initialized to the natural evaluation of the corresponding expressions, e.g., $\boldsymbol{v}_3 = [-2, 2]$, $\boldsymbol{v}_4 = [-3, 1]$, and $\boldsymbol{v}_5 = [0, 9]$. Then, the expanded equivalent problem is:

$$
\begin{aligned}
&\text{minimize } v_{11}\\
&\text{subject to } x_1 + x_2 - v_3 = 0\\
&\qquad v_3 - 1 - v_4 = 0\\
&\qquad v_4^2 - v_5 = 0\\
&\qquad v_1^2 - v_6 = 0\\
&\qquad v_2^2 - v_7 = 0\\
&\qquad v_6 + v_7 - v_8 = 0\\
&\qquad v_8 - 1 - v_9 = 0\\
&\qquad -v_9^2 - v_{10} = 0\\
&\qquad v_5 + v_{10} - v_{11} = 0\\
&\qquad x_1 \in [-1, 1], \quad x_2 \in [-1, 1]
\end{aligned}
$$

Each equality constraint in the expanded problem is analyzed to determine whether the "$=$" may be replaced by "$\leq$" or "$\geq$" such that the original and the expanded problems have the same set of optimizing points. This process is done automatically following a set of rules. In the example, the third, ninth and tenth constraints would be modified by $v_4^2 - v_5 \leq 0$, $-v_9^2 - v_{10} \leq 0$ and $v_5 + v_{10} - v_{11} \leq 0$, respectively.

Finally, the nonlinear constraints of the new problem are replaced by linear underestimators (for "$\leq$" constraints), overestimators (for "$\geq$" constraints) or both (for equality constraints). For example, the nonlinear inequality $v_4^2 - v_5 \leq 0$ can be replaced by the linear relaxation $(4.5)^2 + 9(v_4 - 4.5) - v_5 \leq 0$. The expression $(4.5)^2 + 9(v_4 - 4.5)$ corresponds to the tangent line to v_4^2 at $v_4 = 4.5$ (the midpoint of the interval $\boldsymbol{v}_4^2$). Each equality is replaced by one underestimator and one overestimator; for instance, $v_1^2 - v_6 = 0$ is replaced by $(0.5)^2 + (x_1 - 0.5) - v_6 \geq 0$ and $x_1 - v_6 \leq 0$. Convex operations can be approximated arbitrarily and closely by generating a large number of tangent lines. For instance, in Fig. 6-left, the constraint $x^2 = 0$ is approximated by one secant line (connecting the endpoints of the image of x^2 in the interval) and three tangent lines.

[4] Example from [61].

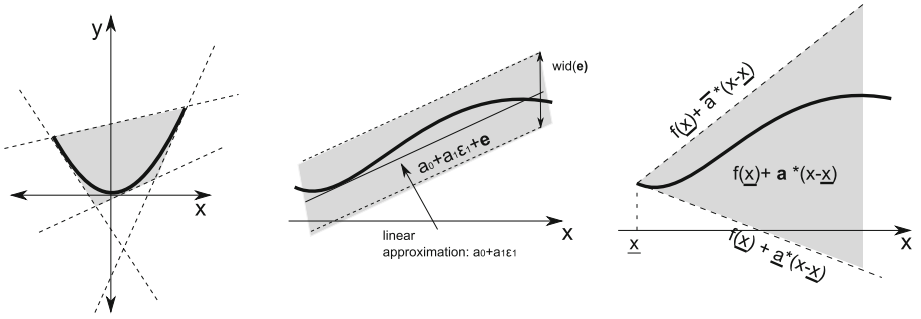


Fig. 6 Examples of linear relaxations related to a function. *Left* one linear overestimator and three linear underestimators applied to the constraint $x^2 = 0$. *Center* arithmetic affine applied to a function. Interval errors are represented by the single interval $\boldsymbol{e}$. *Right* interval Taylor form of a function f. In each case, the *gray area* corresponds to the generated polytope

In [82,86], the authors propose new affine and quadratic extensions for computing efficient and reliable bounds of functions. Extensions based on affine arithmetic are used in interval optimizers such as IBBA and IbexOpt.

Similar to interval arithmetic, computations in affine arithmetic are performed by extending primitive operators to operate on *affine forms* [27]. Affine forms are the representation of quantities in affine arithmetic. They are expressions of the following form:

$$\widehat{a} = a_0 + a_1\varepsilon_i + \cdots + a_p\varepsilon_p$$

where a_i are constants, and ε_i are unknown reals belonging to the interval $\boldsymbol{\epsilon}_i = [-1, 1]$.

Using affine arithmetic, we can construct linear relaxations of a function f over a box $\boldsymbol{x}$ while traversing the expression DAG of f in a bottom-up fashion. First, variables are replaced by their corresponding affine form. That is, each variable x_i is replaced by

$$\text{mid}(\boldsymbol{x}_i) + \frac{\text{wid}(\boldsymbol{x}_i)}{2}\varepsilon_i$$

The natural evaluation of the affine form is equal to interval $\boldsymbol{x}_i$.

Then, in each node corresponding to a linear operator (e.g., $+$ and $-$), the operator is applied to the child affine forms by simply adding or subtracting terms. As a result, we obtain a new affine form, which is attached to the node. Non-linear operators (e.g., $*$, / and unary non-linear functions) are approximated by affine operators, which add a new extra variable related to the non-linear error. For example, the multiplication of two affine forms can be approximated by the following affine form [25]:

$$\widehat{\widehat{a} \times \widehat{b}} := a_0b_0 + \sum_{i=1}^{p}(a_0b_i + a_ib_0)\varepsilon_i + \left(\sum_{i=1}^{p}|a_i| \times \sum_{i=1}^{p}|b_i|\right)\varepsilon_{p+1}.$$

Note that a new extra variable $\varepsilon_{p+1} \in [-1, 1]$ must be added.

At the end, in the root node of expression DAG, the affine arithmetic generates an interval function or, more precisely, a real linear function plus interval errors related to the nonlinear operators (see Fig. 6-center). This interval function corresponds to two linear inequalities, which approximate (overestimate and underestimate) the non-linear function f over box $\boldsymbol{x}$ (consider each variable ε_i, for $i = 1..n$, to be equivalent to $2 * \frac{x_i - \text{mid}(\boldsymbol{x}_i)}{\text{wid}(\boldsymbol{x}_i)}$).

In [4], the contractor `X-Taylor` is proposed. `X-Taylor` linearizes each constraint using an interval Taylor form of them. Unlike the classical Taylor form, which is expanded from

the midpoint of the box (see Fig. 2), `X-Taylor` uses a vertex of the box (see Fig. 6-right), thus creating a convex polytope.

For instance, if we expand the first-order Taylor of a function $f:\mathbb{R}^n \Rightarrow \mathbb{R}$ from the vertex $\underline{x} = (\underline{x_1}, \ldots, \underline{x_n})$ of a box $\boldsymbol{x}$, then we have for all $x \in \boldsymbol{x}$:

$$f(x) \in f(\underline{x}) + \sum_{i=1}^{n} \boldsymbol{a}_i (x_i - \underline{x_i}),$$

where $\boldsymbol{a}_i$ is an interval overestimation of the image of $\frac{\partial f}{\partial x_i}$ over $\boldsymbol{x}$. Replacing the interval $\boldsymbol{a}_i$ with its bounds, we obtain the following two hyperplanes enclosing the function f for all $x \in \boldsymbol{x}$:

$$f(\underline{x}) + \sum_{i=1}^{n} \underline{a_i}(x_i - \underline{x_i}) \leq f(x) \leq f(\underline{x}) + \sum_{i=1}^{n} \overline{a_i}(x_i - \underline{x_i})$$

From each vertex of box $\boldsymbol{x}$, we can analogously construct two hyperplanes enclosing the function f. Thus, we can construct 2^n linear underestimations (respectively overestimations) of f. Because creating a linear relaxation using the 2^n hyperplanes for each function is computationally expensive, several heuristics for selecting some vertices were proposed in [4]. Among them, the heuristic that randomly takes one vertex, and its opposite, offers the best numerical results. Indeed, using more than 2 vertices seems to be counterproductive in practice [4].

5.3.1 Issues and perspectives related to relaxation-based techniques

Non-rigorous algorithms apply more sophisticated relaxation-based techniques to reduce the variable domains (e.g., convexification [117], optimality-based reduction [106], reformulation-linearization technique [89,106,117], outer approximation techniques [31,76,79,87]). A main challenge is making these numerical methods mathematically rigorous. Because these techniques, in general, work with floating-point numbers, they return approximations of the real solution. Using approximations in filtering algorithms results in removing feasible regions and, possibly, the optimal solution. Some techniques have successfully been made mathematically rigorous and have been incorporated into interval solvers. In [94], Neumaier et al. present a technique to compute a safe bound of a linear program based on a standard simplex algorithm. Kearfott proposed a safe implementation of the optimality-based reduction technique in [58]; a reformulation-linearization technique is used in Icos [71]; and safe linearizations are used in several interval solvers [71,96,118].

5.4 Advanced features: adaptive mechanisms for selecting contractors

As times goes by, increasingly more contractors for interval B&B emerge. A typical set of contractors in an interval B&B consists of the following:

- propagation-based contractors: a contractor enforcing hull consistency (e.g., `HC4`, `Box` or `Mohc`), and sometimes contractors enforcing stronger consistencies (e.g., `3B`, `3BCID`).
- convex relaxation-based contractors: different alternatives depending on the convexification technique.
- interval Newton: primarily used for certifying solutions.

The contraction power of these contractors is generally not comparable. For instance, in highly non-linear and non-convex constraints, propagation-based contractors seem to be

more effective, whereas in constraints with a few nonlinear terms, convex relaxation-based contractors offer the best contraction. `Mohc` offers better contraction than `HC4` (particularly when functions are monotonic w.r.t all or several variables) but at higher costs. Similarly, strong consistency contractors are more effective but less efficient than contractors enforcing hull consistency.

An adaptive mechanism for deciding which revision procedure to use is proposed in [6]. Through this mechanism, the algorithm decides, for each constraint, whether it applies a weak but cheap revision method (`HC4-Revise`) or a stronger but more expensive one (`Mohc-Revise`). Similarly to works in finite domain CSPs (e.g., see [114,115]), the stronger contractor is applied only when there are high probabilities of filtering the search space using this contractor. Though `Mohc` suffers a decrease in its contraction power, the adaptive mechanism allows the algorithm to perform better than if any of the revision algorithms is applied alone.

Another adaptive interval-based contractor is proposed in [95]. The algorithm `ACID` (adaptive `3BCID`) partially enforces the CID consistency (similarly to SAC in finite domain CSPs). Instead of performing the procedure on the entire set of variables, `ACID` dynamically selects the number of variables that will be processed. The variables are sorted by decreasing influence using the SmearSumRel heuristic. In this way, the most promising variables are most likely to be processed. When the number of selected variables is 0, `ACID` performs a simple constraint propagation (e.g., a call to `HC4`). The number of processed variables is set through a learning phase. In this phase, the algorithm attempts to find the necessary number of variables to obtain most of the maximal reachable filtering. The learning phase is interleaved several times in the search to auto-adapt the value during the search. `ACID` is the current state-of-the-art contractor, enforcing strong consistencies used by the IbexOpt solver.

We also believe that some ideas from finite-domain CSPs can be adapted to interval-based algorithms. For instance, in [115], the authors experimentally study the behavior of constraint propagation when different local consistencies are applied. They show that in structured problems, propagations resulting in empty domains form clusters of very close calls to the propagation procedure. Considering this observation, they propose several heuristics to detect clusters to apply strong consistencies in them (high probability of pruning) while applying weak consistencies in the rest of the search (low probability of pruning). In [114], the authors use information gathered from a random probing preprocessing phase to select the propagation method that should be used on each constraint. The information provided by the random probing includes the percentage of fruitful revisions for each constraint, the local consistency responsible for each such revision and the number of value deletions caused by a given propagation method. This information is then used to select the best contractor for each constraint.

In interval B&B, it would also be interesting to identify, during the search, which constraints and/or variables have been *actively used* by the different contractors (e.g., constraints responsible for an empty box or filtered variables). Thus, each constraint/variable should be treated by its most promising related contractors.

6 Bisection methods

The bisection consists of two steps: the selection of the variable(s) to be split and the selection of the value(s) in which the box will be split. Commonly, the box is split at the midpoint of the selected interval.

Related to the variable selection, several algorithms have been proposed to dater [3,26,43,62,69,118,119]. Among the most used algorithms include round robin, largest-first and smear-based selection methods. **Round-Robin** is one of the simplest and most used methods. It does not require any information about the system. If the current k-dimensional box $\boldsymbol{x}$ was obtained by bisecting the interval $[x_i]$, then the Round-Robin method will select the variable x_j, with j= $(i + 1)$ *mod* k. The objective is to refrain from neglecting any variable. One weakness of this strategy is that a *bad* initial ordering of variables can lead to disastrous performance. **Largest-First** [90] (also known as the interval width-oriented rule [26]) simply selects the variable with the largest domain. It is based on the assumption that intervals with large diameters have a greater influence on the function evaluation. In the same manner as the Round-Robin technique, the largest-first method does not omit any variable. **Smear-based methods** [62,102] use information of the system to obtain the variable with the greatest *influence*. The influence of a variable x_i on a function f_j is computed by means of the *smear value*. Consider that the current node is associated with box $\boldsymbol{x}$; the smear value is given by

$$\text{smear}(x_i, f_j) = |\boldsymbol{a}_{ij}| * \text{wid}(\boldsymbol{x}_i)$$

where $\boldsymbol{a}_{ij}$ is an interval overestimation of the range of the partial derivative $\frac{\partial f_j}{\partial x_i}$ in $\boldsymbol{x}$.

Selection methods based on the smear value select the variable that maximizes an aggregation (e.g., sum or maximum) of this value in the whole system. A variant proposed in [118] uses a *relative smear value* instead (for more details, refer to [3]):

$$\text{smear}_{rel}(x_i, f_j) = \frac{\text{smear}(x_i, f_j)}{\sum_{k=1}^{n} \text{smear}(x_k, f_j)}$$

According to the experimentations performed in [2] (74 NCOP and 27 NCSP instances), using the relative smear value (heuristic SmearSumRel) seems to be much more robust than just using the smear value.

6.1 Issues and perspectives related to bisection methods

In finite-domain CSPs, many selection heuristics have been proposed to date (e.g., [8,12,15,16,42,112]). In general, they follow the fail first principle "to succeed, try first where you are most likely to fail". The principle states that we should choose next the variable whose success probability is smallest, to minimize the expected branch length and thereby to minimize the search cost.

In interval-based algorithms, only Smear-based heuristics (and possibly largest first) follow the fail-first principle. They select the variable that, if instantiated, will most reduce the image of the functions. This would maximize the probability that at least one constraint is unsatisfied; therefore, the new subproblem would fail. In [3], we present new heuristics based on the Smear function to better understand its behavior.

Experiments related to finite CSPs conducted by Beck et al. [8] led them to believe that the effect of heuristics on the *branching factor* of the search tree (i.e., the number of children at each node) is also crucial for the performance of the algorithm. Well-performing heuristics in finite CSPs, such as dom [42] and Brélaz [16], select the variable that minimizes the branching factor in the node, i.e., the variable with the smallest domain. Other well-known heuristics combine both principles (failing early and reducing the branching factor), e.g., dom/deg [12] and dom/wdeg [15]. The branching factor is not used in interval-based algorithms because, due to the bisection procedure, there are at most two children in each node. However, we wonder whether there is a way to successfully adapt heuristics such as dom or dom/deg to intervals.

More sophisticated heuristics in finite domains, such as dom/wdeg, *look back* in the current search to detect the most important constraints: Each time a node is discarded, the weights of constraints directly related to this discard are augmented by 1. The criterion for selecting the next variable combines two criteria: the variable with the smallest domain and the variable that appears in the most weighted constraints. Another, more recent, look-back strategy is described in [88]. The authors propose selecting the *most active* variable during the propagation. The activity of a variable is computed during the search, and it is augmented each time some value of a variable domain is filtered. Several works also perform probing, i.e., a set of random searches to retrieve information from the problem (e.g., weight for constraints, activity of variables). This information is then used to select the order in which the variables should be selected [15,88,103].

Related to interval B&B, on the one hand, we think that a concept similar to the domain size of variables could be devised (for instance, the expected number of times a variable *needs to be* bisected). Then, heuristics minimizing the branching factor in finite CSPs (such as dom or Brélaz) could be easily adapted to intervals. On the other hand, once there is a high-quality and robust heuristic (e.g., the SmearSumRel heuristic), we think that the next step will be to incorporate some type of look-back strategy, similar to the ones related above.

7 Upper bounding: issues and perspectives

Upper bounding is a crucial component of NCOP solvers. Improving the current best found cost, UB, implies a reduction of the feasible space through the constraint $f_{obj}(x) \leq UB$. Thus, NCOP solvers should put considerable effort into finding solutions with near-optimal costs. However, once a quasi-optimal solution has been found, any effort spent in finding better solutions becomes useless.

In NCOP solvers, low-cost upper-bounding techniques are generally used. They range from simply trying the midpoint point of the box [96] to more sophisticated methods, such as extracting feasible linear regions to minimize a linearization of the objective function using Simplex [5]. In this direction (i.e., cheap techniques), well-known simple iterative methods, such as the gradient-descent [19] or Frank-Wolfe [36,52] methods, can help to find better solutions without much additional effort.

To validate a candidate solution x_c, interval optimizers perform a feasibility test. IBBA and Ibex-opt evaluate, using interval arithmetic, x_c in each function to validate the related constraints. Equality constraints $f_j(x) = 0$ are relaxed ($-\epsilon_{eq} \leq f_j(x) = 0 \leq \epsilon_{eq}$) to be validated. A few solvers work rigorously without relaxing equalities (e.g., GlobSol, Icos). They validate a tiny box, i.e., they guarantee that in a tiny box $\boldsymbol{x}_c$, there exists a solution for the original problem. An upper-bound cost of this solution is computed by evaluating $\boldsymbol{x}_c$ in the objective function with interval arithmetic. The incorporation of this type of certification technique is required in current state-of-the-art optimizers. In a recent work [60], Kearfott proposes an alternative technique that begins with an *approximately feasible point* (i.e., a point feasible in the relaxed system but unfeasible in the original one) and tries to make it feasible by perturbing it. The technique makes use of Gauss–Newton steps.

Global optimizers from the mathematical programming community generally resort to local minimization using sophisticated algorithms, e.g., Lagrangian relaxation-based methods in BARON [107] or the algorithms SNOPT [39] and CONOPT [29] in ANTIGONE [89]. We believe that new strategies should incorporate this type of sophisticated *but expensive* upper-bounding technique for finding solutions. Calling these methods in each node of the

search tree may be counterproductive because of their expense. Thus, to minimize the overhead, they should incorporate mechanisms for controlling the effort. For instance, they could apply an expensive local minimization technique *only* when a new upper bound has been found by the current techniques or only in the root node. Indicators (e.g., estimates of the proximity of the node to a minimizer point [77]) could also be used to decide whether a node should be exploited.

Consider the following experiment. An NCOP instance π can be transformed into an unfeasible NCSP instance π_u by adding the constraint $f_{obj}(x) < x^*$, where x^* is the lower bound of the optimal solution found by solving π. Solving π_u is almost equivalent (in time and the size of the tree search) to solving π using a super-fast upper-bounding method, which finds an ϵ-optimal solution in the first iteration of the NCOP algorithm. Comparing the CPU time spent for solving π and π_u, we can see the highest gain we could expect by improving the upper-bounding method.

8 Selection of the next box: issues and perspectives

When we address NCSPs, the selection of the next box to be treated is not a problem. Because the algorithm requires finding all solutions of the NCSP, it must treat all of the expanded boxes. Thus, to minimize the memory complexity, algorithms usually use a stack for storing the expanded nodes, performing a depth-first search (memory complexity O(1)). A different strategy is presented in [24]. The authors propose an interval-based B&B solver with the objective of finding potential solutions in different areas of the search space in the early stages of the exploration. Its algorithm first executes a classical depth-first search until a solution is found. Then, a new search strategy is used. The box maximizing the distance to the nearest solution found so far is selected next.

In contrast, the selection of the next box is crucial when we address NCOPs. We should next select the box that is most likely to improve the upper bound by the highest value, thus achieving the largest reduction of the feasible space (note the relation with the upper bounding). Thus, for the same reason that the upper bounding is useless once a near-optimal solution is found, the criteria for selecting the next box become irrelevant at that time, too.

One of the most used criteria is minLB. It consists of simply selecting the box with the lowest underestimation of the objective function ($lb = LB$). In the best case, we could find in this box a better new solution with cost lb, and the search would be finished (because $UB - LB = 0$). However, because interval methods compute large overestimations of images, this result is very improbable. Markot et al. [77] propose several interesting policies for selecting the next box. They range from the previously mentioned minLB to more sophisticated heuristics that try to select a box that (1) has an image under f and most likely contains a solution better than UB and (2) seems to have the largest feasible volume. Although these criteria have been tested only on a solver with no contractors (only a feasibility test is used to discard boxes), they show promising results.

KBFS [34] is a search strategy used in discrete-domain CSPs that generalizes the best-first search. In each iteration, KBFS expands the K best nodes instead of only 1, as the best-first strategy does. The authors show that KBFS is better than best-first when the heuristics for estimating the cost are inadmissible (otherwise, best-first is the best strategy) and have large errors. In this case, best first could fall into large subtrees with suboptimal or no solutions, whereas KBFS could escape from them and explore other possibilities. The KBFS strategy could be used in interval B&B because although minLB is an admissible heuristic, the fact

that we can update the UB in each node and thus the search space turns the best-first into a not-best strategy.

9 Conclusions

Interval B&B algorithms are commonly used to solve constraint satisfaction and global optimization problems. In this paper, two generic interval B&B solvers for treating these types of problems are described. They have important components in common: contractors and bisectors.

Newton-based contractors appear to be a good choice when the system is well constrained and the boxes are sufficiently small. Additionally, they are useful to certify the existence of a unique solution in the box. Relaxation-based contractors are not limited to well-constrained systems; however, the linear relaxations may lose important information related to the original problem. CP contractors are efficient for filtering larger boxes, working directly on the original (nonlinear) system of constraints. However, they have a reduced sight of the entire system; therefore, they suffer from the locality problem. Most recent interval-based libraries (e.g., Ibex [23], Icos [70]) allow one to combine these three approaches in one solver strategy, thus making the results more robust. Related to the bisection methods, only a few works have been proposed thus far, with the smear-based methods (in particular, the SmearSumRel heuristic) representing the most promising ones.

The dependency problem has been an obstacle to interval-based solvers from the very beginning. Although there is still work to do, we think that recent sophisticated CP contractors address the problem in a *satisfactory way*. We observed, however, several other issues that should be taken into account to improve the solver performance in the short term:

- Treating the locality problem with constraint propagation-oriented techniques, such as the linear combination of constraints and common subexpression elimination.
- Incorporating successful contraction techniques from non-rigorous algorithms, such as the reformulation linearization-technique, optimization-based reduction, convexification, and the use of constraints related to optimality conditions oriented to reduce the cluster effect. Most of these techniques must be adapted to intervals by making them mathematically rigorous.
- Incorporating adaptive mechanisms for selecting contractors. The idea here is to increase the contraction effort when it is most likely to filter domains.
- Incorporating look-back strategies to improve the variable selection heuristics.
- Improving the upper bounding techniques in optimization problems by incorporating more sophisticated (and possibly expensive) local search algorithms. To minimize the overhead, they should also incorporate mechanisms to control the effort.
- Improving the next box selection in optimization problems. For instance, combining intensification and diversification criteria in a KBFS-like [34] strategy.

Acknowledgments This work is supported by the Fondecyt Project 1120781.

References

1. Araya, I., Neveu, B., Trombettoni, G.: Exploiting common subexpressions in numerical CSPs. In: Principles and Practice of Constraint Programming (CP 2008), pp. 342–357. Springer (2008)

2. Araya, I., Neveu, B., Trombettoni, G.: An interval extension based on occurrence grouping. Computing **94**(2–4), 173–188 (2012)
3. Araya, I., Reyes, V., Oreallana, C.: More smear-based variable selection heuristics for NCSPs. In: International Conference on Tools with Artificial Intelligence (ICTAI 2013), pp. 1004–1011. IEEE (2013)
4. Araya, I., Trombettoni, G., Neveu, B.: A contractor based on convex interval Taylor. In: Proceedings of CPAIOR, LNCS 7298, pp. 1–16 (2012)
5. Araya, I., Trombettoni, G., Neveu, B., Chabert, G.: Upper bounding in inner regions for global optimization under inequality constraints. J. Glob. Optim. **60**, 145–164 (2014). doi:10.1007/s10898-014-0145-7
6. Araya, I., Trombettoni, G., Neveu, B., et al.: Exploiting monotonicity in interval constraint propagation. In: AAAI (2010)
7. Baharev, A., Achterberg, T., Rév, E.: Computation of an extractive distillation column with affine arithmetic. AIChE J. **55**(7), 1695–1704 (2009)
8. Beck, J.C., Prosser, P., Wallace, R.J.: Trying again to fail-first. In: Recent Advances in Constraints, pp. 41–55. Springer (2005)
9. Belotti, P.: Couenne, a users manual (2013). http://www.coin-or.org/Couenne/
10. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex MINLP. Optim. Methods Softw. **24**(4–5), 597–634 (2009)
11. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: International Conference on Logic Programming. Citeseer (1999)
12. Bessiere, C., Régin, J.C.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Principles and Practice of Constraint Programming (CP96), pp. 61–75. Springer (1996)
13. Bliek, C.: Computer methods for design automation. Ph.D. thesis, Massachusetts Institute of Technology (1992)
14. Bournez, O., Maler, O., Pnueli, A.: Orthogonal polyhedra: Representation and computation. In: Hybrid Systems: Computation and Control, pp. 46–60. Springer (1999)
15. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI, vol. 16, p. 146 (2004)
16. Brélaz, D.: New methods to color the vertices of a graph. Commun. ACM **22**(4), 251–256 (1979)
17. Carrizosa, E., Hansen, P., Messine, F.: Improving interval analysis bounds by translations. J. Glob. Optim. **29**(2), 157–172 (2004)
18. Casado, L., Martinez, J., García, I.: Experiments with a new selection criterion in a fast interval optimization algorithm. J. Glob. Optim. **19**(3), 247–264 (2001)
19. Cauchy, A.: Méthode générale pour la résolution des systemes déquations simultanées. C. R. Sci. Paris **25**(1847), 536–538 (1847)
20. Ceberio, M., Granvilliers, L.: Horner's rule for interval evaluation revisited. Computing **69**(1), 51–81 (2002)
21. Ceberio, M., Granvilliers, L.: Solving nonlinear equations by abstraction, Gaussian elimination, and interval methods. In: Frontiers of Combining Systems, pp. 117–131. Springer (2002)
22. Ceberio, M., Kreinovich, V.: Greedy algorithms for optimizing multivariate Horner schemes. ACM SIGSAM Bull. **38**(1), 8–15 (2004)
23. Chabert, G., Jaulin, L.: Contractor programming. Artif. Intell. **173**(11), 1079–1100 (2009)
24. Chenouard, R., Goldsztejn, A., Jermann, C., et al.: Search strategies for an anytime usage of the branch and prune algorithm. In: IJCAI, pp. 468–473 (2009)
25. Comba, J., Stolfi, J.: Affine arithmetic and its applications to computer graphics. In: Proceedings of SIBGRAPI'93—VI Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, pp. 9–18 (1993)
26. Csendes, T., Ratz, D.: Subdivision direction selection in interval methods for global optimization. SIAM J. Numer. Anal. **34**(3), 922–938 (1997)
27. De Figueiredo, L.H., Stolfi, J.: Affine arithmetic: concepts and applications. Numer. Algorithms **37**(1–4), 147–158 (2004)
28. Demidovitch, B., Maron, I., Polonski, V.: Eléments de calcul numérique. Mir (1973)
29. Drud, A.S.: CONOPT: a large-scale GRG code. ORSA J. Comput. **6**(2), 207–216 (1994)
30. Du, K., Kearfott, R.B.: The cluster problem in multivariate global optimization. J. Glob. Optim. **5**(3), 253–265 (1994)
31. Duran, M.A., Grossmann, I.E.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Math. Program. **36**(3), 307–339 (1986)
32. Eldon, H., William, W.: Global optimization using interval analysis (1992)
33. Faltings, B.V., Lottaz, C., et al.: Collaborative design using solution spaces (2000)

34. Felner, A., Kraus, S., Korf, R.E.: KBFS: K-best-first search. Ann. Math. Artif. Intell. **39**(1–2), 19–39 (2003)
35. Floudas, C.A., Pardalos, P.M.: Encyclopedia of Optimization, vol. 1. Springer Science & Business Media, Berlin (2008)
36. Frank, M., Wolfe, P.: An algorithm for quadratic programming. Nav. Res. Logist. Q. **3**(1–2), 95–110 (1956)
37. Frommer, A., Lang, B.: Existence tests for solutions of nonlinear equations using Borsuk's theorem. SIAM J. Numer. Anal. **43**(3), 1348–1361 (2005)
38. Fünfzig, C., Michelucci, D., Foufou, S.: Nonlinear systems solver in floating-point arithmetic using LP reduction. In: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling, pp. 123–134. ACM (2009)
39. Gill, P.E., Murray, W., Saunders, M.A.: SNOPT: an SQP algorithm for large-scale constrained optimization. SIAM J. Optim. **12**(4), 979–1006 (2002)
40. Goldsztejn, A., Granvilliers, L.: A new framework for sharp and efficient resolution of NCSP with manifolds of solutions. Constraints **15**(2), 190–212 (2010)
41. Goldsztejn, A., Lebbah, Y., Michel, C., Rueher, M.: Revisiting the upper bounding process in a safe branch and bound algorithm. In: Principles and Practice of Constraint Programming (CP 2008), pp. 598–602. Springer (2008)
42. Golomb, S.W., Baumert, L.D.: Backtrack programming. J. ACM (JACM) **12**(4), 516–524 (1965)
43. Granvilliers, L.: Adaptive bisection of numerical CSPs. In: Principles and Practice of Constraint Programming (2012), pp. 290–298. Springer (2012)
44. Granvilliers, L., Goldsztejn, A.: A branch-and-bound algorithm for unconstrained global optimization. In: Proceedings of the 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN) (2010)
45. Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: Numerical toolbox for verified computing I (1993)
46. Hansen, E.: Interval arithmetic in matrix computations, Part I. J Soc Ind. Appl. Math. Ser. B Numer. Anal. **2**(2), 308–320 (1965)
47. Hansen, E.: Global optimization using interval analysis: the multi-dimensional case. Numer. Math. **34**(3), 247–270 (1980)
48. Hansen, E.: Bounding the solution of interval linear equations. SIAM J. Numer. Anal. **29**(5), 1493–1503 (1992)
49. Hansen, E.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (1992)
50. Hansen, E., Walster, G.W.: Global Optimization Using Interval Analysis: Revised and Expanded, vol. 264. CRC Press, Boca Raton (2003)
51. Ishii, D., Goldsztejn, A., Jermann, C.: Interval-based projection method for under-constrained numerical systems. Constraints **17**(4), 432–460 (2012)
52. Jaggi, M.: Revisiting Frank–Wolfe: projection-free sparse convex optimization. In: Proceedings of the 30th International Conference on Machine Learning (ICML-13), pp. 427–435 (2013)
53. Jaulin, L.: Localization of an underwater robot using interval constraint propagation. In: Principles and Practice of Constraint Programming (CP 2006), pp. 244–255. Springer (2006)
54. John, F.: Extremum problems with inequalities as subsidiary conditions. In: Studies and Essays Presented to R. Courant on his 60th Birthday (Jan. 8, 1948), pp. 187–204. Interscience, New York (1948)
55. Karush, W.: Minima of functions of several variables with inequalities as side constraints. Ph.D. thesis, Masters thesis, Dept. of Mathematics, University of Chicago (1939)
56. Kearfott, R.B.: Preconditioners for the interval Gauss–Seidel method. SIAM J. Numer. Anal. **27**(3), 804–822 (1990)
57. Kearfott, R.B.: An interval branch and bound algorithm for bound constrained optimization problems. J. Glob. Optim. **2**(3), 259–280 (1992)
58. Kearfott, R.B.: Discussion and empirical comparisons of linear relaxations and alternate techniques in validated deterministic global optimization. Optim. Methods Softw. **21**(5), 715–731 (2006)
59. Kearfott, R.B.: GlobSol user guide. Optim. Methods Softw. **24**(4–5), 687–708 (2009)
60. Kearfott, R.B.: On rigorous upper bounds to a global optimum. J. Glob. Optim. **59**(2–3), 459–476 (2014)
61. Kearfott, R.B., Hongthong, S.: Validated linear relaxations and preprocessing: some experiments. SIAM J. Optim. **16**(2), 418–433 (2005)
62. Kearfott, R.B., Novoa III, M.: Algorithm 681: INTBIS, a portable interval Newton/bisection package. ACM Trans. Math. Softw. (TOMS) **16**(2), 152–157 (1990)
63. Kearfott, R.B., Walster, G.W.: Symbolic preconditioning with Taylor models: some examples. Reliab. Comput. **8**(6), 453–468 (2002)
64. Kieffer, M.: Distributed bounded-error parameter and state estimation in networks of sensors. In: Numerical Validation in Current Hardware Architectures, pp. 189–202. Springer (2009)

65. Kieffer, M., Walter, E.: Centralized and distributed source localization by a network of sensors using guaranteed set estimation. In: 2006 IEEE International Conference on Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings, vol. 4, pp. IV–IV. IEEE (2006)
66. Kolev, L.V.: Use of interval slopes for the irrational part of factorable functions. Reliab. Comput. **3**(1), 83–93 (1997)
67. Krawczyk, R.: Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken. Computing **4**(3), 187–201 (1969)
68. Kueviakoe, I., Lambert, A., Tarroux, P.: Comparison of interval constraint propagation algorithms for vehicle localization. J. Softw. Eng. Appl. **5**, 157 (2013)
69. Lagouanelle, J.L., Soubry, G.: Optimal multisections in interval branch-and-bound methods of global optimization. J. Glob. Optim. **30**(1), 23–38 (2004)
70. Lebbah, Y.: Icos: a branch and bound based solver for rigorous global optimization. Optim. Methods Softw. **24**(4–5), 709–726 (2009)
71. Lebbah, Y., Michel, C., Rueher, M.: An efficient and safe framework for solving optimization problems. J. Comput. Appl. Math. **199**(2), 372–377 (2007)
72. Lhomme, O.: Consistency techniques for numeric CSPs. In: IJCAI, vol. 93, pp. 232–238. Citeseer (1993)
73. Liberti, L.: Writing global optimization software. In: Global Optimization, pp. 211–262. Springer (2006)
74. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**(1), 99–118 (1977)
75. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **4**(4), 379–456 (2003)
76. Maranas, C.D., Floudas, C.A.: Finding all solutions of nonlinearly constrained systems of equations. J. Glob. Optim. **7**(2), 143–182 (1995)
77. Markót, M.C., Fernandez, J., Casado, L.G., Csendes, T.: New interval methods for constrained global optimization. Math. Program. **106**(2), 287–318 (2006)
78. Markót, M.C., Schichl, H.: Bound constrained interval global optimization in the COCONUT environment. J. Glob. Optim. **60**(4), 751–776 (2014)
79. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: part I—convex underestimating problems. Math. Program. **10**(1), 147–175 (1976)
80. Merlet, J.P.: Optimal design for the micro parallel robot MIPS. In: IEEE International Conference on Robotics and Automation, 2002. Proceedings of ICRA'02, vol. 2, pp. 1149–1154. IEEE (2002)
81. Merlet, J.P.: Interval analysis for certified numerical solution of problems in robotics. Int. J. Appl. Math. Comput. Sci. **19**(3), 399–412 (2009)
82. Messine, F.: Extentions of affine arithmetic: application to unconstrained global optimization. J. Univers. Comput. Sci. **8**(11), 992–1015 (2002)
83. Messine, F.: Deterministic global optimization using interval constraint propagation techniques. RAIRO Oper. Res. **38**(04), 277–293 (2004)
84. Messine, F.: A deterministic global optimization algorithm for design problems. In: Essays and Surveys in Global Optimization, pp. 267–294. Springer (2005)
85. Messine, F., Nogarede, B., Lagouanelle, J.L.: Optimal design of electromechanical actuators: a new method based on global optimization. IEEE Trans. Magn. **34**(1), 299–308 (1998)
86. Messine, F., Touhami, A.: A general reliable quadratic form: an extension of affine arithmetic. Reliab. Comput. **12**(3), 171–192 (2006)
87. Meyer, C.A., Floudas, C.A.: Convex envelopes for edge-concave functions. Math. Program. **103**(2), 207–224 (2005)
88. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pp. 228–243. Springer (2012)
89. Misener, R., Floudas, C.A.: ANTIGONE: algorithms for continuous/integer global optimization of nonlinear equations. J. Glob. Optim. **59**(2–3), 503–526 (2013)
90. Moore, R.: Interval Analysis, vol. 60 (1966)
91. Mourad, F., Snoussi, H., Abdallah, F., Richard, C.: Anchor-based localization via interval analysis for mobile ad-hoc sensor networks. IEEE Trans. Signal Process. **57**(8), 3226–3239 (2009)
92. Nataraj, P., Patil, M.D.: Reliable and robust automated synthesis of QFT controller for nonlinear magnetic levitation system using interval constraint satisfaction techniques. In: Constraint Programming and Decision Making, pp. 131–135. Springer (2014)
93. Nataraj, P., Tharewal, S.: An interval analysis algorithm for automated controller synthesis in QFT designs. J. Dyn. Syst. Meas. Contr. **129**(3), 311–321 (2007)
94. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer linear programming. Math. Program. **99**(2), 283–296 (2004)

95. Neveu, B., Trombettoni, G., et al.: Adaptive constructive interval disjunction. In: International Conference on Tools with Artificial Intelligence (ICTAI), pp. 900–906 (2013)
96. Ninin, J., Messine, F., Hansen, P.: A reliable affine relaxation method for global optimization. 4OR (2014). doi:10.1007/s10288-014-0269-0
97. Ortega, J.M., Rheinboldt, W.C.: Iterative Solution of Nonlinear Equations in Several Variables, vol. 30. Siam, Philadelphia (2000)
98. Patil, M.D., Nataraj, P.: QFT prefilter design for multivariable systems using interval constraint satisfaction technique. J. Control Theory Appl. **11**(4), 529–537 (2013)
99. Ramdani, N., Meslem, N., Candau, Y.: Reachability of uncertain nonlinear systems using a nonlinear hybridization. In: Hybrid Systems: Computation and Control, pp. 415–428. Springer, Berlin (2008)
100. Ramdani, N., Nedialkov, N.S.: Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques. Nonlinear Anal. Hybrid Syst. **5**(2), 149–162 (2011)
101. Ratschek, H., Rokne, J.: New Computer Methods for Global Optimization. Horwood, Chichester (1988)
102. Ratz, D.: Automatische ergebnisveri kation bei globalen optimierungsproblemen. Ph.D. thesis, Dissertation, Universit at Karlsruhe (1992)
103. Refalo, P.: Impact-based search strategies for constraint programming. In: Principles and Practice of Constraint Programming (CP 2004), pp. 557–571. Springer (2004)
104. Reynet, O., Voisin, O., Jaulin, L.: Anchor-based localization using distributed interval contractors (2011)
105. Roy, J.M.: Singularities in Deterministic Global Optimization. University of Louisiana at Lafayette (2010)
106. Ryoo, H.S., Sahinidis, N.V.: A branch-and-reduce approach to global optimization. J. Glob. Optim. **8**(2), 107–138 (1996)
107. Sahinidis, N.V.: BARON: a general purpose global optimization software package. J. Glob. Optim. **8**(2), 201–205 (1996)
108. Sam-Haroud, D., Faltings, B.: Consistency techniques for continuous constraints. Constraints **1**(1–2), 85–118 (1996)
109. Schichl, H., Neumaier, A.: Exclusion regions for systems of equations. SIAM J. Numer. Anal. **42**(1), 383–408 (2004)
110. Schichl, H., Neumaier, A.: Interval analysis on directed acyclic graphs for global optimization. J. Glob. Optim. **33**(4), 541–562 (2005)
111. Shectman, J.P., Sahinidis, N.V.: A finite algorithm for global minimization of separable concave programs. J. Glob. Optim. **12**(1), 1–36 (1998)
112. Smith, B.M., Grant, S.A.: Trying harder to fail first. Research report series/University of Leeds, School of Computer Studies LU SCS RR (1997)
113. Soares, R.D.P.: Finding all real solutions of nonlinear systems of equations with discontinuities by a modified affine arithmetic. Comput. Chem. Eng. **48**, 48–57 (2013)
114. Stamatatos, E., Stergiou, K.: Learning how to propagate using random probing. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 263–278. Springer (2009)
115. Stergiou, K.: Heuristics for dynamically adapting propagation in constraint satisfaction problems. AI Commun. **22**(3), 125–141 (2009)
116. Tapia, R.: The Kantorovich theorem for Newton's method. Am. Math. Mon. **78**(4), 389–392 (1971)
117. Tawarmalani, M., Sahinidis, N.V.: Convexification and Global Optimization in Continuous and Mixed-integer Nonlinear Programming: Theory, Algorithms, Software, and Applications, vol. 65. Springer, Berlin (2002)
118. Trombettoni, G., Araya, I., Neveu, B., Chabert, G.: Inner regions and interval linearizations for global optimization. In: AAAI, pp. 99–104 (2011)
119. Trombettoni, G., Chabert, G.: Constructive interval disjunction. In: Principles and Practice of Constraint Programming (CP 2007), pp. 635–650. Springer (2007)
120. Van Hentenryck, P., Michel, L., Deville, Y.: Numerica: A Modeling Language for Global Optimization. MIT Press, Cambridge (2003)
121. Vu, X.H., Sam-Haroud, D., Faltings, B.: Combining multiple inclusion representations in numerical constraint propagation. In: International Conference on Tools with Artificial Intelligence (ICTAI 2004), pp. 458–467. IEEE (2004)
122. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Approximation techniques for non-linear problems with continuum of solutions. In: Abstraction, Reformulation, and Approximation, pp. 224–241. Springer (2002)
123. Vu, X.H., Schichl, H., Sam-Haroud, D.: Using directed acyclic graphs to coordinate propagation and search for numerical constraint satisfaction problems. In: International Conference on Tools with Artificial Intelligence (ICTAI 2004), pp. 72–81. IEEE (2004)

124. Yamamura, K., Kawata, H., Tokue, A.: Interval solution of nonlinear equations using linear programming. BIT Numer. Math. **38**(1), 186–199 (1998)
125. Yannou, B., Simpson, T.W., Barton, R.R.: Towards a conceptual design explorer using metamodeling approaches and constraint programming. In: ASME 2003 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, pp. 605–614. American Society of Mechanical Engineers (2003)

J Glob Optim
DOI 10.1007/s10898-017-0569-y

`lsmear`: a variable selection strategy for interval branch and bound solvers

Ignacio Araya[1] · **Bertrand Neveu**[2]

Received: 30 January 2017 / Accepted: 6 September 2017

Abstract Smear-based variable selection strategies are well-known and commonly used by branch-and-prune interval-based solvers. They estimate the impact of the variables on each constraint of the system by using the partial derivatives and the sizes of the variable domains. Then they aggregate these values, in some way, to estimate the impact of each variable on the whole system. The variable with the greatest impact is then selected. A problem of these strategies is that they, generally, consider all constraints equally important. In this work, we propose a new variable selection strategy which first weights the constraints by using the optimal Lagrangian multipliers of a linearization of the original problem. Then, the impact of the variables is computed with a typical smear-based function but taking into account the weights of the constraints. The strategy isg tested on a set of well-known benchmark instances outperforming significantly the classical variable selection strategies.

Keywords Branch and bound · Interval-based solver · Variable selection · Lagrangian multipliers

1 Introduction

This paper deals with continuous global optimization (nonlinear programming) deterministically handled by interval branch and bound (B&B). The problem is defined by:

$$\min_{x \in \boldsymbol{x}} f(x) s.t. g(x) \leq 0 \tag{1}$$

✉ Ignacio Araya
ignacio.araya@pucv.cl

Bertrand Neveu
Bertrand.Neveu@enpc.fr

[1] Pontificia Universidad Católica de Valparaíso, Escuela de Ing. Informática, 2950 Avenida Brasil, V región, Chile

[2] Imagine LIGM Université Paris–Est, Paris, France

Published online: 12 September 2017

where $f : \mathbb{R}^n \to \mathbb{R}$ is the real-valued objective (non convex) function and $g : \mathbb{R}^n \to \mathbb{R}^m$ is a vector-valued (non convex) function.[1] $x = (x_1, ..., x_i, ...x_n)$ is a vector of variables varying in a domain (i.e., a box) $\boldsymbol{x}$.[2]

Several works have been proposed for finding good branching strategies [1–10]. Most of them use local information (provided by the current node) in order to select the next variable to bisect. Bisection consists of splitting the current box into two boxes generating two subproblems to solve. `round-robin` is one of the simplest methods. It does not require any information about the system. If the current n-dimensional box $\boldsymbol{x}$ was obtained by bisecting the interval $\boldsymbol{x}_i$, then the `round-robin` method will select the variable $x_{i'}$, with $i' = (i + 1)\ mod\ n$. The objective is to refrain from neglecting any variable. One weakness of this strategy is that a *bad* initial ordering of variables can lead to disastrous performance. `largest-first` [8] (also known as the interval width-oriented rule [2]) simply selects the variable with the largest domain. It is based on the assumption that intervals with large diameters have a greater impact on the function image. In the same manner as the `round-robin` technique, the `largest-first` method does not omit any variable. Smear-based methods [4,10] use information about the system to obtain the variable with the greatest *impact*. The impact of a variable x_i on a function g_j is computed by means of the *smear value*. Consider that the current node is associated with box $\boldsymbol{x}$; the smear value is given by

$$\text{smear}(x_i, g_j) = |\boldsymbol{a}_{ji}| * \text{wid}(\boldsymbol{x}_i)$$

where $\boldsymbol{a}_{ji}$ is an interval overestimate of the range of the partial derivative $\frac{\partial g_j}{\partial x_i}$ in $\boldsymbol{x}$[3]. $|\boldsymbol{a}_{ji}|$ is the *magnitude* of the interval $\boldsymbol{a}_{ji}$, i.e., $|\boldsymbol{a}_{ji}| = \max(|\underline{a_{ji}}|, |\overline{a_{ji}}|)$.

Theoretically, the smear value of a variable x_i related to a function g_j estimates (using a first-order Taylor approximation of g_j) the reduction of the image size of g_j consequent upon a reduction of the domain size of x_i [1].

Selection methods based on the smear value select the variable that maximizes an aggregation of this value in the whole system. For instance, the strategy `smearmax` [10] selects the variable x_i which maximizes $\max_{j=1..m}(\text{smear}(x_i, g_j))$. The strategy `smearsum` selects the variable x_i which maximizes $\sum_{j=1}^{m}(\text{smear}(x_i, g_j))$. A variant proposed in [1,6] uses a *relative smear value* instead:

$$\text{smear}_{rel}(x_i, g_j) = \frac{\text{smear}(x_i, g_j)}{\sum_{k=1}^{n} \text{smear}(x_k, g_j)}$$

Then, the `smearsumrel` strategy selects the variable x_i, which maximizes $\sum_{j=1}^{n}(\text{smear}_{rel}(x_i, g_j))$. Generally, smear-based strategies also include the objective function in the vector g for computing the impact of variables.

Tawarmalani and Sahinidis [11] present an algorithm called *ViolationTransfer*, to estimate the impact of a variable on the problem. ViolationTransfer works with the Lagrangian function of a relaxation of the problem and an optimal solution of the relaxation $x^\star$. For each variable, an interval $\boldsymbol{x}_i^v \subset \boldsymbol{x}_i$ is defined. $\boldsymbol{x}^v$ is the smallest box such that it contains $x^\star$ and each univariate constraint $g_j(x_1^\star, ..., x_{i-1}^\star, x_i, x_{i+1}^\star, x_n^\star) \leq 0\ (j = 1..m)$ is feasible for at least one value in $\boldsymbol{x}_i^v$.

[1] The branching strategies proposed in this paper can also apply to problems having equality constraints.

[2] An interval $\boldsymbol{x}_i = [\underline{x_i}, \overline{x_i}]$ defines the set of reals x_i s.t. $\underline{x_i} \leq x_i \leq \overline{x_i}$. A *box* $\boldsymbol{x}$ is a Cartesian product of intervals $\boldsymbol{x}_1 \times ... \times \boldsymbol{x}_i \times ... \times \boldsymbol{x}_n$.

[3] The range of partial derivatives can be computed by using an automatic differentiation method [10].

Then, for each variable x_i, the difference between the bounds of the image of the Lagrangian function over the interval $\boldsymbol{x}_i^v$ is estimated. In each estimation all the variables are fixed except x_i. The assumption is that branching on the variable maximizing the image width is likely to improve the lower bound of the objective function in the subproblems. If we consider a linear relaxation: $\min \sum_{i=1}^{n} c_i x_i$, subject to $\sum_{i=1}^{n} a_{ji} x_i \leq 0$ for all $j \in \{1, ..., m'\}$, an estimate of the change of the Lagrangian function related to the variable x_i is given by:

$$(\overline{x_i^v} - \underline{x_i^v}). \left| c_i + \sum_{j=1}^{m'} \lambda_j^* a_{ji} \right|$$

where λ^* is the *optimal dual solution* to the linear relaxation. ViolationTransfer selects the variable maximizing this value.

In this article we propose `lsmear`, a new variable selection strategy for interval B&B solvers. In a few words, the method selects the variable maximizing the smear value of the Lagrangian function of the problem. In the Lagrangian function, the Lagrange multipliers are replaced by the dual optimal of a linear approximation of the problem. Related to the ViolationTransfer strategy our approach has some important differences:

1. `lsmear` uses a simple *linear approximation* of the original problem instead of sophisticated convex relaxation techniques.
2. `lsmear` estimates the impact of each variable in the Lagrangian function of the *original problem*. Instead, ViolationTransfer estimates the impact of each variable in the Lagrangian function of a *linear relaxation*.
3. The computation of $\boldsymbol{x}^v$ requires the solver to use a reformulated problem in which multidimensional functions are replaced with either univariate or bilinear functions [11]. For the moment, and in order to maintain the simplicity and generality of the approach, `lsmear` uses directly $\boldsymbol{x}$ instead of $\boldsymbol{x}^v$.

In Sect. 2 we present the necessary background. Section 3 gives a brief introduction about the Lagrange function and the optimal Lagrange multipliers. In Sect. 4 we explain our approach and some variants. Section 5 carries out an empirical study of different variants of the proposed strategy. Finally, conclusions are given in Sect. 6.

2 Background and the interval solver

a An **interval** $\boldsymbol{x}_i = [\underline{x}_i, \overline{x}_i]$ defines the set of reals x_i s.t. $\underline{x}_i \leq x_i \leq \overline{x}_i$, where $\underline{x}_i$ and $\overline{x}_i$ are floating-point numbers. $\mathbb{IR}$ denotes the set of all intervals. The size or *width* of $\boldsymbol{x}_i$ is $\text{wid}(\boldsymbol{x}_i) = \overline{x}_i - \underline{x}_i$. $\text{mid}(\boldsymbol{x}_i)$ denotes the midpoint of $\boldsymbol{x}_i$. A **box** $\boldsymbol{x} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)$ determines the Cartesian product of intervals $\boldsymbol{x}_1 \times ... \times \boldsymbol{x}_n$. The *size* of a box is $\text{wid}(\boldsymbol{x}) = \max_{x_i \in x} \text{wid}(\boldsymbol{x}_i)$.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is *factorable* if it can be computed in a finite number of simple steps, starting with model variables and real constants, using elementary unary and binary operators. Factorable functions are extended to intervals. A function $\boldsymbol{f} : \mathbb{IR}^n \to \mathbb{IR}$ is an **interval extension** of a real function $f : \mathbb{R}^n \to \mathbb{R}$ if it computes an enclosure of the image of f over any box $\boldsymbol{x}$, i.e., $\forall \boldsymbol{x} \in \mathbb{IR}^n$, $\boldsymbol{f}(\boldsymbol{x}) \supseteq \{f(x), x \in \boldsymbol{x}\}$. Interval arithmetic defines the extension of the classic elementary operators (e.g., $+, -, \cdot, /$, power, exp, log, sin, cos,...). For instance, $[a, b] + [c, d] = [a + b, c + d]$, $[log]([a, b]) = [log(a), log(b)]$, etc. The **natural extension** of a factorable function f corresponds to the mapping of f to intervals using the corresponding interval operators.

A continuous (or numerical) constrained global optimization problem (NCOP) is defined as follows.

Definition 1 (*Numerical constrained global optimization*) Consider a vector of variables $x = (x_1, ..., x_i, ...x_n)$ varying in a box $\boldsymbol{x}$, a real-valued factorable function $f : \mathbb{R}^n \to \mathbb{R}$, and a vector-valued factorable function $g : \mathbb{R}^n \to \mathbb{R}^m$.

Given the system $S = (f, g, \boldsymbol{x})$, the numerical constrained global optimization problem consists in finding:

$$\min_{x \in \boldsymbol{x}} f(x) \;\; subject\ to \;\; g(x) \leq 0$$

f denotes the **objective function**; $g(x) \leq 0$ corresponds to the set of **constraints**. x is said to be **feasible** if it satisfies the constraints. x^* denotes the **optimal solution** of the problem and $f(x^*)$ corresponds to the optimal value.

An **active constraint** at a feasible point x' is a constraint $g_j(x) \leq 0$ such that $g_j(x') = 0$. An **inactive constraint** at a point x' is a constraint $g_j(x) \leq 0$ such that $g_j(x') < 0$. Active constraints restrict the domain of feasibility in neighborhoods of x', while inactive constraints have no influence on these neighborhoods. Thus, inactive constraints at the optimal solution x^* could be discarded from the system without altering either the optimal solution or the optimal value of the problem.

The constraints defined in an NCOP are numerical. They are equations and inequalities using mathematical operators like $+, \cdot, /, exp, log$ and sin.

The standard interval NCOP solver

Interval solvers for NCOPs generally find one **ϵ-optimal** solution, i.e., one feasible point x' such that $f(x') - f(x^*) \leq \epsilon$, where ϵ corresponds to the desired precision of the method.[4] Algorithm 1 shows the basic structure of an interval branch and bound solver. For simplicity we consider that f, g and x are global variables. Additionally, a variable x_o, with initial domain $\boldsymbol{x}_o = [-\infty, +\infty]$ is included in the set of variables x and a new constraint $x_o = f(x)$ is included in the set of constraints (actually, functions $f(x) - x_o$ and $x_o - f(x)$ are included in g). In this way, $\boldsymbol{x}_o$ keeps updated the bounds of f in the box.

Starting from an initial box $\boldsymbol{x}$, the solver builds a search tree by following a branch & bound schema. In each iteration a box from the list L of remaining boxes is selected and processed (first line of the while-loop). Generally, the box minimizing a lower bound of the objective function is selected [12–14]. A sophisticated method called `FeasibleDiving` [15] selects the next box in two different ways. The first way, called *diving*, does a depth-first-like search until a leaf of the tree is reached. When a leaf is found, the strategy swaps and selects the box minimizing its lower bound. Then, another diving is performed and so on.

Then, nodes are treated by two methods: bisection and filtering. The *bisection* consists in splitting the current box into two (or more) sub-boxes by dividing one (or more) interval(s) in the middle.

New boxes are then treated by the *filtering* method one by one. This procedure is composed by one or several contraction methods or *contractors*. Contractors attempt to filter the boxes by eliminating inconsistent values from within their bounds without loss of solutions. If all the values in a box are filtered, an empty box is returned. The filtering procedure is also

[4] Due to the relative precision of floating point numbers, when the optimal cost is too large a relative precision is required, i.e., $\frac{f(x') - f(x^*)}{|f(x')|} \leq \epsilon$.

responsible for computing the lower bound of the box. This is achieved by filtering the domain of the variable x_o. Thus, after filtering, a new lower bound of the objective function in the box is given by $\underline{x}_o$. Several contractors have been proposed so far, they include contractors based on constraint propagation [7,16–19] and contractors based on linear relaxations [20–22].

Data: f, g, x
procedure `Solver` $(\boldsymbol{x}, \epsilon)$; **out:** x_{UB}, UB
  $L \leftarrow \boldsymbol{x}$; $UB \leftarrow +\infty$; $LB \leftarrow -\infty$; ;
  while $L \neq \emptyset$ **and** $UB - LB > \epsilon$ **and** $\frac{UB-LB}{|UB|} > \epsilon$ **do**
    $\boldsymbol{x} \leftarrow$ select a box from L;
    $(\boldsymbol{x}^l, \boldsymbol{x}^r) \leftarrow$ `bisect`$(\boldsymbol{x})$;
    for $\boldsymbol{x} \in \{\boldsymbol{x}^l, \boldsymbol{x}^r\}$ **do**
      $\boldsymbol{x} \leftarrow$ `filtering`$(\boldsymbol{x})$;
      if $\boldsymbol{x} \neq \emptyset$ **then**
        `upper-bounding`$(\boldsymbol{x}, UB, x_{UB})$;
        `insert`$(L, \boldsymbol{x})$;
      end
    end
    $LB \leftarrow \min_{\boldsymbol{x} \in L} \underline{x}_o$;
  end
end.

Algorithm 1: Algorithm NCOP-solver

The *upper bounding* consists in finding feasible solutions in $\boldsymbol{x}$ with costs lower than the current UB. If a best solution is found, the procedure updates UB and the best found solution x_{UB}. Updating UB implies a *global* reduction of the feasible space through the constraint $f(x) \leq UB$. This constraint is considered by the filtering procedure. Before filtering, the procedure updates the upper bound of the variable x_o in the box: $\boldsymbol{x}_o \leftarrow \boldsymbol{x}_o \cap [-\infty, UB]$. To find feasible points, generally local search methods are used (e.g., selecting the midpoint of the box [13], applying the Newton method [12,23], extracting convex inner regions [14]), plus some feasibility tests (e.g., the natural evaluation of the candidate point [13,14], the Borsuk proof [12], etc.). Then, $\boldsymbol{x}$ inserted into the list L.

LB is the current computed lower bound of the objective function. It corresponds to the minimum value of the lower bounds for all the leaf boxes in the tree. The search concludes when one of the following termination criteria is reached:

- If $UB - LB \leq \epsilon$ or $\frac{UB-LB}{|UB|} \leq \epsilon$, the solver returns an ϵ-optimum solution of the problem x_{UB} and its cost UB. Note that if some boxes still remain in L, the lower bound of these boxes should necessarily be equal or greater than $UB - \epsilon$ or $UB - |UB|.\epsilon$.
- If $L = \emptyset$ and no feasible solution was found, then the problem is unfeasible.

3 Optimal Lagrange multipliers

In this section we give an introduction to the *Lagrangian* and the so-called *optimal Lagrange multipliers* which will be used by our approach to weight the impact of constraints in the optimization problems.

The Lagrangian $L : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}$ associated to an optimization problem $\min_{x \in \boldsymbol{x}} f(x)\ s.t.\ g(x) \leq 0$ (or *primal problem*) is defined by:

$$L(x, \lambda) = f(x) + \sum_{j=1}^{m} \lambda_j g_j(x)$$

where λ_j is the *Lagrange multiplier* associated to the j-th inequality constraint $g_j(x) \leq 0$.

The Lagrangian basically augments the objective function with a weighted sum of the constraint functions. The *Lagrange dual function* $h : \mathbb{R}^m \rightarrow \mathbb{R}$ is defined by:

$$h(\lambda) = \inf_{x \in \boldsymbol{x}} L(x, \lambda)$$

An important property of the dual function is that for any value of $\lambda \geq 0$: $h(\lambda) \leq f(x^*)$. Finally, the *Lagrange dual problem* associated with the problem (1) is defined by:

$$\max_{\lambda} h(\lambda)\ subject\ to\ \lambda \geq 0$$

λ^* denotes the dual optimal solution of the problem and its scalar components are called **optimal Lagrange multipliers**. As seen from function L, the Lagrange multipliers are associated to each constraint of the system. Thus, in a way, the optimal Lagrange multiplier related to any constraint estimates the rate of change in the optimal value, consequent upon changes in the constraint function. In particular, the value of the optimal Lagrange multipliers is 0 for inactive constraints of the problem.

Consider the following linear program in inequality form, i.e.,

$$\min_{x} c^T x\ subject\ to\ Ax \leq b$$

where A is the coefficient matrix, b and c are vectors of coefficients and c^T is the transpose of vector c. The dual problem associated to the primal one can be formulated in the following way:

$$\max_{\lambda} -b^T \lambda\ subject\ to\ A^T \lambda + c = 0, \lambda \geq 0$$

4 `lsmear`, a smear-based strategy using optimal Lagrange multipliers

A main issue related to the smear-based strategies is that these strategies do not weight the importance of each constraint, i.e., they consider all the constraints equally important (even between active and inactive constraints).

On the other hand, as noted in the previous section, if constraints of the Lagrangian function of the problem are weighted by means of the optimal Lagrange multipliers, the problem of minimizing the Lagrangian function can be seen as an approximation of the original optimization problem.

Thus, `lsmear`, our proposed algorithm, is basically a smear-based strategy which uses an *estimation* of the optimal Lagrange multipliers for weighting the constraints. The algorithm works in two phases: first, a linearization of the global optimization problem is generated. Each function $g_j(x)$ is approximated by using the first order term of its Taylor expansion around the midpoint of the box, i.e.,

$$gl_j(x) = g_j(\text{mid}(\boldsymbol{x})) + \sum_{i=1}^{n} \text{mid}(\boldsymbol{J}_{ji}).(x_i - \text{mid}(\boldsymbol{x}_i))$$

where $\boldsymbol{J}_{ji}$ is an interval overestimation of the image of $\frac{\partial g_j}{\partial x_i}$ over $\boldsymbol{x}$ computed by using automatic differentiation [10]. Note that instead of using the partial derivatives in the midpoint of the box we use the midpoint of the overestimation of the partial derivatives (i.e., $\text{mid}(\boldsymbol{J}_{ji})$).

The generated linear optimization problem includes the bound constraints, i.e., $\underline{x_i} \leq \boldsymbol{x}_i \leq \overline{x_i}$, and it is solved by using the simplex method.[5] If the linear program does not have solutions or if the optimal value is unbounded, then the `smearsum` strategy simply selects the next variable to bisect. Otherwise, if an optimum exists, then a second phase is carried out. In this phase, the strategy computes the smear values of the variables on the following function:

$$L(x) = x_o + \sum_{i=1}^{m} \lambda_j^* g_j(x)$$

where λ^* corresponds to the dual optimal solution of the linear problem. The function L is equivalent to the Lagrangian of the original problem but the Lagrange multipliers have been replaced by the optimal Lagrange multipliers of the linear approximation. Finally, `lsmear` selects and bisects the variable that maximizes its smear value.

An interesting thing about L is that it approximates the original problem by only one objective function. When there is no constraint, there is no disagreement between different smear-based methods either: they all should select the same variable that maximizes the smear value.

procedure *lsmear*($\boldsymbol{x}$, $\boldsymbol{J}$, g); **out:** *var*
  /* Phase 1: linearization and solving the linear program*/ ;
  $gl \leftarrow g(\text{mid}(\boldsymbol{x})) + \text{mid}(\boldsymbol{J}).(x - \text{mid}(\boldsymbol{x}))$;
  $\boldsymbol{t} \leftarrow x_o$; $\boldsymbol{x}_o \leftarrow [-\infty, \infty]$;
  $\lambda^* \leftarrow \text{optimize}(\min x_o,\ \text{subject to: } \underline{x_i} \leq x_i \leq \overline{x_i},\ gl_j(x) \leq 0)$;
  $\boldsymbol{x}_o \leftarrow \boldsymbol{t}$;
  if $\lambda^* \neq \varnothing$ **then**
    /* Phase 2: computing the impact of $L(x)$ */ ;
    max_impact $\leftarrow 0$;
    for $i \in \{1..n\}$ **do**
      $\boldsymbol{D} \leftarrow \lambda_i^*$;
      for $j \in \{1..m\}$ **do**
        $\boldsymbol{D} \leftarrow \boldsymbol{D} + \lambda_{n+j}^* . \boldsymbol{J}_{ji}$;
      end
      if $|\boldsymbol{D}.wid(\boldsymbol{x}_i)| > max\_impact$ **then**
        max_impact $\leftarrow |\boldsymbol{D}.\text{wid}(\boldsymbol{x}_i)|$;
        var $\leftarrow i$;
      end
    end
    return var;
  else
    return smearsum($\boldsymbol{x}$, $\boldsymbol{J}$);
  end
end.

Algorithm 2: `lsmear`

[5] Actually we need to solve the dual problem, however we use a linear solver (SoPlex [24]) which solves both, the primal and the dual problems

Algorithm 2 shows our approach. $\boldsymbol{J}$ corresponds to the Jacobian matrix which contains the interval overestimations of the partial derivatives over $\boldsymbol{x}$. In the linear program (Phase 1), the interval related to the objective variable is unbounded to enhance the chances for successfully finding an optimal solution. In Phase 2, for each variable x_i we first compute $\boldsymbol{D}$, which is an interval overestimation of $\frac{\partial L}{\partial x_i}$ over $\boldsymbol{x}$. The overestimation is obtained by adding the products of the interval partial derivatives on each constraint ($\boldsymbol{J}_{ji}$) and the corresponding dual optimal value (λ^*_{n+j}). $\boldsymbol{D}$ is initialized with the dual optimal value related to the bounded constraint, i.e., λ^*_i (the partial derivative related to the i-th bound constraint over the variable x_i is 1). Then, the smear impact of the variable corresponds to the magnitude of the product of the interval partial derivative and the width of the related interval, i.e., $|\boldsymbol{D}.\text{wid}(\boldsymbol{x}_i)|$. The variable with the maximum impact is saved and returned. If the linear program does not have solutions or if the optimal value is unbounded, then the `smearsum` method is launched instead. This method uses the same Jacobian matrix used by `lsmear`.

4.1 Variants

We implemented four variants of the basic strategy. The first of them, `lsmear-MG` (Midpoint Gradient) generates the linearization by using the gradient in the midpoint of the box, instead of using the midpoint of the *overestimate* of the partial derivatives in the box. Thus, `lsmear-MG` requires to compute two gradients, one for generating the linearization, and the other for computing the smear impact of the composed function.

A second variant is called `lsmear-LI` (Linear Impact). It computes the impact using the Lagrangian of the linearized problem, i.e.,

$$L'(x) = x_o + \sum_{i=1}^{m} \lambda^*_j gl_j(x)$$

To do so, we only need to replace $\boldsymbol{J}_{ji}$, by $\text{mid}(\boldsymbol{J}_{ji})$ in Phase 2. $\text{mid}(\boldsymbol{J}_{ji})$ corresponds to the partial derivative $\frac{\partial gl_j}{\partial x_i}$.

`lsmear-LR` (Linear Relaxation), uses a linear relaxation of the system generated by the linear-relaxation-based contractor of Ibex. This contractor generates a relaxed linear program by relaxing each constraint using two methods: X-Taylor and AF2. After the relaxation, each nonlinear constraint is represented by, between 0 to several, linear constraints in the linear program. In order to compute the vector λ^* which will be used to weight the constraints for computing the variable impacts, we have to map the dual optimal values related to the linear program to the λ^* vector related to the original nonlinear constraints. Thus, for instance, if a nonlinear constraint g_i was approximated by three linear constraints, then λ^*_i is computed by adding the dual optimal values related to these three constraints. Furthermore, for implementing `lsmear-LR` we have to modify slightly the solver algorithm 1. Because we need that the linear-relaxation-based contractor computes λ^*, and this contractor is inside the `pruning` method, we compute and choose the next variable to bisect, related to a box $\boldsymbol{x}$, just after pruning this box (without even bisecting $\boldsymbol{x}$). The chosen variable is stored in the corresponding node until the node is treated again and a bisection on this variable domain is performed.

`lsmear-LRI` (Linear Relaxation Impact) uses the same linear relaxation used by `lsmear-LR` for computing the vector λ^*. Then, it computes the impact using the Lagrangian of the linear relaxation instead of the Lagrangian of the original problem. This variant follows the same idea as ViolationTransfer but using a different convex relaxation and the box $\boldsymbol{x}$ instead of $\boldsymbol{x}^v$.

Remark that, the implementation of `lsmear-LR`, `lsmear-LRI` and ViolationTransfer require to modify the convex-relaxation-based contractor of the solver in order to obtain the dual optimal values. That is not the case of strategies `lsmear`, `lsmear-LI` and `lsmear-MG` which use a cheap and independent linearization-based method for computing these values.

5 Experiments

In order to validate our approach, we implemented the `lsmear` method and its variants in IbexOpt, a state-of-the-art optimizer of the Interval-Based EXplorer library (Ibex [25]).

IbexOpt is composed of several contractors including: `HC4` [16], `ACID(HC4)` [19] and a linear relaxation based contractor combining two relaxation methods: `AF2` [13] and `XNewton` [22]. For the upper bounding IbexOpt uses `inHC4` and `InnerPolytope` [14]. The selection of the next box is performed by `FeasibleDiving` [15]. Precision parameter ϵ was set to 10^{-6} (value commonly used by other solver benchmarks, e.g., [26,27]).

All the experiments were run on a server PowerEdge T420, with 2 quad-processors Intel Xeon, 2.20 GHz and 8 GB RAM.

The instances were selected from the series 1 and 2 of the COCONUT constrained global optimization benchmarks.[6] We selected all the problems solved by some strategy in a time comprised between 2s and 7200s (76 instances). Due to some random choices performed inside Ibex, each strategy was run five times on each instance. In order to avoid outliers we removed the lowest and highest CPU time and we report the average of the remaining three executions. Timeouts indicate that the strategy could not solve the instance in less than 7200 seconds in any of the three executions.

Definition 2 **Average relative gain.** We define as relative time $t_r(a, b, \pi)$ the ratio between the mean time ($\bar{t}$) taken by a strategy a and the sum of mean times taken by the strategies a and b in solving an instance π, i.e., $t_r(a, b, \pi) = \frac{\bar{t}(a,\pi)}{\bar{t}(a,\pi)+\bar{t}(b,\pi)}$. Thus, the **average relative gain** of a strategy a w.r.t. a strategy b is given by $\frac{\sum_{\pi\in\Pi}(t_r(b,a,\pi))}{\sum_{\pi\in\Pi}(t_r(a,b,\pi))}$, where Π is the set of the considered instances.

5.1 `lsmear` v/s classical strategies

In a first series of experiments, we compared the proposed strategy `lsmear` with some of the classical variable selection strategies: `round-robin` (rr), `largest-first` (lf), `smearsum` (ssum), `smearmax` (smax) and `smearsumrel` (ssr). All strategies are restricted to select variables with interval sizes larger than 10^{-7}. This condition avoids, in some pathological cases, to bisect some variables to almost null-size intervals.[7]

Figure 1 summarizes the comparison among the six strategies. From the set of instances solved in [2, 7200] seconds by at least one of the six strategies, each curve reports the proportion of instances solved by the corresponding strategy in less than or equal to *factor* times the best reported CPU time. Factor corresponds to the CPU time spent by the strategy divided by the best CPU time spent on each instance.

[6] www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html.

[7] When a box reaches a size lower than 10^{-7}, the bisector selects the next variable without taking into account the restriction. This situation is very rare and does not occur in any of the selected instances.

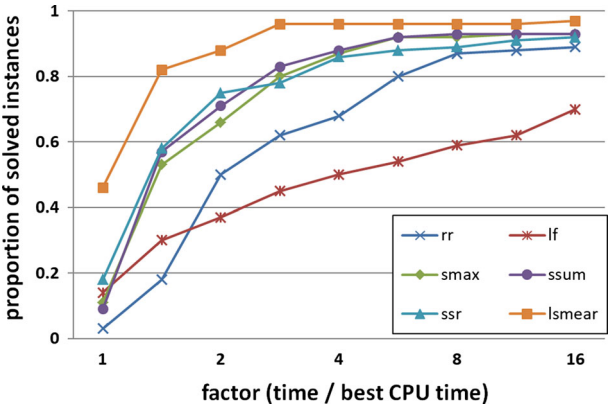


Fig. 1 Performance profile. Comparison among `lsmear` and the classical variable selection strategies

From the figure we observe that lf reports the worst results. The smear-based strategies: ssum, smax and ssr report comparable performances. `lsmear` clearly outperforms all the classical variable selection strategies. Note, for instance, that almost 90% of the instances are solved by `lsmear` in less than twice the best CPU time reported by all the strategies.

Table 1 reports the results of the strategies for a subset of instances. The subset comprises all the instances such that the ratio between the lowest and largest CPU time among the strategies is larger than or equal to 5.0. For each strategy we report the CPU time and the size of the tree search in number of treated nodes. The last column reports, for each instance, the gain of `lsmear` w.r.t. the best classical strategy on that particular instance. The gain is computed: $best\_time/time(\texttt{lsmear})$. The last row reports the CPU time spent by each strategy in the whole set of 76 instances. Timeouts are represented by the letters "to". A superindex i in the CPU time indicates that i of the 3 executions of the corresponding strategy reached a timeout. In this case, the CPU time and the number of nodes are computed with the executions completed in less than 7200 seconds.

Comparing the times and number of boxes for different strategies, remark that solving the linear program by the `lsmear` bisector seems to produce a negligible run-time overhead. This is mainly due to the contractor of the solver, based on X-Taylor and AF2, is much more expensive: it requires to solve up to $2n$ linear programs in each node for attempting to improve the bounds of the n variable domains.

Figure 2 compares the CPU times among `lsmear` and the classical strategies. The average relative gain of `lsmear` w.r.t. the other strategies is 2.04 (rr), 2.48 (lf), 1.34 (smax), 1.30 (ssum) and 1.32 (ssr). We then consider that ssum is the best classical strategy and will be used as reference in the next sections.

5.2 Variants of `lsmear`

In a second series of experiments we compared the different proposed variants of `lsmear`. Figure 3 summarizes the comparison among the strategies.

`ssum-active` corresponds to a variant of `smearsum` in which *some* inactive constraints are detected and discarded. We discard those constraints $g_j(x) \leq 0$ such that the enclosure of the image, computed by means of the natural extension, of g_j over the current box is strictly lower than 0. From the set of instances solved in [2, 7200] seconds by at least one of

Table 1 Average CPU time and number of boxes for the strategies rr, lf, ssum, smax and ssr and `lsmear` in a subset of instances

Instance	rr		lf		ssum		smax		ssr		lsmear		gain
	time	#box	time	#box	time	#box	time	#box	time	#box	time	#box	
ex5_4_4	1020	12,626	to	to	**200**	**2626**	222	3248	330	4148	229	2773	0.87
ex6_1_1	22	3002	86	20,788	**13**	**1505**	**13**	1625	**13**	1590	**13**	1615	1.00
ex6_1_3	227	12,109	899	85,173	63	3215	59	3163	73	3516	**54**	**2894**	1.09
ex6_2_6	156	88,643	3577	2,581,127	28	14,583	28	14,565	32	16,621	**25**	**12,711**	1.13
ex6_2_8	67	36,964	1629	952,892	19	9535	19	9376	19	9439	**16**	**8456**	1.18
ex6_2_9	50	25,153	2702	2,136,817	15	6157	13	5553	11	4816	**11**	**4796**	1.00
ex6_2_10	3191	1,150,155	to	to	**268**	**68,736**	269	68,996	269	68,993	275	68,746	0.97
ex6_2_11	31	19,959	564	365,123	11	5932	11	5823	11	6259	**10**	**5191**	1.10
ex6_2_12	13	7529	236	233,560	6.0	2681	5.2	2396	4.9	2276	**3.9**	**1833**	1.26
ex7_2_3	**7.2**	**1703**	23	7123	17	5380	20	7029	8.0	2075	2753	1,877,486	0.00
ex7_2_9	20	3090	18	2614	14	1720	**14**	**1686**	to	to	34	11639	0.39
ex7_3_4	to	2,615,244	to	to	14	1609	20	2035	**4.8**	**462**	5.3	588	0.91
ex7_3_5	5.7	295	19	917	7.9	364	8.0	388	**3.9**	**247**	4.2	250	0.93
ex8_4_4	263	5894	384	10,239	38	867	61	1476	**41**	**959**	61	1566	0.61
ex8_4_5	2258	87,265	**39**	**1127**	210	8724	220	9681	221	10,292	612	73,127	0.06
ex8_5_1	6.5	1744	to	to	5.3	1157	4.7	1024	5.8	1279	**4.1**	**888**	1.15
ex8_5_2	14	3267	to	to	15	3109	12	2542	17	3415	**10**	**2126**	1.26
ex14_2_1	1.0	270	7.0	2209	0.5	77	**0.4**	63	0.5	70	**0.4**	**61**	1.00
ex14_2_3	2.0	284	28	4573	2.0	259	2.0	255	2.0	251	**1.7**	**207**	1.18
ex14_2_7	52	7567	1553	223,378	23	2099	24	2252	**22**	**2087**	25	2209	0.91
bearing	27	9610	114	39,701	44	11,798	48	13,251	**5.6**	**690**	15	1713	0.38
chem	3734	520,476	to	to	7173(2)	838,170	to	to	6513	803,920	**393**	**50,555**	9.16
harker	1400	69,031	to	to	3035	138,881	2410	113,515	to	to	**1071**	**50555**	1.31
immun	**3.7**	**661**	16	1949	19	2552	16	2336	31	4670	9.3	1606	0.40

Table 1 continued

Instance	rr		lf		ssum		smax		ssr		lsmear		gain
	time	#box	time	#box	time	#box	time	#box	time	#box	time	#box	
launch	114	2358	96	1501	47	772	46	733	46	763	**15**	**214**	3.01
linear	to	to	**56**	**1112**	214	5511	120	3501	226	5857	137	3811	0.41
process	1.9	319	14	1822	2.2	339	1.6	233	**1.3**	**194**	1.7	233	0.76
ramsey	18	361	10	182	**10**	**160**	11	189	to	to	11	207	0.89
srcpm	to	to	to	to	487	19,515	331	13,985	to	to	**97**	**3559**	3.43
batch	to	30,392	to	to	105	863	105	843	98	733	**54**	**432**	1.83
disc2	to	to	to	to	to	to	to	to	33	673	**32**	**502**	1.03
dualc1	11	1588	3050	306,956	5.1	485	5.2	**474**	**5.0**	477	**5.0**	494	1.00
dualc2	1.0	160	1263	161,200	**0.8**	107	**0.8**	101	0.9	107	**0.8**	**93**	1.00
dualc5	69	17,525	to	to	**53**	**12,604**	**53**	12,616	54	11,992	58	14,301	0.91
haifas	269	98,827	to	to	4532	1,607,782	to	to	2.0	890	**1.7**	**722**	1.18
haldmads	to	to	**39**	**4390**	6271	232,187	5973	263,801	6500	294,930	5189	216,019	0.01
hs087	2.3	392	8.9	1448	10	1543	28	4061	1.0	145	**0.6**	**65**	1.67
hs093	3102	1,735,441	to	to	670	227,936	**654**	**219,361**	658	222,024	744	254,136	0.88
hs103	1068	172,019	to	to	726	95,312	790	108,327	**590**	**75179**	623	76,421	0.95
hs117	1981	146,455	to	to	383	20,215	581	30,717	520	28,723	**272**	**14,503**	1.41
hs119	20	576	187	6049	14	381	**13**	**374**	18	448	14	428	0.94
makela3	3.5	885	2.8	368	**0.9**	**149**	4.9	1144	**0.9**	**149**	1.2	193	0.75
mistake	1063	141,163	to	to	1077	136,450	**601**	**71,593**	to	to	732	81,966	0.82
odfits	66	16,905	92	27,451	19	3513	17	2792	to	to	**15**	**2561**	1.13
robot	1725	447,608	**343**	**74,593**	674	153,963	922	220,925	573	128,116	367	79,535	0.94
total time	77,602		144,472		43,867		47,474		65,789		**21,035**		2.09

In bold, the best CPU times and the smallest number of boxes reported on each instance

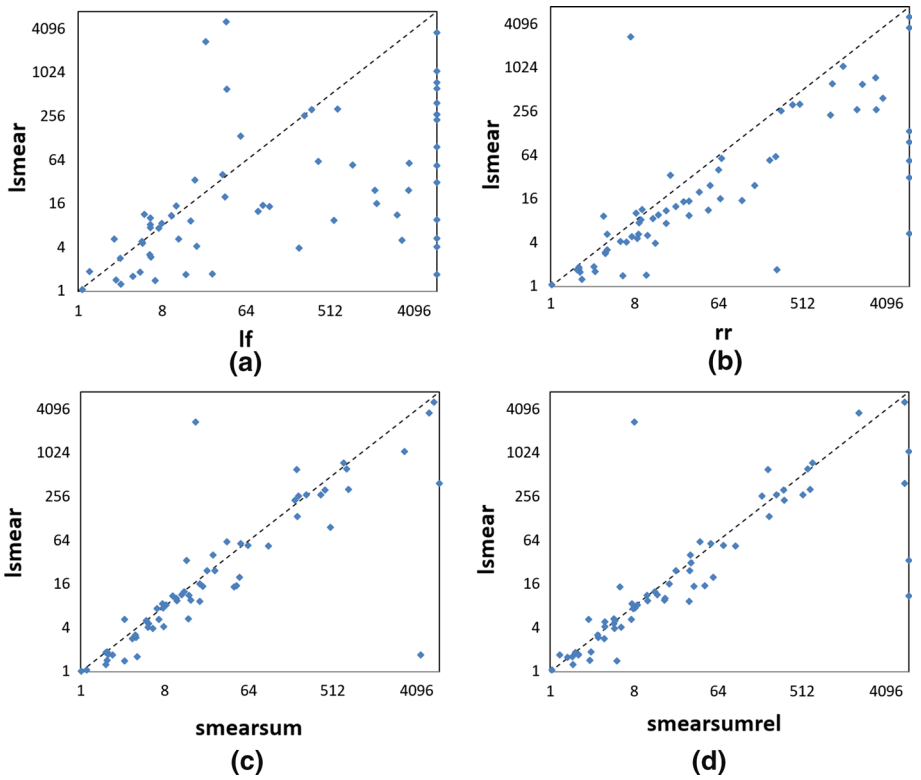


Fig. 2 Each plot compares the CPU times of two strategies for each of the instances. Each point corresponds to an instance and its coordinates indicate the average CPU time (in seconds) required by each of the two strategies in solving this instance. For example, the bottom right point in the *lsmear versus smearsum* plot corresponds to an instance solved in about 2s by the `lsmear` strategy and about 4096s by `smearsum`. Note that most of the points appear in the lower-right side of the figure, which highlights that `lsmear` dominates the other strategies on most instances. **a** `lsmear` versus `largest-first`. **b** `lsmear` versus `round-robin`. **c** `lsmear` versus `smearsum`. **d** `lsmear` versus `smearsumrel`

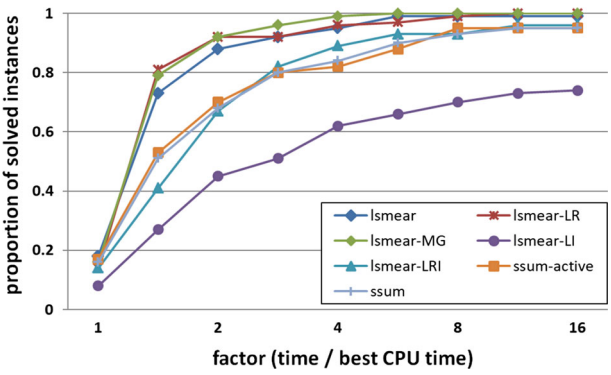


Fig. 3 Performance profile. Comparison between the different variants of `lsmear`

Table 2 Average CPU time and gain for all the strategies in a subset of instances. In bold, the best CPU times and the smallest number of boxes reported on each instance

Instance	lsmear			lsmear-MG			lsmear-LR		
	time	#box	gain	time	#box	gain	time	#box	gain
hs087	**0.6**	65	1.00	**0.6**	**46**	1.00	5.7	953	0.11
bearing	**15**	**1713**	1.05	16	1877	0.96	57	14,197	0.26
harker	**1071**	**50,555**	1.41	1514	81,121	0.71	3865	203,997	0.28
immun	9.3	1606	0.54	8.6	1660	0.58	**5.0**	**829**	1.72
linear	137	3811	0.19	**25**	**1557**	4.53	115	3360	0.22
ex7_2_3	2753	1,877,486	0.00	**11**	**3557**	1.52	16	4846	0.66
srcpm	**97**	**3559**	1.13	110	3552	0.88	305	12573	0.32
optprloc	2.8	9	0.36	2.8	9	0.36	**1.0**	**11**	2.80
ex7_3_4	5.3	588	0.83	**4.4**	**403**	1.20	11	1268	0.41
ex8_2_1	4.6	10	0.26	4.7	11	0.26	**1.2**	**12**	3.83
ex8_4_5	612	73,127	0.35	647	548,282	0.33	**211**	**8826**	2.90
hs114	1.5	190	0.60	1.7	268	0.53	**0.9**	**122**	1.67
total time	21,035		0.91	**19,181**		1.07	20,429		0.94

the six strategies, each curve reports the proportion of instances solved by the corresponding strategy in less than *factor* times the best reported CPU time.

Note that `lsmear-MG`, `lsmear-LR` and `lsmear` show the best performance. The average relative gain of the best variant, `lsmear-MG`, w.r.t. the others is 1.07 (`lsmear`), 1.05 (`lsmear-LR`), 2.25 (`lsmear-LI`), 1.43 (`lsmear-LRI`), 1.42 (`ssum-active`) and 1.43 (ssum). The variants `lsmear-LRI` and `lsmear-LI`, which compute the impact using the Lagrangian of the linear relaxation, instead of the Lagrangian of the original problem, report the worst results.

`ssum-active` reports results quite similar to those of `smearsum`, highlighting that only detecting active constraints but without pondering them, does not improve the performance of the smear-based strategies.

Table 2 reports the results of `lsmear` and its best variants for a subset of instances. The subset comprises all the instances such that the ratio between the lowest and largest CPU time among the strategies is larger or equal than 2.0. For each strategy we report the CPU time, the number of boxes of the search tree and the gain (i.e., the quotient between the best CPU time reported by the other strategies and the CPU time reported by the corresponding strategy). The last row reports the CPU time spent by each strategy in the whole set of instances.

Figure 4-left compares the CPU times among `lsmear` and `lsmear-MG`. Note that, excepting two or three instances in which `lsmear-MG` reports the best performance, the results are quite similar. Figure 4-right compares the CPU times among `lsmear-LR` and `lsmear-MG`. Both strategies report comparable results in the set of instances. However, we prefer `lsmear-MG`, because (1) according to the average relative gain (1.05) it is slightly better than `lsmear-LR` and (2) its implementation is simpler: it does not require to modify the contractor of the solver.

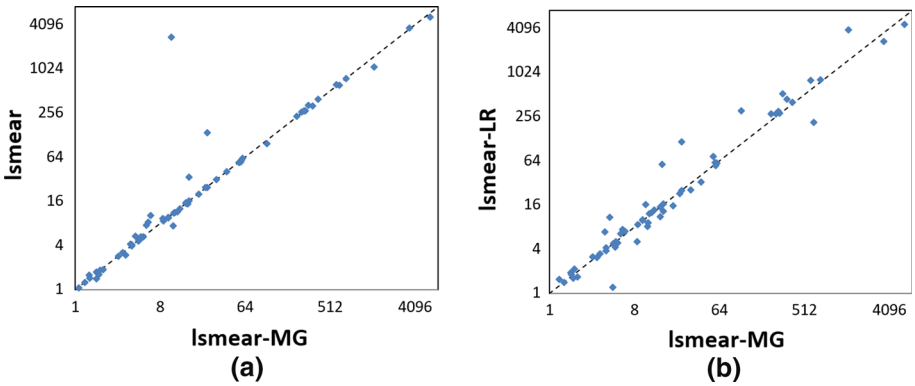


Fig. 4 Comparison between `lsmear-MG`, `lsmear` and `lsmear-LR`. Each plot compares the CPU times of two strategies for each of the instances. Each point corresponds to an instance and its coordinates indicate the average CPU time (in seconds) required by each of the two strategies in solving this instance. **a** `lsmear` versus `lsmear-MG`. **b** `lsmear` versus `lsmear-LR`

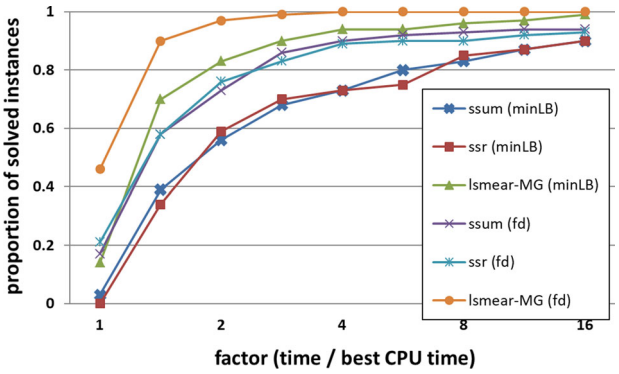


Fig. 5 Performance profile. Comparison of variable selection strategies using different search strategies

5.3 Evaluating `lsmear` with different search strategies

In this section we evaluate the performance of `lsmear` when two different search strategies are used: `FeasibleDiving` and `minLB`. `minLB` is a commonly used search strategy which selects next the box minimizing the lower bound of the objective function [12–14]. In [15] we showed that `FeasibleDiving` outperforms significantly `minLB` on interval-based solvers.

Figure 5 summarizes the comparison among the strategies `smearsum`, `smearsumrel` and `lsmear-MG` using `minLB` and the `FeasibleDiving` (fd) search strategy.

From the set of instances solved in [2, 7200] seconds by at least one of the strategies, each curve reports the proportion of instances solved by the corresponding strategy in less than *factor* times the best reported CPU time. First note that the `FeasibleDiving` strategy outperforms the classical `minLB` approach no matter which variable selection strategy is used. Similarly, the `lsmear-MG` strategy outperforms the `smearsum` and `smearsumrel` methods no matter which node selection strategy is used. Furthermore, note that the performance of `lsmear-MG` using the `minLB` strategy is slightly better than `smearsum` and `smearsumrel` using the `FeasibleDiving` search strategy. The average relative gain of

Table 3 Average CPU time for the strategies in a subset of instances. In bold, the best CPU time reported on each instance

	`minLB`				`FeasibleDiving`			
	ssum	ssr	MG	gain	ssum	ssr	MG	gain
ex7_2_9	20	to	34	0.59	**14**	to	16	0.85
ex7_3_4	32	4.9	6.9	0.71	14	4.8	**4.4**	1.11
ex8_4_5	to	to	to	1.00	**210**	221	647	0.32
ex14_1_7	924	285	296	0.97	221	**188**	254	0.74
alkyl	9.3	4.4	0.9	4.66	1.9	2.1	**0.7**	2.65
bearing	28	8.0	6.3	1.27	44	**5.6**	16	0.36
chem	7167$^{(2)}$	6477	397	16.3	7173$^{(2)}$	6513	**381**	17.1
harker	3522	to	2082	1.69	3035	to	**1514**	2.00
hhfair	9.1	12	21	0.44	**6.7**	7.8	11	0.61
himmel16	110	163	31	3.51	**26**	32	41	0.65
hydro	121	174	134	0.90	20	35	**15**	1.37
immun	271	2887	128	2.12	19	31	**8.6**	2.23
launch	to	45	21	2.19	47	46	**15**	3.05
linear	321	335	36	9.03	214	226	**25**	8.43
process	5.6	3.4	1.1	2.99	2.2	1.3	**0.6**	2.21
ramsey	55	to	20	2.72	**10**	to	11	0.88
srcpm	**103**	to	134	0.77	487	to	110	4.44
batch	365	436	92	3.96	105	98	**55**	1.78
disc2	to	190	239	0.80	to	33	**32**	1.03
haifas	2062	3.3	3.4	0.95	4532	2.0	**1.7**	1.18
himmelbk	177	144	29	5.02	51	57	**21**	2.45
hs087	70	0.9	0.9	0.93	10	1.0	**0.6**	1.79
hs104	20	54	7.7	2.60	11	17	**6.4**	1.68
hs114	6.8	10	1.8	3.70	3	5.2	**1.7**	1.76
mistake	735	5002	603	1.22	1077	to	**542**	1.99
odfits	18	to	15	1.24	19	to	**14**	1.28
total time	59,274	76,392	29,136	2.03	43,867	65,789	**19,181**	2.29

`lsmear-MG` w.r.t. `smearsum` when the `minLB` search strategy is used is 1.63. The average relative gain of `lsmear-MG` w.r.t. `smearsum` when the `FeasibleDiving` search strategy is used is 1.43.

Finally, table 3 reports the results of the same strategies for a subset of instances. The subset comprises all the instances such that the ratio between the lowest and largest CPU time among the strategies is larger or equal than 3.0. For each strategy we report the CPU time. We also report the gain of `lsmear-MG` w.r.t. the best competitor for each search strategy. The last row reports the CPU time spent by each strategy in the whole set of instances.

The best times are highlighted in bold. Note that most of them correspond to the `lsmear-MG` strategy.

6 Conclusions

In this work we present `lsmear`, a new variable selection strategy for interval-based solvers. Unlike the well-known smear-based strategies, `lsmear` weights the constraints by using the optimal Lagrangian multipliers of a linearization of the problem and computes the smear value of a Lagrangian-like function. The Lagrangian multipliers are simply computed by solving a Taylor-based linearization of the original problem by using the simplex method.

The results are promising. `lsmear` and `lsmear-MG` (its best variant) significantly outperform all the bisection strategies used by classical interval-based solvers. They also outperform `lsmear-LRI` a variant closer to the method ViolationTransfer which also uses the optimal dual values. The difference in performance between `lsmear-MG` and `lsmear-LRI` is mainly due to that, after computing the dual values, `lsmear-MG` computes the variable impacts using the original constraints of the system instead of using a linear relaxation of them.

Related to ViolationTransfer, `lsmear` and `lsmear-MG` have another advantage: they are independent of the convex-relaxation-based contractor used by the solver. ViolationTransfer is intrusive in the sense that it requires to modify the contractor for obtaining the optimal Lagrangian multipliers. Furthermore, ViolationTransfer requires to modify the general strategy of the solver: variables to bisect are chosen just after pruning the box, thus, this election is saved in the node until the node is treated again and the bisection is actually performed.

In future work we will attempt to include the mechanism of ViolationTransfer which also uses the constraint violation errors to weight the variables. According to this strategy, variables related *only* to linear constraints would not have any impact and they should not be bisected. This make sense, since contractors based on linear relaxations are optimal for reducing these variable domains. However, it is also true that bisecting this variables could help, indirectly, to reduce the domains of other variables related to nonlinear constraints.

Acknowledgements Ignacio Araya is supported by the Fondecyt Project 1160224.

References

1. Araya, I., Reyes, V., Oreallana, C.: More smear-based variable selection heuristics for NCSPs. In: International Conference on Tools with Artificial Intelligence (ICTAI 2013). IEEE, pp. 1004–1011 (2013)
2. Csendes, T., Ratz, D.: Subdivision direction selection in interval methods for global optimization. SIAM J. Numer. Anal. **34**(3), 922–938 (1997)
3. Granvilliers, L.: Adaptive bisection of numerical CSPs. In: Principles and Practice of Constraint Programming, pp. 290–298. Springer, New York (2012)
4. Kearfott, R.B., Novoa III, M.: Algorithm 681: INTBIS, a portable interval Newton/bisection package. ACM Trans. Math. Softw. (TOMS) **16**(2), 152–157 (1990)
5. Lagouanelle, J.-L., Soubry, G.: Optimal multisections in interval branch-and-bound methods of global optimization. J. Glob. Optim. **30**(1), 23–38 (2004)
6. Trombettoni, G., Araya, I., Neveu, B., Chabert, G.: Inner regions and interval linearizations for global optimization. In: AAAI Conference on Artificial Intelligence, pp. 99–104 (2011)
7. Trombettoni, G., Chabert, G.: Constructive interval disjunction. Princ. Pract. Constr. Program. **2007**, 635–650 (2007)
8. Moore, R.: Interval analysis, vol. 60. Prentice-Hall Englewood Cliffs, New Jersey (1966)
9. Ratz, D.: Automatische ergebnisverikation bei globalen optimierungsproblemen. Ph.D. dissertation, Universität Karlsruhe (1992)
10. Hansen, E., Walster, G.W.: Global Optimization Using Interval Analysis: Revised and Expanded, vol. 264. CRC Press, Boca Raton (2003)
11. Tawarmalani, M., Sahinidis, N.V.: Global optimization of mixed-integer nonlinear programs: a theoretical and computational study. Math. Program. **99**(3), 563–591 (2004)

12. Lebbah, Y.: Icos: a branch and bound based solver for rigorous global optimization. Optim. Methods Softw. **24**(4–5), 709–726 (2009)
13. Ninin, J., Messine, F., Hansen, P.: A reliable affine relaxation method for global optimization. 4OR **13**(3), 247–277 (2015)
14. Araya, I., Trombettoni, G., Neveu, B., Chabert, G.: Upper bounding in inner regions for global optimization under inequality constraints. J. Optim. Glob. (2014). doi:10.1007/s10898-014-0145-7
15. Neveu, B., Trombettoni, G., Araya, I.: Node selection strategies in interval branch and bound algorithms. J. Glob. Optim. **64**(2), 289–304 (2016)
16. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revising hull and box consistency. In: International Conference on Logic Programming, Citeseer (1999)
17. Araya, I., Trombettoni, G., Neveu, B., et al.: Exploiting monotonicity in interval constraint propagation. In: AAAI (2010)
18. Lhomme, O.: Consistency techniques for numeric CSPs. In: IJCAI. Citeseer, pp. 232–238 (1993)
19. Neveu, B., Trombettoni, G., et al.: Adaptive constructive interval disjunction. In: International Conference on Tools with Artificial Intelligence (ICTAI), pp. 900–906 (2013)
20. Lebbah, Y., Michel, C., Rueher, M.: An efficient and safe framework for solving optimization problems. J. Comput. Appl. Math. **199**(2), 372–377 (2007)
21. Baharev, A., Achterberg, T., Rév, E.: Computation of an extractive distillation column with affine arithmetic. AIChE J. **55**(7), 1695–1704 (2009)
22. Araya, I., Trombettoni, G., Neveu, B.: A contractor based on convex interval taylor. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pp. 1–16. Springer, New York (2012)
23. Goldsztejn, A., Lebbah, Y., Michel, C., Rueher, M.: Revisiting the upper bounding process in a safe branch and bound algorithm. In: Principles and Practice of Constraint Programming (CP), pp. 598–602. Springer, New York (2008)
24. Wunderling, R.: Soplex: The sequential object-oriented simplex class library. http://www.zib.de/Optimization/Software/Soplex/soplex.php (1997)
25. Chabert, G., Jaulin, L.: Contractor programming. Artif. Intell. **173**, 1079–1100 (2009)
26. Tawarmalani, M., Sahinidis, N.V.: A polyhedral branch-and-cut approach to global optimization. Math. Program. **103**(2), 225–249 (2005)
27. Misener, R., Floudas, C.A.: ANTIGONE: Algorithms for continuous/integer global optimization of nonlinear equations. J. Glob. Optim. **59**(2), 503–526 (2014)

FRG: A Fill-and-Reduce Greedy Algorithm for the Container Pre-Marshalling Problem

Abstract

We address the Container Pre-Marshalling Problem (CPMP). The CPMP consists in ordering containers in stacks such that the retrieval of these containers is carried out without additional movements. The ordering has to be done in a minimum number of steps. Target-guided constructive heuristics report very good results in a short time. At each step, they select one poorly located container and rearrange it to an adequate position by a sequence of movements. The sequence of movements is generally generated by following a set of rules. In this work, we propose a different and more direct approach. Whenever possible, ordered stacks are *filled* by directly moving badly placed containers into them such that the containers become well placed. If it is not possible, then a stack is emptied or *reduced* to have more available slots, and the process is repeated. Despite the fact that the fill-and-reduce algorithm is not as specifically targeted as the target-guided algorithms (meaning that it does not have a predetermined goal for each movement), it performs better than the target-guided algorithms in the majority of the large instances of classical benchmark sets. Furthermore, when embedded in a beam search algorithm, it reports the best results compared to traditional techniques that do not use machine learning.

Keywords: (O) Combinatorial optimization, Container Premarshalling Problem, Constructive Algorithms, Beam Search Transportation

Acknowledgments

This work is supported by the Fondecyt Project 1200035.

1. Introduction

Containerization has strongly improved international trade. Containers allow goods to be carried between different places without directly handling

freight. One of the main challenges faced by world container ports is the pressure to accommodate larger ships [20]. Several constraints, such as the ships' size, draft restrictions, and special handling requirements, force berth and crane operations to be meticulously organized and synchronized to offer fast service.

In a yard, a container passes through three stages. First, the container is placed into a stack by using assignment strategies. The objective here is to avoid blocking the retrieval of other containers. It is difficult to predict the retrieval order; thus, in the second stage, a container premarshalling mechanism is applied to reorder containers that were placed into *bad or blocking* locations. In the final stage, containers are extracted.

The Container Pre-Marshalling Problem (CPMP) is related to the second stage of the process. The CPMP consists in ordering containers in stacks such that the retrieval of these containers is carried out without additional movements. The objective is to reach such a configuration with a minimal sequence of container movements.

A yard is generally divided into several blocks. A block is composed of several parallel bays. The CPMP involves a single bay of the yard or *layout*. Each layout is formed by several stacks aligned side by side (see Fig. 1). In each stack, containers are stacked vertically. The height of stacks is constrained by the height of the operating equipment. One stack can typically store 5-12 containers. Containers are categorized into various groups. Each group is assigned a group value related to the future retrieval order of containers[1]. A layout is *feasible* when all the stacks are ordered, i.e., the containers in each stack are placed into a descending order of group values from the bottom up. We say that a container is *well-placed* if it is supported directly by the ground or another well-placed container with equal or larger group value; otherwise, it is *badly placed*. Containers can be moved from one stack to another, respecting the maximum height of the layout. The objective of the CPMP is to find a minimal sequence of moves such that the layout becomes feasible. It is worth highlighting that some layouts cannot be arranged due to the reduced number of empty slots. Boge et al. (2020) [3] propose a simple rule to know whether a layout can be successfully arranged or not. According to [18], the CPMP is NP-Hard, i.e., there is no known

[1]It is assumed that the retrieval order is known beforehand, and no container arrives at or leaves the layout during premarshalling.

algorithm that can solve it in polynomial time.

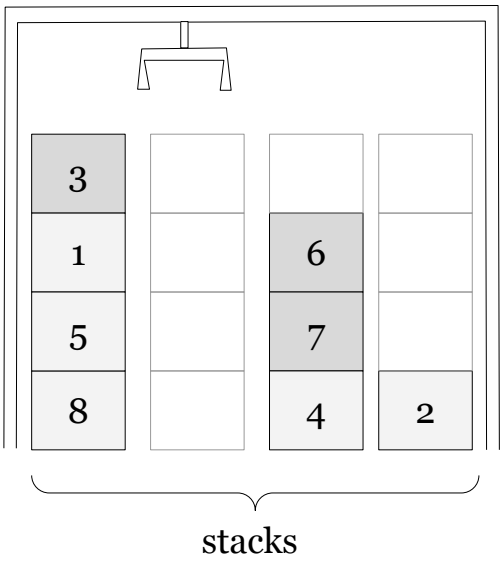


Figure 1: Example of a container layout.

Several exact techniques for solving the CPMP can be found in the literature. Integer programming models are proposed in [15, 6, 17]. A branch-and-price technique is applied by van Brink et al. (2014) [22]. The A* and IDA* approaches are presented by Expósito et al. (2012) [7] and Tierney et al. (2017) [21], respectively. Zhang et al. (2015) [25] propose a branch & bound approach that determines which branches are explored first based on a lower bound. A similar strategy is used by Tanaka et al. (2018) [19], with a different branching rule, an extra dominance rule, and a new lower bound that extends the lower bound presented in [4]. Tanaka et al. (2019) [20] extend the previous works by incorporating new lower bounds, new dominance rules and a memoization-based heuristic for finding feasible solutions. Jiménez-Piqueras et al. (2022) [12] propose a constraint programming approach for solving the problem. The best exact approaches (e.g., [19, 20]) are able to find optimal solutions to almost all problems up to a height of seven, making heuristic methods particularly interesting when we need to solve larger instances (height > 7) in a short time.

The heuristic strategies for solving the CPMP are mainly based on constructive approaches. These approaches generally start with the initial layout

and generate new layouts by applying promising moves. Layouts are heuristically evaluated, and the best ones are considered for the next iterations. At the end, the shortest found sequence of moves reaching a feasible layout is returned. Bortfeldt and Forster (2012) [4] describe a tree search procedure for solving the problem. In the procedure, solutions are constructed by compound moves instead of single moves. Gheith et al. (2014) [9] propose a rule-based heuristic procedure for solving the problem and then develop a variable chromosome length genetic algorithm [8]. A biased random-key genetic algorithm is proposed by Hottung et al. (2016) [11]. Hottung et al. (2020) [10] propose DLTS, a tree search guided with branching and bounding decisions made with two deep neural networks. DLTS reports the best results to date on a set of classical instances. While deep learning techniques offer very good results, they require being exhaustively trained with similar (and previously solved) training instances prior to solving the real ones. In addition, it is difficult to know how deep learning techniques do what they do. Thus, it becomes difficult to use their acquired knowledge in similar or related problems.

On the other hand, more traditional *no-machine-learning* heuristics also report very competitive results as white boxes in the sense that we can know clearly how they make their decisions in each step. The interpretability of algorithms is important, mainly for researchers designing new algorithms for solving the same or similar problems (for example, by analyzing a simple algorithm, we can identify which variables we should consider for making good decisions in a more sophisticated approach). Among them, the target-guided greedy heuristics report the best results to date [23, 24, 7, 13]. These heuristics handle containers (targets) in descending order of group values. At each step, a container with a large group value is rearranged and fixed to an *adequate* position in the layout by a sequence of moves. Wang et al. (2017) [24] verify the feasibility of the resultant state, ensuring that a feasible solution can be found at the end. Target-guided heuristics can be implemented independently as greedy algorithms or be a major component embedded in tree search algorithms such as beam search (e.g., Wang et al. (2015) [23]).

In this work, we propose a fill and reduce greedy algorithm (`FRG`). It is based on two simple procedures: (1) *filling* ordered stacks by moving directly badly placed containers into ordered stacks such that the containers become well-placed and (2) *reducing* one small stack, by moving its containers into other stacks, in order to have more available slots for well placing containers.

Furthermore, we define a set of intuitive heuristic rules that are considered by our greedy algorithm when performing its procedures. The main contributions of our work are as follows:

- A greedy algorithm (`FRG`) that applies simple and intuitive rules in a novel way. Although it is more *blind* than target-guided ones, it outperforms them in most large instances of classical benchmark sets.
- A beam search approach for exploiting `FRG`. The beam search mixes single and compound moves, reporting the best results compared to no-machine-learning algorithms.

In Section 2, we describe the problem and its related concepts. In Section 3, we analyze the single moves in more detail, and on this basis, we define heuristic methods for selecting promising moves. Section 4 presents our greedy algorithm for solving the CPMP. In Section 5, we present a simple beam search approach that uses the proposed greedy algorithm for evaluating states. In Section 6, we report the comparison with state-of-the-art strategies and experimental analysis. Finally, Section 7 concludes our work.

2. Container Premarshalling problem (CPMP)

The CPMP consists of an initial layout L of N containers, which are distributed in S stacks and H tiers. The number of empty slots is $E = S \cdot H - N$. Fig. 1 shows an example of a layout with $S = 4$, $H = 4$, $N = 8$, and $E = 8$ (containers are represented by boxes).

Every container is labeled a group value $g \in \{1, ..., G\}$ (in Fig. 1, the group values of the containers are marked inside). A container is **well-placed** if it is supported directly by the ground or another well-placed container with equal or larger group value; otherwise, it is **badly placed** (in the figure, light gray containers are well-placed, while dark gray containers are badly placed). B is the number of badly placed containers in a layout ($B = 3$ in the figure). Containers can be moved from one stack to another, respecting the maximum height of the layout. A layout is **feasible** when all the stacks are ordered, i.e., the containers in each stack are well placed. The objective of the CPMP is to find a minimal sequence of moves, such that after applying the sequence to the initial layout, the layout is feasible.

The height of a stack s is denoted by $h(s)$, and $e(s) = H - h(s)$ denotes the number of empty slots in stack s. Note that the height of stacks should

not exceed H. The container at the top of a stack s is denoted $top(s)$. The group value of a container c is denoted by $g(c)$. We also say that the group value of a stack s is equal to the group value at the top of the stack, i.e., $g(s) = g(top(s))$. An empty stack has a group value equal to G.

As previously mentioned, containers can be moved from one stack to another. We employ different names for container *single* moves as in [4]. A bad-good (BG) move is a move in which a badly placed container is moved to a stack where it becomes well placed. Similarly, a bad-bad (BB) move involves moving a badly placed container to another bad location. Good-bad (GB) and good-good (GG) moves are defined analogously. Furthermore, BX and GX refer to moves from a bad or good location, respectively, to any other stack. XG and XB moves are defined analogously.

3. Basic rules related to single moves

Before presenting our greedy algorithm for solving the problem, in this section, we highlight some observations related to the single moves that will be considered by the heuristic rules inside the algorithm.

3.1. Unblocked containers

Let us first define the concept of blocked/unblocked containers of a stack.

Definition 1 (Blocked/unblocked containers)**.** *A container c in a stack s is* **unblocked** *if (1) c is the top container of s or (2) the group values from the topmost containers of s to c included are nonincreasing. All the other containers of s are* **blocked***.*

From the definition, we can straightforwardly infer the following proposition.

Proposition 1. *Provided that $e(s_d)$ is large enough, a sequence of unblocked containers from a stack s_o can be well placed into a unique stack s_d, with $g(s_d) \geq g(s_o)$, by performing consecutive XG moves.*

After placing all the unblocked containers into s_d, the blocked containers from s_o cannot be directly well placed into s_d.

For instance, in Fig. 2-left, the first stack has 2 unblocked containers (i.e., they have nonincreasing group values: $6 \geq 1$). These containers could be well placed into a stack with a group value equal to or larger than 6, resulting

in the final stack s'_1 of the figure. Containers with group values of 5 and 8 cannot be well placed into the same stack without performing additional moves; thus, we say that they are **blocked**. Another example is shown in Fig. 2-right. In this case, all the containers of the stack s_2 are unblocked (i.e., they have nonincreasing group values: $7 \geq 3 \geq 2$).

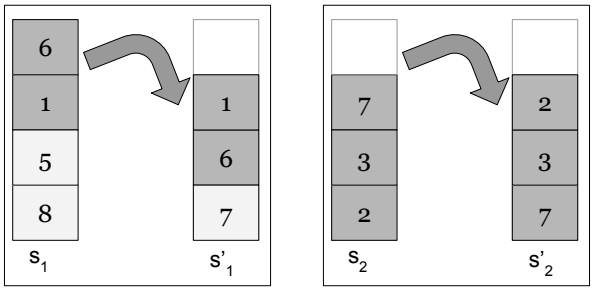


Figure 2: Examples of unblocked containers (dark gray containers of stacks s_1 and s_2). They can be well placed into a stack with a larger group value only by performing XG moves.

The presence of unblocked containers in a layout can yield benefits. They can be directly well-placed into a minimal number of ordered stacks, just providing the ordered stacks: (1) have enough available slots; and (2) have larger group values than the unblocked containers.

3.2. XG moves

We noted that when XG moves are performed, it is convenient to minimize the difference between the group value of the placed container and the group value of the destination stack. This is because, in this way, we increase the chance of placing a larger number of containers properly in the destination stack.

See the example of Fig. 3. The empty second stack is filled with containers that are badly placed into other stacks. Recall that the group value of an empty stack is equal to the maximum group value of all the containers, i.e., $G = 8$. The container with a group value equal to 7 is placed first, minimizing the difference between the group values. Then, the container with a group value equal to 6 is placed, and so on. Note that by applying this strategy, we solve the particular example problem optimally (of course, this is not the

general case).[2]

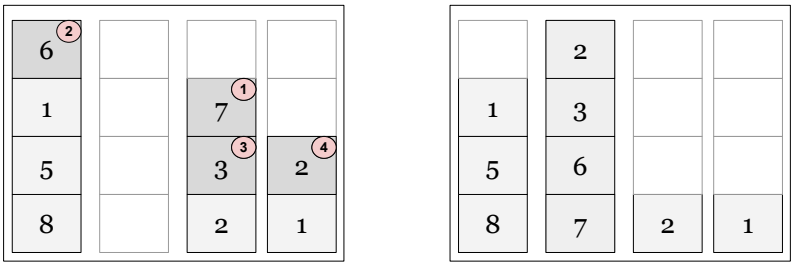


Figure 3: Example of performing BG moves, each move minimizing the difference between the group value of the container and the destination stack. On the left, the original layout. Numbers in circles represent the order in which the containers are picked up and placed into the second stack. On the right, the resulting feasible layout.

3.3. XB moves

When XB moves are carried out and the destination stack is *initially* unordered, to avoid blocking bad-placed containers, we consider that it is convenient that the group value of the placed container is larger than, or equal to, the group value of the destination stack. Additionally, minimizing the difference is preferred because, in this way, we can place more nonblocking containers in the stacks.

See the layout on the left of Fig. 4. To well-place the containers with group values of 7, 3, and 6, we decided to empty the first stack by moving the container c marked with a star. In this case, the best move consists in placing the container c in the third stack s_3 (sequence at the top part of the figure) because $g(c) = 5 \geq g(s_3) = 3$; thus, no additional container is blocked. Finally, all the bad-placed containers can be placed into the first stack (Fig. 4-top-right). If, however, the marked container c is placed into the fourth stack (sequence at the bottom part of the figure), then, after placing the container with a group value equal to 7 in the first stack, the container with a group value of 6 cannot be well placed into the first stack because it is blocked by c.

[2]All the bad-placed containers of the example are unblocked; for that reason, they can be placed in the empty stack by only performing BG moves.

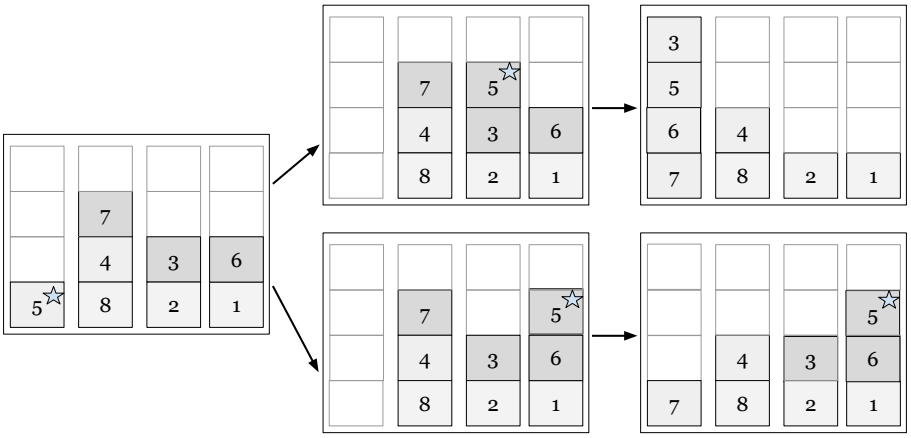


Figure 4: Example of performing an XB move. In the first sequence, the container marked with a star is placed into the third stack. In the second sequence, the container is placed into the fourth stack.

On the other hand, if the destination stack is ordered, blocking well-placed containers is considered to be less of an issue. This is because well-placed containers are less likely to require movement. Finally, if the only option is to block bad-placed containers, we prefer to perform moves to stacks with minimum size.

3.4. Heuristic methods for selecting moves

Based on previous observations, we defined the two basic heuristic methods that will be used inside our greedy algorithm for selecting moves. The method select_destination (Algorithm 1) selects a destination stack for placing a container c from some other stack.

First, the method prefers XG moves minimizing $g(s_d) - g(s_o)$; then, XB moves minimizing $g(s_o) - g(s_d)$ such that s_d is unordered and $g(s_d) \leq g(s_o)$; then, XB moves such that s_d is ordered; and finally, the first stack has the minimum size.

We also define a method for selecting the BG move minimizing the difference between the group value of the origin stack and the group value of the destination stack (Algorithm 2). In case where no BG move exists, the method returns **null**.

```
1  select_destination (c, S_d)
2  |  V ← {s_d | s_d ∈ S_d, h(s_d) < H}     # only Valid moves
3  |                                        # are considered
4  |  XG_d ← {s_d | s_d ∈ V, is_ordered(s_d), g(s_d) ≥ g(c)}
5  |  if XG_d ≠ ∅ then return arg min_{s_d ∈ XG_d} g(s_d) − g(c)
6  |  XB_d^1 ← {s_d | s_d ∈ V, is_unordered(s_d), g(s_d) ≤ g(c)}
7  |  if XB_d^1 ≠ ∅ then return arg min_{s_d ∈ XB_d^1} g(c) − g(s_d)
8  |  XB_d^2 ← {s_d | s_d ∈ V, is_ordered(s_d)}
9  |  if XB_d^2 ≠ ∅ then return arg min_{s_d ∈ XB_d^2} h(s_d)
10 |  return arg min_{s_d ∈ V} h(s_d)
```

Algorithm 1: Heuristic method for selecting a destination stack for a given container c.

```
1  select_BG_move (S)
2  |  BG ← {(s_o, s_d) | s_o, s_d ∈ S, is_valid_BG_move(s_o, s_d)}
3  |  if BG ≠ ∅ then
4  |  |  return arg min_{(s_o, s_d) ∈ BG} g(s_d) − g(s_o)
5  |  else
6  |  |  return null
```

Algorithm 2: Heuristic method for selecting a promising BG move.

4. The greedy algorithm

As in other works, we propose a greedy algorithm for solving the problem. Starting from an initial layout, the algorithm iterates performing two kinds of moves: (1) BG moves that fill ordered stacks (i.e., filling moves) and (2) reduction moves that place containers from small stacks into other stacks.

Algorithm 3 shows the pseudocode of the greedy algorithm `FRG` (fill and reduce greedy). It receives as input the initial layout $L = (\mathcal{S}, s_r)$ and performs moves iteratively until reaching a feasible layout. $\mathcal{S}$ corresponds to the set of stacks of L. The variable s_r points to the stack that is currently being reduced. As we prioritize BG moves, this variable is initialized to *none*.

First, the algorithm attempts to fill ordered stacks of the initial layout L by performing BG moves. Thus, whenever a BG move is possible, such a move is selected by the method select_BG_move (Line 4), and it is applied to L (Line 6). If such a move does not exist or s_r is *none* (i.e., move is equal to *none*), then a stack reduction is started by the method `reduction_move` (Line 5).

```
1 FRG (L = (S, s_r : none))
2     while L is not feasible do
3         move ← none
4         if s_r = none then move ← select_BG_move(S)
5         if move = none then L, move ← reduction_move(L)
6         L ← apply_move(L, move)
7         if stopping_reduction_criterion(L) = true then s_r ← none
8     return L
```

Algorithm 3: Fill and reduce greedy algorithm for solving the CPMP.

The method is described in Algorithm 4. When s_r is *none*, the method first selects a stack from $\mathcal{S}$ to be reduced (Line 3). We prioritize first to select stacks that have been selected fewer times. In case of a tie, the stack with minimum height is selected. If two or more stacks have the same height, then the *unordered* stack with the highest average group value is selected; otherwise, the *ordered* stack with the lowest average group value is selected. After selecting a stack to reduce (or when s_r is not *none*), the method starts (resp. continues), reducing the stack s_r. A reduction move simply consists in taking the top container c from the top of s_r and placing it into a stack selected by the method `select_destination`. The layout L (with a possible new value of s_r) and the move are returned. The move is then performed in Line 6 of the main procedure. In the following iterations of the main loop of `FRG`, the stack s_r will continue to be reduced until the method stopping_reduction_criterion returns *true* (Line 7 of the `FRG` procedure). If this is the case, s_r is set to *none*, and BG moves are prioritized again. Iterations are performed until a feasible layout is reached. The final layout L is returned at the end of the algorithm. The performed moves are stored in the layout structure.

In the next section, we detail the stopping criterion for reducing a stack.

```
1 reduction_move (L = (S, s_r) )
2   if s_r =none then
3     s_r ← select a stack from S # by frequency, then height,
        then...
4     c ← top(s_r)
5     s_d ← select_destination(c, S \ {s_r})
6     return L, (s_r, s_d)
```

Algorithm 4: Stack reducing procedure.

4.1. The stopping reduction criterion

Reduction moves are performed until one of the following conditions is reached (consider that $\mathcal{S}'$ corresponds to the set of unordered stacks in the layout L, except for s_r):

1. all the unblocked bad-placed containers in L can be well-placed into s_r, i.e.,
$$g(s_r) \geq \max_{s \in \mathcal{S}'} g(s), \text{ or}$$
2. the unblocked bad-placed containers of L that can be well-placed into s_r fill, at least, all the empty slots of s_r minus r, i.e.,
$$\sum_{\substack{s \in \mathcal{S}' \\ g(s) \leq g(s_r)}} ub(s) \geq e(s_r) - r,$$
where the function $ub(s)$ computes the number of unblocked bad-placed containers of a stack s and $r \geq 0$ is an integer parameter. When $r = 0$, we ensure that the remaining space s_r can be entirely filled with unblocked bad-placed containers of the layout after reduction. If we are optimistic, we may anticipate that performing BG moves post-reduction will unblock some other bad-placed containers, allowing us to relax the condition and consider selecting a non-zero value of r. In our experiments, we discovered that setting $r = 1$ produced the most satisfactory outcomes on average. Thus, we adopt this value in the subsequent analysis.

In Fig. 5, we consider that the stack s_r is being reduced. Dark gray containers correspond to the unblocked bad-placed containers that can be well placed into stack s_r (containers with group values 9 and 8 are also

unblocked, but they cannot be well placed because $g(s_2) > g(s_r)$). As they can fill the empty slots of s_r minus 1, the reduction finishes.

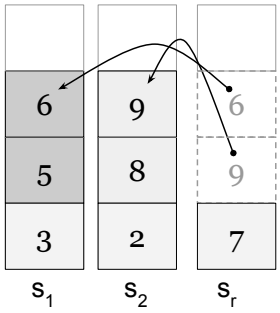


Figure 5: Example of reducing a stack s_r. After moving containers with group values 6 and 9, s_r has one container left. Dark gray containers correspond to the unblocked bad-placed containers that can be well placed into stack s_r.

Theorem 1. *(Execution time) Without considering the advanced features (see Section 4.3), an iteration of* `FRG` *is performed in* $O(S^2)$.

The proof is in Appendix A.

4.2. Termination of the algorithm

The main objective of reducing a stack is to make some space for well-placing containers by further BG moves. Thus, the algorithm may not terminate if, after performing a reduction (and the further BG moves), the number of bad-placed containers is not reduced.[3] This, assuming that if the arrangement of containers is identical, then the same stack should be selected for reduction (i.e., without considering frequency when selecting the stack s_r in Algorithm 4).

This situation only occurs when reducing an ordered stack s_r and its containers block all the badly placed containers of the other stacks. In this case, after performing the BG moves, we return to the same starting state. Fig. 6 shows an example. After reducing the stack s_r, its container with a group value of 6 blocks the bad-placed containers of the second stack. As

[3]It is commonly assumed that the available slots in a layout always allow for the emptying of any stack in the bay, i.e., $N \leq (S-1)H$.

there are no more unblocked bad-placed containers, further BG moves will cause a return to the initial state.

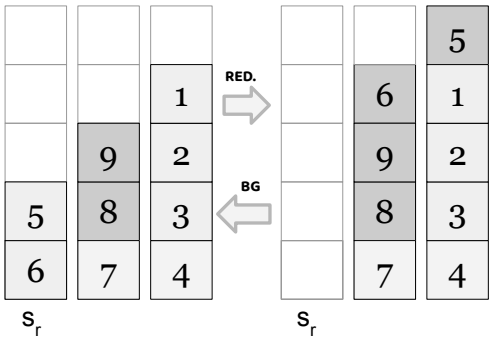


Figure 6: In this example, the algorithm does not terminate. The initial state on the left is reduced, resulting in the right layout. Then, BG moves cause a return to the initial state.

Note that, according to the selection criterion for destination stacks (see Algorithm 1), and assuming that after each move, the current top of s_r blocks the tops of unordered stacks that have not yet received a well-placed container from s_r, the stacks in the layout are filled sequentially, starting with the ordered stacks, then moving on to the unordered stacks from smallest to largest (based on their heights). To effectively block all the badly placed containers, the stack s_r should have enough well-placed containers to fill all the other stacks except the largest unordered stack s', as well as an additional container to block s'.

Thus, considering that the stack s_r has n_r containers and s' is the largest unordered stack, the algorithm may not terminate if the following conditions are met (nontermination necessary conditions):

- $n_r \geq E' + 1$, where E' is the number of empty slots of all stacks except s_r and s'. $+1$ corresponds to the additional container required to block s'.

- s' is not full (because if it is full, then the top bad-placed container of s' could be well-placed into s_r after reduction);

In Fig. 6-left, s_r has $n_r = 2$ well-placed containers, the unordered stack with maximum height s' corresponds to the second stack, and the number of empty slots of all other stacks except s_r and s' is $E' = 1$. Thus, Condition 2

is met, i.e., $n_r \geq E'+1$. Note that conditions are necessary but not sufficient. We also require that containers in s_r effectively block bad-placed containers of other stacks (e.g., if we change 9 by 5 in the figure, the corresponding container would be unblocked after reducing s_r).

It is important to note that the necessary non-termination conditions are based on the assumption that if the arrangement of containers is identical, then the same stack must be selected for reduction. Therefore, by rotating the selected stacks (e.g., by prioritizing the stack that have been selected fewer times as in Algorithm 4), the algorithm may terminate even if the situation of Fig. 6 is reached for a particular stack s_r.

Proposition 2 (Termination, sufficient condition)**.** *Consider a CPMP with S stacks, H tiers, and N containers. The algorithm* `FRG` *terminates if $N < H(S-2)+3$.*

Proof. The best-case scenario for the second nontermination necessary condition occur when the size of s' is the smallest possible, i.e., $h(s') = 2$. In order to block s', containers from s_r have first to completely fill all the other stacks, then we require at least one additional container in s_r for blocking the stack s' (see Fig. 7).

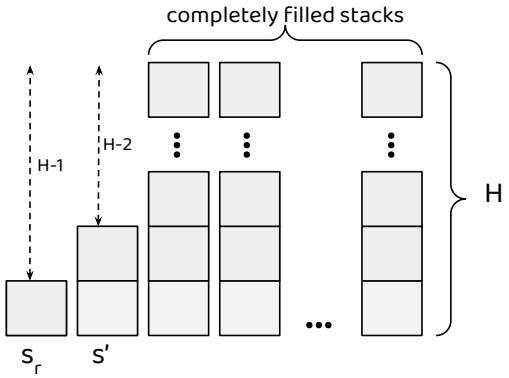


Figure 7: In the example, reducing the stack s_r can block the unordered stack s' with minimal height. Note that such a situation only may occur if the number of empty slots in the layout is lower than or equal to $(H-1)+(H-2)$.

Thus according to the best-case scenario shown in the figure, the non-termination conditions *can only be met* when $E \leq (H-1)+(H-2)$. As $E = S \cdot H - N$, then $N \geq H(S-2)+3$. Otherwise, the algorithm terminates. □

4.3. Advanced Features

In this section, we introduce two techniques to improve the efficiency of the algorithm. Firstly, we present a method for preventing the generation of max-height ordered stacks, which can cause blockages and reduce the effectiveness of the algorithm. Secondly, we describe an assignment procedure that assigns containers to stacks in a more intelligent way, with the goal of keeping them unblocked and avoiding blockages during the stacking process.

Preventing the generation of max-height ordered stacks.

Ordered stacks with $h(s) = H$ (i.e., max-height ordered stacks) cannot be used as auxiliary stacks, and thus, having them increases the chances of blocking other stacks after a reduction. We prevent the generation and decrease the number of such max-height ordered stacks in the following ways:

- (prevention) When there are fewer than M stacks with available slots in the layout, the procedure select_BG_move does not consider moves resulting in a max-height ordered stack;
- (decreasing) When there are fewer than $M/2$ stacks with available slots in the layout, the procedure select_BG_move also considers GG moves that pick up the top container of a max-height ordered stack.

In unreported experiments, we observed that the best results are obtained when the value of M is around the average height of the stacks (i.e., $\frac{N}{S}$); thus, in our implementation, we fixed the value of M to $\frac{N}{S}$.

Keeping containers unblocked

We propose a sophisticated assignment procedure that assigns, in a more intelligent way, the containers of s_r to other stacks. The goal is to keep these containers unblocked; thus, for example, after emptying s_r, they could be placed back *ordered* into the same stack. The method is launched only if the number of nonfull stacks (different from s_r), i.e., stacks with $h(s) < H$, is smaller than the number of containers in s_r. Otherwise, performing the basic reduction moves would be enough to keep these containers unblocked.

Algorithm 5 shows a pseudocode of the procedure. First, the algorithm defines a dictionary or map $\mathcal{A}$ for storing stack assignment s for the containers in s_r. The set $\mathcal{S}_d$ is initialized with feasible destination stacks, and the vector C is initialized with the containers of the stack s_r from bottom to

top. Then, while C is not empty, the procedure `genSeq` generates a sequence of containers $seq = \{c_1, c_2, ...c_n\}$ such that $g(c_1) \geq g(c_2)... \geq g(c_n)$. In this sequence, $i > j$ implies that c_i is below c_j in stack s_r. These conditions allow containers in the sequence to be assigned to a stack without blocking each other. The procedure `genSeq` is explained in detail in page 20.

To guarantee that we can place all containers from the set C into some stack $s_i \in \mathcal{S}_d$, we need to ensure that $|seq|$ is at least $|C| - e(\mathcal{S}_d) + \min_{s_i \in \mathcal{S}_d} e(s_i)$. This condition allows the remaining stacks $\mathcal{S}_d \setminus s_i$ to accommodate up to $|C| - |seq|$ containers that are not part of the sequence $|seq|$. If such a sequence exists (i.e., $|seq| > 0$), the m first elements of the sequence are *assigned*, in reverse order, to a stack $s_d \in S_d$ selected by the procedure select_destination_seq (Lines 8-12). m is the minimum between the number of empty slots in s_d and the size of the sequence, i.e., $\min(\mathrm{e}(s_d), |seq|)$. s_d is removed from $\mathcal{S}_d$, and assigned containers are removed from C. Then, the while-loop is repeated until C is empty or such a sequence does not exist.

The method select_destination_seq selects a stack $s_i \in \mathcal{S}_d$ such that it must be filled by a number of elements less than or equal to the number of elements in the sequence, i.e., $|C| - \sum_{\substack{s_j \in \mathcal{S}_d \\ s_j \neq s_i}} \mathrm{e}(s_j) \leq |seq|$. Stacks with the same number of empty slots as the length of the sequence are prioritized, i.e., $\mathrm{e}(s_i) = |seq|$. Ties are broken by the method `select_destination` with c equal to the first element that would be placed into s_i, i.e., the m-th element in the sequence.

Ultimately, the method returns a map of assignment s $\mathcal{A}$ and a set of destination stacks $\mathcal{S}_d$ that can be used for placing nonassigned containers.

Fig.8 shows an example of the unblocking_assignment procedure. Values above each stack s_i correspond to the minimal size that a sequence must have to be assigned to the stack, i.e., $|C| - \sum_{\substack{s_j \in \{s_1, s_2, s_3\} \\ s_j \neq s_i}} \mathrm{e}(s_j)$. For example, as s_r has 4 containers, a sequence of size 1 or more must be assigned to the stacks s_1 and s_2. The stack s_3 may or may not be filled with an element of s_r.

In the first iteration, the procedure `genSeq` generates a sequence of containers of any size (because the minimal size is 0). This method prioritizes large group values; thus, the generated sequence considers the containers with group values: $\{8, 7, 1\}$. The sequence is assigned by the procedure `select_destination_seq` to the stack s_1. As the sequence size is larger than the number of empty slots, only the first 2 elements are assigned (i.e.,

```
1  unblocking_assignment (L = (s_r, S, −, −))
2      A ← {} # map of assignments
3      S_d ← {s ∈ S, h(s) < H, s ≠ s_r}   # feasible destinations
4      C ← get_containers(s_r)
5      while C is not empty do
6          seq ← genSeq( C, |C| − e(S_d) + min_{s_i ∈ S_d} e(s_i) )
7          if |seq| > 0 then
8              m ← min(e(s_d), |seq|)
9              s_d ← select_destination_seq(seq, S_d, h(s_o))
10             foreach c ∈ seq[m..1] do
11                 A[c] ← s_d
12                 C ← remove c from C
13             S_d ← remove s_d from S_d
14         else
15             break
16     return A, S_d
```

Algorithm 5: Unblocking assignment procedure.

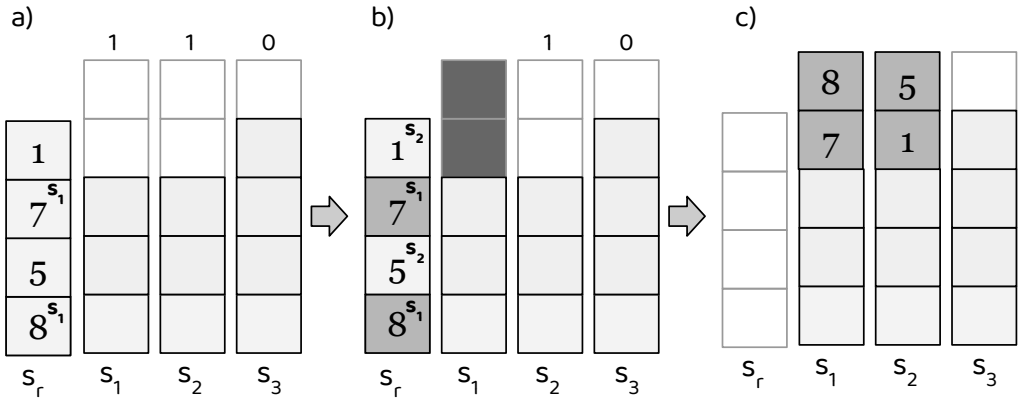


Figure 8: An example of the procedure unblocking_assignment . a), b), and c) show the iterative assignment of the containers to different destinations. d) shows the final layout after applying the corresponding reduction moves.

8 and 7). In the second iteration (see Fig. 8-b), the sequence with group values $\{5, 1\}$ is generated. This sequence is assigned to the stack s_2. After

all containers are assigned, reduction moves are actually applied, resulting in Fig. 8-c.

To incorporate the assignment mechanism into Algorithm 3, we replace the procedure `reduction_move` with a sophisticated variant described by Algorithm 6.

```
1  reduction_move_v2 (L = (s_r, S, A, S_d)):
2      if s_r = none then
3          s_r ← select a stack from S (by frequency, then by height, then by
             g.v. average)
4          if card({s ∈ S \ {s_r}, h(s) < H}) < h(s_r) then
5              A, S_d ← unblocking_assignment (L)
6          else
7              A ← {}; S_d ← S \ {s_r}
8      c ← top(s_r)
9      if c ∈ A then s_d ← A[c]
10     else s_d ← select_destination(c, S_d)
11     return L, (s_r, s_d)
```

Algorithm 6: Stack reducing procedure.

First, note that the method `unblocking_assignment` is called in Line 5 only if the number of nonfull stacks different from s_r is smaller than the number of containers in s_r. Otherwise, no assignment s are considered (i.e., $\mathcal{A}$ is empty). When s_r is not *none*, the top container c of s_r is selected. If the container was previously assigned (i.e., $c \in \mathcal{A}$), then the destination stack s_d is set to the corresponding assignment ; otherwise, the destination is selected by using the standard method `select_destination`. Note that the assignment map $\mathcal{A}$ and the set $\mathcal{S}_d$ are new special variables associated with the layout. They are initialized when s_r is *none*, i.e., when a new stack is reduced, and they are used in the following calls of the procedure `reduction_move_v2` until the stopping reduction criterion is met in the main procedure `FRG`.

Note that it is possible that the method `unblocking_assignment` generates only a few or no assignment s for the containers in s_r (for example, with stacks sorted in reverse order, we can only generate decreasing sequences of size 1). In this case, all destination stacks would be selected by the method `select_destination`, causing blockages. This will result (after refilling s_r

with BG moves) in an ordered stack s_r of a smaller height that may require more reductions, which could possibly arrive to a case of nontermination (see Section 4.2).

Generating a subsequence of size at least min_sz

Algorithm 7 describes the method `genSeq`. This method receives as input a sequence of containers $C = \{c_1, c_2, ..., c_h\}$ and returns a subsequence[4] of size at least min_sz with the nonincreasing *lexicographically largest* group values (e.g., {5, 4, 1} is lexicographically larger than {5, 3, 3}). If such a subsequence does not exist, the method returns an empty subsequence.

The method first computes, for each $i \in \{1, ..., h\}$, the length lds_i of the longest decreasing subsequence (LDS) in C, which starts in c_i. This is done by an implementation of the Patience Sorting [16] algorithm. The algorithm is used to find the length of the longest increasing subsequence (LIS) for the reverse sequence, i.e., $\{c_h, ..., c_1\}$. Patience sorting is iterative, and in each iteration i, it computes the length of the LIS of $\{c_h, ..., c_i\}$, which ends in c_i. This is equivalent to the LDS of C, which starts in c_i. Thus, the vector lds is computed in one call of Patience Sorting, i.e., in time $O(h \log h)$.

```
1 genSeq (C, min_sz)
2     lds ← LDS(C)
3     if lds_1 < min_sz then return {}
4     seq ← {}; C' ← sort(C); i' ← −1
5     for c_i ∈ C' do
6         if i > i' and lds_i + size(seq) ≥ min_sz then
7             seq ← append(seq, c_i); i' ← i
8     return seq
```

Algorithm 7: Sequence generator.

If the LDS of the sequence C is lower than min_sz (i.e., $lds_1 < min\_sz$), the method immediately returns an empty sequence. Otherwise, we construct the lexicographically largest subsequence of size at least min_sz. First, a list of containers C' sorted by decreasing group values is generated. Then, to

[4] A subsequence is a sequence that can be derived from C by deleting some or no elements *without changing the order of the remaining elements.*

generate a *decreasing subsequence*, we start with an empty sequence *seq* and take elements from the list C'.

In each iteration, the element $c_i \in C'$ maximizing $g(c_i)$ is appended to the sequence if (1) c_i is after the last element of the sequence $c_{i'}$ in the original sequence C (i.e., $i > i'$), and (2) the size of the current sequence *seq* plus the length of the LDS starting from c_i is larger than or equal to min_sz.

Note that in each iteration of the procedure `genSeq`, we select the first feasible element maximizing its group value; thus, the resulting sequence is lexicographically the largest.

Theorem 2. *(Execution time of FRG) Considering the advanced features, an iteration of* `FRG` *runs in amortized time* $O(S^2 + H \log H)$.

The proof is in Appendix A.

5. The beam search approach

As in [23], we implemented a beam search algorithm `BS-FRG`, which, in our case, uses the greedy algorithm `FRG` for evaluating candidate layouts. Algorithm 8 shows the procedure. `BS-FRG` maintains a set $\mathcal{B}$ of at most nb layouts (beams). The set is initialized in Line 2 with the initial layout L. In each iteration, for each layout L' in $\mathcal{B}$, an set of *candidate* layouts $\mathcal{C}$ is generated by performing different moves, all of them starting from L' (function generate_candidates). Each candidate is evaluated by using the algorithm FRG, which generates a feasible layout L''. The length of the path (sequence of moves) from the initial layout L to the feasible layout L'' corresponds to the evaluation of the candidate (lower is better). The best nb candidates are kept for the next iteration of the beam search, and the process is repeated. At the end of the algorithm, the best solution found by FRG among *all* the evaluated candidates is returned.

The function generate_candidates generates each candidate layout by applying one move to the current layout L'. We define two different types of moves: **single moves** and **compound moves**. The single moves consider all moves $(s_o, s_d) \in \mathcal{S}^2$, such that $s_o \neq s_d$, $h(s_o) > 0$, and $h(s_d) < H$. Note that the size of the set is $O(S^2)$. To reduce this size, we limit the number of moves by the origin stack, i.e., for each stack s_o, we select at most k destination stacks s_d generating at most k moves (s_o, s_d). The k *most promising* destinations are chosen by using the procedure `select_destination`. In this way, the size of the set of single moves is reduced to $O(kS)$. We consider

```
1 BS-FRG (L = (s_r, S, A, S_d), nb)
2 |  B ← {L} # the beams of beam search
3 |  while B is not empty do
4 |  |  C ← {} # set of candidates
5 |  |  foreach L' in B do
6 |  |  |  C ← C ∪ generate_candidates(L')
7 |  |  B ← best nb layouts in C # according to FRG
8 |  return the best solution found by FRG
```

Algorithm 8: The `BS-FRG` algorithm.

an additional single move m' that consists in performing one iteration of the algorithm `FRG` starting from the current layout L'. Note that this movement seems to be redundant; however, unlike the *seemingly equivalent* single move having the same origin and destination containers as m', performing an iteration of `FRG` may change the value of the internal variables of the layout related to the reduction moves, i.e., s_r, $\mathcal{A}$ and $\mathcal{S}_d$, and thus the evaluation of the *seemingly equivalent* related candidates may differ. It is important to note that whenever a simple move is performed, the smart assignation procedure is terminated, i.e., s_r is set to **none**.

The set of compound moves is defined by $\{R_1, R_2, ..., R_S\}$. R_i corresponds to performing a reduction of the stack s_i by applying reduction moves until the stopping criterion is met for the stack. By using these sets, we defined two beam search strategies: `BSs-FRG`, which only uses single moves for generating candidates, and `BS*-FRG`, which uses both sets of moves for generating candidates. Strategies are parameterized with the number of beams nb and the number of destinations per stack k in the set of single moves.

The beam search algorithm proposed in [23] uses a more sophisticated procedure for generating and evaluating candidate layouts. Instead of applying a set of moves, it performs a tree search with limited width and depth. The leaves of this tree are evaluated with the underlying greedy algorithm (TGH), and the best reached costs (i.e., the minimum number of container moves) are associated with the first layout of the corresponding paths. The best candidates are kept for the next iteration, and the process is repeated. The algorithm has three configurable parameters: probing tree depth (td), probing tree width (tw), and beam width (bw).

6. Experiments

For the experiments, we implemented `FRG` and `BS-FRG` in C++. The experiments were run on a PowerEdge T420 server with 2 quadprocessor Intel Xeon, 2.20 Ghz and Ubuntu Linux operating system. The `BS-FRG` has two configurable parameters: nb and k.

We compare our approaches with the state-of-the-art greedy constructive algorithms. They are the target-guided heuristic (TGH) [23], the feasibility-based heuristic FBH [24], and the multiheuristic proposed in [13]. We also include BS-B, a beam search approach that uses the greedy algorithm TGH for evaluating states; a biased random-key genetic algorithm (BRKGA) proposed in [11]; and DLTS, a tree search algorithm which makes decisions by using a deep neural network [10]. For BRKGA, we report results from [11]. BRKGA was implemented in C#, and the experiments were carried out on a cluster of Intel Xeon E5-2670 processors using one single thread per execution [11]. For DLTS, the results are reported from [10]. DLTS was implemented in Python 3 using Keras 1.1.0 [14] with Theano 0.8.2 [1] as the backend for the implementation of the deep neural networks. While neural networks were trained using six cores for a few days, the results reported on the tables use a single core. The results of the rest of the competitor algorithms are reported in [24]. Codes were written in Java, and the experiments were carried out on a computer with an Intel Core i7 3.40 GHz and Windows 7 operating system.

Related to adjustable parameters, it is worth noting that the FBH greedy algorithm has two parameters (the number of protected tiers P1 and a threshold value P2). Experimentally, the authors observed that the best parameter values are $P1 = \frac{1}{3}H$ and $P2 = \frac{1}{4}N$ for CVS instances and $P1 = \frac{1}{2}H$ and $P2 = \frac{1}{6}N$ for BF instances. BS-B has three configurable parameters: probing tree depth (td), probing tree width (tw), and beam width (bw). In the final experiments, the authors fix these values to $bw = tw = 20$ and $td = 3$ [23]. The BRKGA has 8 adjustable parameters. They were adjusted by using the algorithm configurator GGA [2] 3 times for 72 wall-clock hours, allowing each execution of GGA to use up to 16 cores simultaneously for tuning. DLTS use two artificial neural networks. In addition to the hyperparameters of the networks, the algorithm has another 3 adjustable parameters: bounding network query frequency, lower bound uncertainty adjustment, and branch pruning adjustment. The training set for the networks was generated by using more than 900,000 reference solutions. Each solution was generated by

an iterative deepening search [19] limited from 10 to 30 minutes (7 days of solving). Networks were trained using 6 cores, resulting in a training time ranging from several hours to a few days [10] for each model. The resulting networks contain 60,000 to 800,000 weights.

We test our algorithm on the CVS and BF datasets. The CVS instances, proposed by [5], are classified into 21 groups, each consisting of 40 instances. For any instance, the heights of stacks in the initial layout are $h(s_i) = H - 2$; hence, $N = S(H - 2)$. The BF instances, which were proposed by Bortfeldt and Forster in [4], consist of 32 instance sets. Each group consists of 20 instances. In the BF instances, the size of the layout is $S = 16$ or 20 and $H = 5$ or 8. The number of containers N is either $0.6 \cdot S \cdot H$ or $0.8 \cdot S \cdot H$, the number of Groups G is either $0.2 \cdot N$ or $0.4 \cdot N$, and the number of disorderly containers B is either $0.6 \cdot N$ or $0.75 \cdot N$ in the initial layout. It is interesting to note that all BF instances and most of the CVS instances meet the termination condition[5].

6.1. Setting k and the beam size (nb)

Plots in Fig. 9 report results for our beam search with different values for the parameters nb and k compared with the best results reported by BS-B and BRKGA. The plot shows the average cost (i.e., number of container moves) of the best solution found as a function of the beam size of `BSs-FRG` and `BS*-FRG` with different values of k. BRKGA reports results for 19 of the 21 CVS instance groups; thus, to perform the comparison, the CVS plot considers the same groups.

First, note that for the same value of k, `BS*-FRG` outperforms `BSs-FRG` on average. `BS*-FRG` also outperforms BRKGA and BS-B when nb is 100 or more. The value of k has a significant impact on the results. When all feasible moves are considered in the set of single moves (i.e., $k = S$), we can appreciate a slight improvement in the results reported by the CVS instances. However, considering all feasible moves has a large impact on CPU time (strategies require between 2 and 5 times more time than using $k = 3$). Thus, in further experiments, we set the value of k to 3 in CVS instances. Related to BF instances, the best results are reported when $k = 1$. This may be because, as BF instances have a larger number of stacks, when the value of k increases, many moves are considered for each state in the beam search

[5]Groups 3-3, 4-4, 5-4, 5-5, 6-6, 10-6, and 10-10 do not meet this condition.

(e.g., 60 when $S = 20$ and $k = 3$). This may be counterproductive due to the reduced proportion of candidates that are selected (e.g., 1 of every 60). Candidates are selected exclusively by their evaluations, which can reduce the heterogeneity in the selected set. In further experiments, we set the value of k to 1 in BF instances.

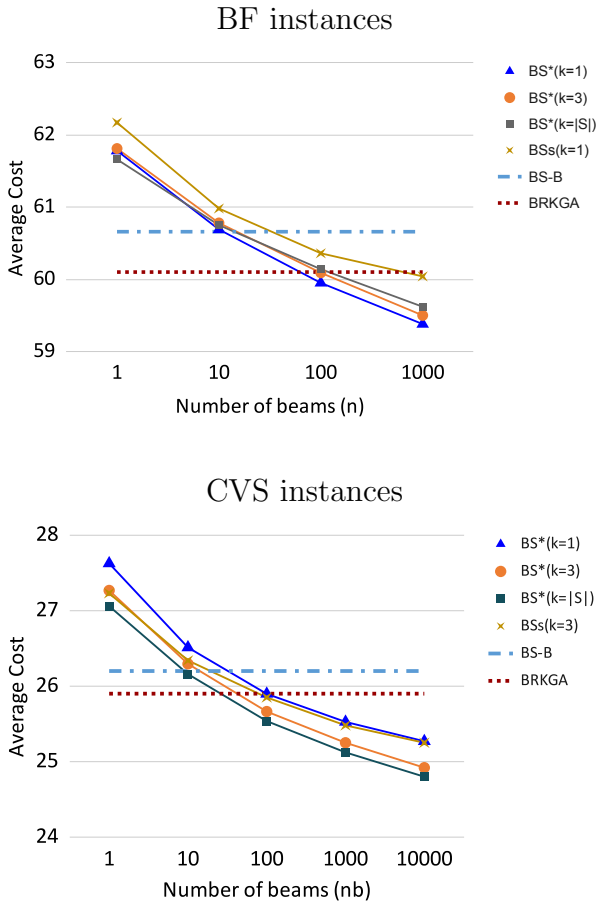


Figure 9: Results reported by `BS-FRG` with different beam sizes and values of k.

To set an appropriate number of beams nb, we study our most similar competitor, that is, the beam search algorithm BS-B [23]. BS-B uses a beam width of 20. However, children for each beam are generated by invoking the greedy algorithm TGH on each leaf in a limited search tree (width=20 and depth=3). Thus, BS-B may perform up to $20 \cdot 20^3 = 64000$ TGH calls in each iteration of the beam search. In our case, for each beam, we generate

$kS+S+1$ children at most; then, `BS*-FRG` may perform up to $nb \cdot (kS+S+1)$ `FRG` calls in each level of the beam search. Considering that $nb = 1000$, $k = 3$, and $S = 16$ (i.e., a large instance), `BS*-FRG` would perform up to 65000 calls in each level of the search tree. Thus, we consider that $nb = 1000$ is a reasonable beam size for comparing our approach and BS-B in a relatively fair way.

					Greedy algorithms						
Instances	S	H	N	B	TGH	Multi	FBH	FRG	BS-B	BRKGA	BS*-FRG
BF1	16	5	48	29	**29.1**	30.2	29.2	29.2	**29.1**	**29.1**	**29.1**
BF2	16	5	48	36	**36.0**	37.1	**36.0**	**36.0**	**36.0**	**36.0**	**36.0**
BF3	16	5	48	29	29.5	31.2	29.4	**29.3**	**29.1**	**29.1**	**29.1**
BF4	16	5	48	36	**36.0**	37.5	36.2	**36.0**	**36.0**	**36.0**	**36.0**
BF5	16	5	64	39	48.3	48.0	46.3	**45.3**	41.4	41.3	**41.1**
BF6	16	5	64	48	57.7	56.4	55.5	**54.6**	50.2	49.9	**49.2**
BF7	16	5	64	39	53.5	51.0	50.0	**47.4**	43.1	42.5	**42.1**
BF8	16	5	64	48	60.1	58.6	57.6	**56.0**	51.2	50.7	**49.7**
BF9	16	8	77	47	60.4	59.6	56.3	**53.8**	50.4	50.6	**50.3**
BF10	16	8	77	58	62.1	65.6	61.6	**59.9**	**58.8**	58.9	**58.8**
BF11	16	8	77	47	61.1	61.5	55.0	**54.2**	51.2	51.2	**50.7**
BF12	16	8	77	58	63.5	68.2	61.5	**60.2**	58.7	58.9	**58.4**
BF13	16	8	103	62	107.5	91.0	96.5	**88.4**	75.4	74.7	**74.2**
BF14	16	8	103	78	125.3	**106.8**	116.1	107.7	93.1	91.0	**89.2**
BF15	16	8	103	62	110.9	96.3	99.5	**89.7**	78.7	76.4	**74.4**
BF16	16	8	103	78	133.3	112.1	115.5	**109.4**	93.6	92.3	**90.2**
BF17	20	5	60	36	**36.5**	39.0	36.6	36.6	**36.3**	**36.3**	**36.3**
BF18	20	5	60	45	**45.0**	46.4	45.2	**45.0**	**45.0**	**45.0**	**45.0**
BF19	20	5	60	36	36.8	39.9	36.8	**36.6**	**36.5**	**36.5**	**36.5**
BF20	20	5	60	45	**45.0**	47.0	45.1	**45.0**	**45.0**	**45.0**	**45.0**
BF21	20	5	80	48	60.8	60.4	56.6	**55.8**	51.6	**51.3**	**51.3**
BF22	20	5	80	60	68.9	70.5	65.6	**66.0**	61.8	61.3	**60.9**
BF23	20	5	80	48	60.9	61.7	**55.3**	56.5	51.0	50.9	**50.7**
BF24	20	5	80	60	71.0	72.7	68.0	**66.3**	62.1	61.8	**60.9**
BF25	20	8	96	58	70.0	73.3	68.0	**64.4**	61.5	61.6	**61.3**
BF26	20	8	96	72	74.4	81.1	75.8	**72.9**	72.4	72.5	**72.3**
BF27	20	8	96	58	71.8	73.4	65.7	**64.4**	61.9	61.8	**61.4**
BF28	20	8	96	72	76.1	83.6	76.5	**73.5**	72.7	72.8	**72.6**
BF29	20	8	128	77	120.6	113.2	115.9	**104.3**	92.1	89.9	**88.6**
BF30	20	8	128	96	143.1	128.8	129.6	**125.0**	110.3	107.0	**105.5**
BF31	20	8	128	77	128.2	116.9	115.9	**105.9**	94.0	91.1	**89.3**
BF32	20	8	128	96	147.3	133.2	134.1	**126.0**	111.8	108.5	**106.5**
Av.1-8	16	5	56	38	43.8	43.7	42.5	**41.7**	39.5	39.3	**39.0**
Av.9-16	16	8	90	61.3	90.5	82.6	82.7	**77.9**	70.0	69.3	**68.3**
Av.17-24	20	5	70	47.3	53.1	54.7	51.1	**51.0**	48.6	48.5	**48.3**
Av.25-32	20	8	112	75.8	103.9	100.4	97.7	**92.0**	84.6	83.2	**82.2**
Average	18	6.5	82	55.6	72.8	70.4	68.5	**65.6**	60.7	60.1	**59.4**
Av.time(s)					2.08ms	27.4ms	4.71ms	0.50ms	118.9	142.8	18.3

Table 1: Results for the BF instances. In black, we highlight both the best results for the greedy algorithms and the best results for the population-based strategies.

6.2. Comparing `FRG` *with other greedy algorithms*

Tables 1 and 2 report the results of different strategies on the BF and CVS instance sets, respectively. In each table, the first column identifies the instance set, and the following four columns provide the instance characteristics (number of stacks, height, number of containers, and average number of badly placed containers). Then, we report the average costs (i.e., average number of container moves) reached by different greedy strategies, including the proposed `FRG`. The last row shows the average time in seconds (milliseconds for the greedy algorithms) reported by the strategies. In the last columns, we compare the results reported by state-of-the-art strategies (BS-B, BRKGA, and DLTS) and our beam search algorithm `BS*-FRG`. Note that DLTS reports results only for some CVS groups.

Related to the BF instances and *greedy algorithms*, note that `FRG` offers the best results in almost every instance set. In these instances, there are no differences between using and not using the advanced features described in Section 4.3. Additionally, conditions for applying sophisticated mechanisms are rarely met in these instances because they have numerous stacks.

Related to the CVS instances, there are important differences between applying `FRG` without the sophisticated mechanism (`FRG`$^-$) and the variant including these mechanisms (`FRG`). In particular, `FRG` significantly outperforms FRG$^-$ on instances not meeting the termination condition (e.g., 3-3, 5-4, 5-5, 6-6, and 10-6). On the other instances, both algorithms report comparable results. The asterisk (*) in instance 10-6, related to FRG$^-$, indicates that the strategy failed in solving one of the instances of this group; thus, the average corresponds to 39 of the 40 instances.

Related to greedy algorithms, `FRG` outperforms the other algorithms in most of the instances and reports the best average cost. Note that in very large instances (e.g., 10-6, 10-10), `FRG` significantly outperforms other strategies (e.g., it is 18% better than FBH in the instance 10-6). In small instances (i.e., 3-x), FBH reports better results on average.

6.3. Comparing the approach with state-of-the art algorithms

For the BF instances (Table 1), `BS*-FRG` (with $nb = 1000$) offers the best results in every group, reducing the average cost reached by BS-B by 1.3 and the average cost reached by BRKGA by 0.7. Related to the CVS instances (Table 2), `BS*-FRG` (with $nb = 1000$) also outperforms the costs reached by BS-B on every group of instances. For the comparison with BRKGA and

Instances	S	H	N	B	Greedy algorithms					BS-B	BS*-FRG $nb = 1000$	BRKGA	BS*-FRG $nb = 10000$	DLTS
					TGH	Multi	FBH	FRG$^-$	FRG					
3-3	3	5	9	4.3	13	**11.3**	**11.3**	13.3	11.7	9.4	**8.9**	9.6	**8.8**	
3-4	4	5	12	5.4	12.2	11	**10.8**	11.3	11.3	9.5	**9.1**	9.2	**9.1**	
3-5	5	5	15	6.8	12.8	12.2	**12.1**	12.3	12.3	10.5	**10.2**	10.3	**10.2**	10.3
3-6	6	5	18	8.2	14.4	14	**13.0**	13.1	13.1	11.6	**11.3**	11.5	**11.3**	
3-7	7	5	21	9.7	16	15.8	14.8	**14.6**	**14.6**	13.1	**12.9**	**12.9**	**12.9**	12.9
3-8	8	5	24	10.5	16.6	16.9	15.7	**15.6**	**15.6**	13.9	**13.5**	13.9	**13.5**	
4-4	4	6	16	9.1	23.4	**20.4**	22.8	21.4	21.5	17.0	**16.2**	16.3	**16.0**	
4-5	5	6	20	12.1	26.7	23.8	23.1	22.9	**22.7**	18.9	**18.2**	18.8	**18.1**	17.9
4-6	6	6	24	13.7	27.6	25.9	24.8	**24.7**	**24.7**	20.3	**19.7**	20.3	**19.6**	
4-7	7	6	28	16.2	29.9	29.6	27.6	**27.4**	**27.4**	23.2	**22.4**	22.7	**22.1**	22.1
5-4	4	7	20	13	44.8	31.8	35.1	35.4	**33.6**	26.4	**24.7**	27.0	**24.4**	
5-5	5	7	25	15.9	42.4	33.6	35.3	35.9	**35.1**	27.4	**26.5**	27.9	**25.9**	25.1
5-6	6	7	30	20.2	50.6	40.7	39.9	**39.2**	39.5	32.1	**31.1**	32.0	**30.7**	
5-7	7	7	35	23.1	48.8	44.4	41.7	**41.4**	41.5	34.2	**33.4**	33.8	**32.8**	32.0
5-8	8	7	40	26.8	56.7	50.5	47.5	45.7	**45.2**	38.6	**37.4**	37.8	**36.8**	
5-9	9	7	45	30.2	57.5	55.3	50.5	**49.5**	49.7	42.2	**40.7**	41.1	**40.2**	
5-10	10	7	50	33.1	62.8	59.6	54.6	**52.1**	52.3	44.9	**43.6**	44.0	**42.9**	42.0
6-6	6	8	36	25.3	74.3	54	55.2	53.9	**52.6**	43.8	**41.8**	44.3	**40.9**	
6-10	10	8	60	42.6	88.6	79.6	75.6	70.4	**70.3**	60.6	**58.4**	59.4	**57.5**	
10-6	6	12	60	50.2	332.3	150.3	140.6	181.3*	**117.9**	116.2	**100.8**		99.6	
10-10	10	12	100	83.1	302.9	181.6	179.2	161.1	**159.9**	150.2	**139.8**		138.0	
Av	6.5	6.8	32.8	21.9	64.5	45.8	44.3	44.9	**41.5**	36.4	**34.3**	25.9*	**24.9***	
Av.time(s)					0.21ms	6.47ms	0.58ms	0.13ms	0.14ms	16.9	5.1	96.3	50.1	50.0

Table 2: Results for the CVS instances. In black, we highlight the best results for the greedy algorithms; BS-B vs. BS*-FRG ($nb = 1000$) and BRKGA vs. BS*-FRG ($nb = 10000$).

to have more comparable CPU times, we increased nb to 10000. `BS*-FRG` outperforms BRKGA in 18 of the 19 compared groups.

DLTS reaches the best results in 4 of the 7 CVS groups on which the strategy reports results. Recall, however, that `BS*-FRG` is a simpler algorithm with only one adjustable parameter (k). On the other hand, DLTS is a very sophisticated strategy that uses 2 neural networks for making decisions and requires a specific training for each kind of instance. The results in the table were reported by DLTS using 3 different trained models where the training instances considered 5, 7, and 10 stacks.

6.4. Comparing the approach with exact methods

The best *exact approach* for solving the CPMP is the branch and bound algorithm PT^A proposed in [20]. This strategy, running for an hour in CPU time on a server with Intel Xeon X5650 CPUs at 2.67 GHz and CentOS 6.6 operating system, is able to optimally solve 73.1% (468/640) of the BF instances and 81.5% (684/840) of the CVS instances.

Considering only these instances, `FRG` optimally solves 38.6% (181/468) of the BF instances and 7.9% (54/684) of the CVS instances. In turn, `BS*-FRG` (with $nb = 1000$) optimally solves 82.5% (386/468) of the BF instances and 52.3% (358/684) of the CVS instances (58.6% when $nb = 10000$). In BF instances, `BS*-FRG` reports an average gap to optimal solutions of 0.45%. In CVS instances, the average gap is 2.9%.

Although `BS*-FRG` finds a large number of optimal solutions, we think that its major potential is related to solving difficult CPMP instances. PT^A is generally not able to solve instance sets with $H > 7$ even in one day of CPU time [20]; however, `BS-FRG` reports very good quality solutions on these instances in just approximately 30 seconds of CPU time.

7. Conclusions

In this work, we propose `FRG`, a greedy heuristic for solving the CPMP. It is based on only two kinds of moves: BG moves that fill already ordered stacks and moves for reducing stacks. In addition to being simple, the algorithm is also fast: each container placement has a worst-case quadratic time complexity in the number of stacks (assuming that $S^2 \leq H \log H$). In most of the classical benchmark sets, the simple version (without the advanced features) terminates and reports very competitive results.

Compared to other greedy heuristics (in particular target-guided ones), `FRG` outperforms them in most large instances of classical benchmark sets. When embedded in a beam search algorithm (`BSG*-FRG`), the strategy reports the best results compared to traditional strategies (that do not incorporate machine learning). On the other hand, while exact algorithms perform very well on small and medium size instances (e.g., when $H \leq 7$), they generally are not able to solve large instances even in one day of CPU time [20]. Thus, `BSG*-FRG` becomes an interesting alternative for finding quality solutions quickly in large CPMP instances.

In future work, we plan to use `FRG` for finding feasible solutions (upper bounds) in tree-search-based exact algorithms. Computing upper bounds is important in exact methods because we may discard nodes that cannot reach better solutions (i.e., nodes such that the lower bound is larger than the upper bound) and thus reduce the search space. Improving upper bounding techniques generally implies a faster convergence to the optimal solution.

Declarations

All authors declare that they have no conflicts of interest.

Appendix A. Lemmas and proofs

Lemma 1. *The method* `select_destination` *runs in time* $O(S)$.

Proof. In our implementation, stacks are implemented by using arrays and maintaining an auxiliary counter variable for storing the number of ordered items in each stack (i.e., the number of consecutive items with nonincreasing group values from the bottom to the top of the stack). Each time a move involving the stack is done, the related counter is accordingly updated in $O(1)$ execution time. Using this variable, knowing if a stack is ordered is also performed in $O(1)$ time (we only need to compare the counter with the number of items currently in the stack; if it is equal, then the stack is ordered; otherwise, it is unordered). Taking this into account, we note that the method `select_destination` has a time complexity of $O(S)$ because each set (i.e., V,XG_d, XB_d^1, and XB_d^2) can be generated in linear time in the number of stacks and the calls to the method arg min can also be performed in linear time in the size of these sets. □

Lemma 2. *The method* `select_BG_move` *has an execution time of* $O(S^2)$.

Proof. The method has an execution time of $O(S^2)$ because it needs to compare group values for each valid BG move (which are at most S^2), and identifying BG moves is $O(1)$ for each move because we simply need to verify that s_o is unordered and s_d is ordered (which is $O(1)$ according to the proof of Lemma 1). □

Lemma 3. *The method* `reduction_move` *is performed in* $O(S)$ *time.*

Proof. Computing the height $h(s)$ of a stack is $O(1)$. Considering that each stack maintains the up-to-date sum of group values of its containers, computing the average group values is also $O(1)$. Then, the selection of the stack is performed in $O(S)$. As the method `select_destination` performs in $O(S)$ time (according to Lemma 1), then the method `reduction_move` also has a time complexity of $O(S)$. □

Lemma 4. *The method stopping_reduction_criterion runs in time* $O(S)$.

Proof. Consider that each container c stores the number ub_c of unblocked bad-placed containers of its corresponding stack without taking into account the containers above c. This number can be updated in $O(1)$ time each time a move is performed.[6] Then, $ub(s)$, i.e., ub_c of the top container of s, can also be retrieved in $O(1)$. Finally, each condition of the method stopping_reduction_criterion has an execution time of $O(S)$ because they are required to perform $O(1)$ instructions on, at most, each stack in the layout. □

Proof of Theorem 1. The execution time of each iteration of the `FRG` algorithm is dominated by the procedure `select_BG_move`, which, according to Lemma 2, is $O(S^2)$. □

Lemma 5. *The method* `genSeq` *runs in time* $O(h \log h)$, *with* $h = |C|$

Proof. Note that the time complexity of the while-loop of the method is linear in h. The method LDS is $O(h \log h)$, and if we use a classical $O(h \log h)$ sorting algorithm for generating C', then the whole method also runs in $O(h \log h)$ time. □

[6] Considering that a container c is placed above a container c', if the related move is XB and $g(c) \geq g(c')$, then $ub_c \leftarrow ub_{c'} + 1$; if the move is XB and $g(c) < g(c')$, then $ub_c \leftarrow 1$; and if the move is XG, then $ub_c \leftarrow 0$.

Lemma 6. *The amortized time of assigning (or not) one container by using the method* `unblocking_assignment` *is* $O(S + H \log H)$.

Proof. The method `unblocking_assignment` , in the worst case, performs $|C| = h$ calls to the method `genSeq` (each one resulting in a sequence of size $m = 1$), which, according to Lemma 5, is $O(h \log h)$. As $m = 1$, the execution time of the if-block in the method `unblocking_assignment` (Lines 7-15) is dominated by the method select_destination_seq, which, similar to the method `select_destination` (Lemma 1), runs in time $O(S)$. Thus, the while-loop runs in worst-case time $O(h \cdot (S + h \log h)) \leq O(h \cdot (S + H \log H))$, with h being the number of elements to be assigned (or not). Finally, the amortized time of assigning (or not) one container is $O(S + H \log H)$. □

Lemma 7. *Each reduction move is performed by the method* `reduction_move_v2` *in* $O(S + H \log H)$ *amortized time.*

Proof. The method `reduction_move_v2`, in the worst case, calls `unblocking_assignment` first and, in further iterations, places the containers of s_r in other stacks by using the method `select_destination`. According to Lemma 6, assigning (or not) one container runs in amortized time $O(S + H \log H)$. According to Lemma 1, selecting its destination runs in time $O(S)$. Thus, each reduction move is performed in $O(S + H \log H)$ amortized time. □

Proof of Theorem 2. The execution time of the `FRG` algorithm is dominated by the methods `select_BG_move`, which is $O(S^2)$ (Lemma 2), and `reduction_move_v2`, which is $O(S + H \log H)$ (Lemma 7). □

References

[1] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al., Theano: A Python framework for fast computation of mathematical expressions, arXiv e-prints (2016) arXiv–1605.

[2] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: International Conference on Principles and Practice of Constraint Programming, Springer, 142–157, 2009.

[3] S. Boge, M. Goerigk, S. Knust, Robust optimization for premarshalling with uncertain priority classes, European Journal of Operational Research 287 (1) (2020) 191–210.

[4] A. Bortfeldt, F. Forster, A tree search procedure for the container premarshalling problem, European Journal of Operational Research 217 (3) (2012) 531–540.

[5] M. Caserta, S. Voß, M. Sniedovich, Applying the corridor method to a blocks relocation problem, OR spectrum 33 (4) (2011) 915–929.

[6] M. M. da Silva, S. Toulouse, R. W. Calvo, A new effective unified model for solving the pre-marshalling and block relocation problems, European Journal of Operational Research 271 (1) (2018) 40–56.

[7] C. Expósito-Izquierdo, B. Melián-Batista, M. Moreno-Vega, Premarshalling problem: Heuristic solution method and instances generator, Expert Systems with Applications 39 (9) (2012) 8337–8349.

[8] M. Gheith, A. B. Eltawil, N. A. Harraz, Solving the container premarshalling problem using variable length genetic algorithms, Engineering Optimization 48 (4) (2016) 687–705.

[9] M. S. Gheith, A. B. Eltawil, N. A. Harraz, A rule-based heuristic procedure for the container pre-marshalling problem, in: 2014 IEEE International Conference on Industrial Engineering and Engineering Management, IEEE, 662–666, 2014.

[10] A. Hottung, S. Tanaka, K. Tierney, Deep learning assisted heuristic tree search for the container pre-marshalling problem, Computers & Operations Research 113 (2020) 104781.

[11] A. Hottung, K. Tierney, A biased random-key genetic algorithm for the container pre-marshalling problem, Computers & Operations Research 75 (2016) 83–102.

[12] C. Jiménez-Piqueras, R. Ruiz, C. Parreño-Torres, R. Alvarez-Valdes, A constraint programming approach for the premarshalling problem, European Journal of Operational Research .

[13] R. Jovanovic, M. Tuba, S. Voß, A multi-heuristic approach for solving the pre-marshalling problem, Central European Journal of Operations Research 25 (1) (2017) 1–28.

[14] N. Ketkar, Introduction to keras, in: Deep learning with Python, Springer, 97–111, 2017.

[15] Y. Lee, N.-Y. Hsu, An optimization model for the container pre-marshalling problem, Computers & operations research 34 (11) (2007) 3295–3313.

[16] C. L. Mallows, Patience sorting, SIAM review 5 (4) (1963) 375.

[17] C. Parreño-Torres, R. Alvarez-Valdes, R. Ruiz, Integer programming models for the pre-marshalling problem, European Journal of Operational Research 274 (1) (2019) 142–154.

[18] M. Prandtstetter, A dynamic programming based branch-and-bound algorithm for the container pre-marshalling problem, Technical repot, IT Austrian institute of technology .

[19] S. Tanaka, K. Tierney, Solving real-world sized container pre-marshalling problems with an iterative deepening branch-and-bound algorithm, European Journal of Operational Research 264 (1) (2018) 165–180.

[20] S. Tanaka, K. Tierney, C. Parreño-Torres, R. Alvarez-Valdes, R. Ruiz, A branch and bound approach for large pre-marshalling problems, European Journal of Operational Research 278 (1) (2019) 211–225.

[21] K. Tierney, D. Pacino, S. Voß, Solving the pre-marshalling problem to optimality with A* and IDA, Flexible Services and Manufacturing Journal 29 (2) (2017) 223–259.

[22] M. van Brink, R. van der Zwaan, A branch and price procedure for the container premarshalling problem, in: European Symposium on Algorithms, Springer, 798–809, 2014.

[23] N. Wang, B. Jin, A. Lim, Target-guided algorithms for the container pre-marshalling problem, Omega 53 (2015) 67–77.

[24] N. Wang, B. Jin, Z. Zhang, A. Lim, A feasibility-based heuristic for the container pre-marshalling problem, European Journal of Operational Research 256 (1) (2017) 90–101.

[25] R. Zhang, Z.-Z. Jiang, W. Y. Yun, Stack pre-marshalling problem: a heuristic-guided branch-and-bound algorithm., International Journal of Industrial Engineering 22 (5).

Noname manuscript No.
(will be inserted by the editor)

Non-Convex Optimization: Using Preconditioning Matrices for Optimally Improving Variable Bounds in Linear Relaxations

Victor Reyes[1] · **Ignacio Araya**[1]

Received: date / Accepted: date

Abstract The performance of branch-and-bound algorithms for solving non-convex optimization problems greatly depends on convex relaxations techniques. They generate convex regions which are used for improving the bounds of variable domains. In particular, convex polyhedral regions can be represented by an underdetermined system $A.x = b$. Then, bounds of variable domains can be improved by minimizing and maximizing variables in the linear system. Reducing optimally variable domains in underdetermined linear systems, however, is an expensive task. It requires to solve up to two linear programs for each variable (one for each variable bound).

Sub-optimal strategies, like preconditioning, may offer satisfactory approximations of the optimal reduction at a lower cost. In underdetermined systems, a preconditioner P can be chosen such that $P.A$ is close to a diagonal matrix. Thus, the projection of the equivalent system $P.A.x = P.b$ over x, by using an iterative method like Gauss-Seidel, can be significantly improved.

In this paper, we show how to generate an *optimal* preconditioner, i.e., a preconditioner that helps the Gauss-Seidel method to reduce *optimally* the variable domains. Despite the cost of generating the preconditioner, it can be re-used in sub-regions of the search space without losing too much effectiveness. Experiment results show that, when used for contracting domains in random generated underdetermined linear systems, the approach is significantly more effective than pseudoinverse and Gauss elimination techniques.

Acknowledgements Ignacio Araya is supported by Fondecyt project 1200035.

Victor Reyes
E-mail: vareyes.cs@gmail.com

Ignacio Araya
Tel.: +56-322273765
E-mail: ignacio.araya@pucv.cl

[1] Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile

1 Introduction

The performance of branch-and-bound (B&B) techniques for solving global optimization (or constraint satisfaction) greatly depends on convex relaxations techniques. These techniques generate a convex and generally polyhedral region which certainly contains the optimal solution of the problem. A polyhedral region can be represented by an *underdetermined linear system* $A.x = b$ (i.e., $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$), where x includes some variables of the original problem and auxiliary variables with an unbounded bound for the inequalities. Once the relaxation is generated, the objective is to reduce (or contract) the variable domains of the original problem. Lower and upper bounds of variable domains can be found by minimizing and maximizing each variable of the linear system. The method, that solves the 2n linear programs, is called optimization-based bound tightening (OBBT [1]) or PolytopeHull [2] and it is used by several global optimization solvers such as αBB [3], ANTIGONE [4], Couenne [5], LaGO [6], SCIP [7] and IbexOpt [8,2]. Due to its expensiveness, OBBT is mostly applied at the root node and within the search tree only with limited frequency or based on its success rate. ANTIGONE, for instance, measures the success of OBBT by the reduction of the box volume and disables it for all children nodes once the rate of reduction drops below a given threshold. The method is expensive and some improvements has been proposed in order to: (1) reduce the number of linear programs to be solved; (2) accelerate the convergence of the simplex algorithm and; (3) generate projection inequalities which approximate the contraction performed by the 2n linear programs [9].

In this work, we deal with the same problem as OBBT, i.e., we want to improve the domain bounds of a variable vector x by using an underdetermined linear system $A.x = b$. In other words, we want to find a minimal *box* $\boldsymbol{x}'$ such that all the solutions of the system belong to $\boldsymbol{x}'$. A **box** $\boldsymbol{x}$ is a Cartesian product of *intervals* $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_i \times \ldots \times \boldsymbol{x}_n$. An **interval** $\boldsymbol{x}_i = [\underline{x_i}, \overline{x_i}]$ defines the set of values $x_i \in \mathbb{R}$, such that $\underline{x_i} \leq x_i \leq \overline{x_i}$. An interval variant of the Gauss-Seidel algorithm can be used for *contracting* $\boldsymbol{x}$ (i.e., reducing the domains of the variables). However, it does not work well without a proper preconditioning [10]. In underdetermined systems, a conditioner matrix P can be chosen such that $P.A$ is close to a diagonal matrix. Thus, the projection of the equivalent system $P.A.x = P.b$ over x, by using an iterative method like Gauss-Seidel, can be significantly improved. Gauss-Seidel applies the following operation for contracting the domain of each variable x_k:

$$\boldsymbol{x}_k \leftarrow \boldsymbol{x}_k \cap \frac{1}{\hat{a}_{ik}} \left(b_i - \sum_{j,\ j \neq k}^{m} \hat{a}_{ij} \boldsymbol{x}_j \right), \quad \forall i \in \{1..m\} \tag{1}$$

where $\hat{a}_{ij}$ are the coefficients of the matrix $\hat{A} = P.A$.

Techniques used for preconditioning non-square matrices include the Gauss-Jordan elimination method [11] which can provide a preconditioning matrix P by retrieving the row (or column) operations performed in A. This method constructs a pseudo-diagonal matrix in an iterative way, by selecting a subset of m variables as pivots. Usually, the current maximum absolute value of A is selected as pivot in each step. In [12] the authors propose to select the pivot by using five priority rules (e.g., to select columns with at least two values, to select rows with less values, etc).

Another technique is the least squares method. This technique constructs the system $A^T A.x = A^T.b$, which provide us the solution $x = A^T(AA^T)^{-1}b$ known as the Moore-Penrose pseudoinverse. However, the matrix $(AA^T)^{-1}$ may not exist if A is not full rank. In that case, the well-known Singular-Value Decomposition method [13] (also known as SVD) can be used in order to determine the pseudoinverse of A.

In this work, we propose three methods which construct preconditioning matrices in order to deal with underdetermined linear systems. Two of them are based on solving linear programs. The first one constructs a $n \times m$ matrix P by solving n linear programs. The solution of the i-th linear program corresponds to the i-th vector of P and minimizes the size of the projection over the variable x_i (right part in (1)). The second method constructs a $2n \times m$ matrix P by solving $2n$ linear programs. The solution of the first n linear programs corresponds to rows in P that maximize the lower bound of the projection over each of the variables. The other n linear programs minimize the upper bound of the projections. Furthermore, P is able to improve the domain bounds of the variables optimally. Finally, we realized that the problem of finding the optimal preconditioning matrix is equivalent to find the dual feasible solutions of the 2n linear programs solved by OBBT. An equivalent method is proposed in [9] where the authors, instead of generating a preconditioning matrix, generate directly a set of redundant inequalities for improving the bounds of the variables.

We also propose a heuristic for constructing preconditioners based on the Gauss-Jordan pivoting. It takes into account the current variable domains and constructs preconditioners that, when used with a Gauss-Seidel approach, offer a better contraction compared to the ones generated by other state-of-the-art heuristics.

The paper is organized as follows. Section 2 provides basic notions related to interval arithmetic and some remarks about linear systems. We present an example in Section 3. In Section 4 we describe in detail the three contributions of this paper. Section 5 reports the experimental results. Finally, section 6 presents our conclusions and the future work.

2 Background

In this section we introduce some basics concepts related to interval arithmetic and interval linear systems.

2.1 Intervals

An **interval** $\boldsymbol{x}_i = [\underline{x}_i, \overline{x}_i]$ defines the set of reals x_i s.t. $\underline{x}_i \leq x_i \leq \overline{x}_i$, where $\underline{x}_i$ and $\overline{x}_i$ are floating-point numbers. The size or **width** of $\boldsymbol{x}_i$ is defined as $\text{wid}(\boldsymbol{x}_i) = \overline{x}_i - \underline{x}_i$. $\text{mid}(\boldsymbol{x}_i)$ denotes the midpoint of $\boldsymbol{x}_i$, where $\text{mid}(\boldsymbol{x}_i) = \frac{\overline{x}_i + \underline{x}_i}{2}$. A **box** $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ represents the Cartesian product of intervals $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$. The **size** of a box is $\text{wid}(\boldsymbol{x}) = \max_{\boldsymbol{x}_i \in \boldsymbol{x}} \text{wid}(\boldsymbol{x}_i)$. The **perimeter** of a box is $\text{per}(\boldsymbol{x}) = \sum_{i=1}^{n} \text{wid}(\boldsymbol{x}_i)$. A **hull** of a set of vectors in $\mathbb{R}^n$ corresponds to the minimal box containing all of these vectors.

Interval arithmetic defines the extension of unary and binary operators, for instance:

$$\begin{aligned}
\boldsymbol{x}_1 + \boldsymbol{x}_2 &= [\underline{x}_1 + \underline{x}_2, \overline{x}_1 + \overline{x}_2] \\
\boldsymbol{x}_1 - \boldsymbol{x}_2 &= [\underline{x}_1 - \overline{x}_2, \overline{x}_1 - \underline{x}_2] \\
\boldsymbol{x}_1 * \boldsymbol{x}_2 &= [\min(\underline{x}_1\underline{x}_2, \underline{x}_1\overline{x}_2, \overline{x}_1\underline{x}_2, \overline{x}_1\overline{x}_2), \max(\underline{x}_1\underline{x}_2, \underline{x}_1\overline{x}_2, \overline{x}_1\underline{x}_2, \overline{x}_1\overline{x}_2)] \\
\log(\boldsymbol{x}_1) &= [\log(\underline{x}_1), \log(\overline{x_1})]
\end{aligned}$$

A function $f : \mathbb{IR}^n \to \mathbb{IR}$ is **factorable** if it can be computed in a finite number of simple steps, using unary and binary operators. $\boldsymbol{f} : \mathbb{IR}^n \to \mathbb{IR}$ is said to be an **extension** of a real factorable function f to intervals if:

$$\forall \boldsymbol{x} \in \mathbb{IR}^n, \boldsymbol{f}(\boldsymbol{x}) \supseteq \{f(x), x \in \boldsymbol{x}\}.$$

The optimal image $\boldsymbol{f}_{opt}$ is the sharpest interval containing the image of $f(x)$ over $\boldsymbol{x}$. There are several kinds of extensions, in particular, the **natural extension** $\boldsymbol{f}_N$ corresponds to mapping a real n-dimensional function f to intervals by using interval arithmetic.

2.2 Linear systems

A linear system $A.x = b$ is a set of $m > 1$ equations defined by a set of $n > 1$ variables. Without loss of generality, we consider A as a matrix of real coefficients with m rows and n columns. $b \in \mathbb{R}^m$ corresponds to a m-size vector of real values and $x \in \mathbb{R}^n$ corresponds to the vector of variables. Initial domains of variables are represented by a box $\boldsymbol{x}$ which is generally reduced or *contracted* in order to converge to the *hull of solutions*. Linear systems can be classified in three types: square, overdetermined and underdetermined.

Square systems are the most common and studied type, where the number of linear independent equations is equal to the number of variables. Cheap sub-optimal methods can be used for contracting $\boldsymbol{x}$. For instance, the Gauss-Seidel algorithm updates, at each step, the box $\boldsymbol{x}$ by performing the contraction step (1). None of these methods work well without a proper preconditioning. A good, but computationally expensive, preconditioning matrix when A is square, corresponds to $P = A^{-1}$, i.e., the inverse of A. If A is not singular (or numerically close to it), then $P.A$ will correspond to the identity matrix, and the problem can be directly solved: $x = P.b$.

The second type of linear systems belongs to the overdetermined category. In this case the number of equations is greater than the number of variables. As it is not possible to compute the inverse matrix ($m \neq n$), other sub-optimal methods, such as the Gauss-Jordan elimination technique [11], can be used for generating a preconditioner P through the row operations performed by the method. The Gauss-Jordan elimination is a variant of the Gaussian elimination technique. This method is usually used to solve linear systems and to find the inverse of any invertible matrix. Gauss-Jordan transforms a sub-matrix $n \times n$ of A into a pseudo-identity matrix. The algorithm selects an element a_{ij} (known as the pivot) of A, and by performing row operations, it leaves a_{ij} equal to 1 and the other elements of the column j equal to 0. Note that any element in the column j can not be selected as pivot in the future.

A more recent technique, known as the subsquares approach, is proposed in [14]. This method extracts, sequentially, square systems $n \times n$ from the original overdetermined system, performing a contraction of $\boldsymbol{x}$ by using each one of these systems. As there are $\binom{m}{n}$ possible combinations of square systems, the authors propose an heuristic to only select a fraction of them. Even if it does not compute the hull, it gives good results compared to other classical approaches.

Finally, the underdetermined linear systems, which are studied in this paper, have more variables than equations (i.e., $m < n$). In this case, if we apply the Gauss-Jordan elimination, a matrix $P.A = [I\ R]$ is generated, where I corresponds to an identity matrix of size $m \times m$ and R represents a residual $m \times (m - n)$ matrix. As $P.A$ is not diagonal, the contraction is not optimal. Small values for the residual matrix are preferred in order to obtain better contractions. Thus, the order in which the pivots are selected is crucial. A reasonable and widely used strategy for the pivoting process corresponds to select, in each iteration of the Gauss-Jordan elimination, the current maximum absolute value of the matrix A [15].

3 Example: contracting a linear system

In order to explain preconditioning and to describe the new proposed methods, we will consider an underdetermined linear system example. First note that a linear system of constraints $a \leq \sum x'_i + c \leq b$ can be represented by a linear system $A'.x' = y$, where $y \in [a - c, b - c]^m$ is an auxiliary vector of variables. $A'.x' = y$ is equivalent to the linear system $A.x = 0$, where $x = \begin{pmatrix} x' \\ y \end{pmatrix}$ and $A = \begin{pmatrix} A' & -I \end{pmatrix}$, where I is an identity matrix of size $m \times m$. Thus in the following, and without loss of generality, we deal with the problem $A.x = 0$, with $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$.

Example 1 *Consider the following underdetermined linear system (coefficients and domains were randomly generated):*

$$\begin{pmatrix} -7.31 & 6.95 & 5.28 & -4.90 & -0.09 \\ -1.01 & 3.03 & 5.77 & -8.12 & -9.43 \\ 6.72 & -1.34 & 5.25 & -9.96 & -1.09 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

where the domains are $\boldsymbol{x}_1 = [-1.565, 2.880]$, $\boldsymbol{x}_2 = [0.478, 4.463]$, $\boldsymbol{x}_3 = [-1.038, 6.032]$, $\boldsymbol{x}_4 = [0.048, 3.615]$ *and* $\boldsymbol{x}_5 = [-1.076, 2.647]$.
If we apply the Gauss-Seidel steps (1) directly, no contraction is afforded in any of the five variables. On the other hand, if we apply the Gauss-Jordan technique by pivoting the maximum absolute value of A in each iteration, we obtain the following preconditioning matrix:

$$\begin{pmatrix} -0.091 & -0.004 & 0.048 \\ 0.069 & -0.113 & 0.058 \\ -0.069 & 0.009 & -0.073 \end{pmatrix}$$

Consequently, for P.A we obtain:

$$\begin{pmatrix} 1 & -0.714 & -0.253 & 0 & 0 \\ 0 & 0.059 & 0.017 & 0 & 1 \\ 0 & -0.353 & -0.699 & 1 & 0 \end{pmatrix}$$

By applying contraction steps (1) on the new system P.A.x = P.b, we obtain a contraction in some domains: $\boldsymbol{x}_1 \leftarrow [0.078, 2.880]$, $\boldsymbol{x}_2 \leftarrow [0.478, 4.400]$, $\boldsymbol{x}_3 \leftarrow [-1.038, 4.922]$ *and* $\boldsymbol{x}_5 \leftarrow [-0.352, -0.009]$

4 Toward an optimal contraction of underdetermined linear systems

In this section we describe in detail three new approaches for dealing with underdetermined linear systems $A.x = b$. All of them construct a preconditioning matrix P which attempts to improve the projection performed by the Gauss-Seidel contraction step (1).

The first proposal corresponds to an improvement of the Gauss-pivot selection heuristic by taking into account information of the box $\boldsymbol{x}$. The second and third proposals aim to construct the preconditioning matrix by solving linear programs.

4.1 Improving the Gauss-pivoting heuristic

At the end of Section 2.2, we explained that the Gauss-Jordan technique can be used for generating a preconditioning matrix P for the system $A.x = b$. From (1) we can see that, in order to increase the likelihood of contracting a variable $\boldsymbol{x}_k$, the interval evaluation of $\frac{1}{\hat{a}_{ik}} \left(b_i - \sum_{j,\, j \neq k}^{m} \hat{a}_{ij} \boldsymbol{x}_j \right)$, where $\hat{a}_{ij}$ are the coefficients of the matrix $\hat{A} = P.A$, should be as tight as possible. The width of this interval is:

$$\frac{1}{|\hat{a}_{ik}|} \left(\sum_{j, j \neq k} \hat{a}_{ij} \cdot \mathrm{wid}(\boldsymbol{x}_j) \right) \tag{2}$$

When applying Gauss-Jordan elimination, an indirect way of reducing this size is by pivoting the variable which maximizes the value of $|\hat{a}_{ik}|$ in each iteration. When doing this, in a way, we are selecting the row i for contracting $\boldsymbol{x}_k$: $|\hat{a}_{ik}|$ is the largest value in the row i, thus, according to (2), when $|\hat{a}_{ik}|$ is large more likely we will obtain a tight projection over $\boldsymbol{x}_k$. In addition, the Gauss-Jordan method *removes* the coefficient related to this variable from the other rows benefiting the contraction over the other variables. Note the $P.A$ matrix in Example 1. In each row, the pivoted value is 1 and corresponds to the largest absolute value. Other values in the related columns are canceled benefiting projection over the other variables.

Following the same idea, we think that it is also relevant to take into account the width of the next pivoting variable. For instance, if the width of $\boldsymbol{x}_k$ in (1) is too small, the likelihood of contracting this variable will be small too. On the contrary, if $\boldsymbol{x}_k$ is large, it is more likely to contract its domain. If we normalize

variable domains to intervals $\boldsymbol{e}_j = [-1, 1]$, i.e., $\boldsymbol{e}_j := \frac{\boldsymbol{x}_j}{\text{wid}(\boldsymbol{x}_j)} - \text{mid}(\boldsymbol{x}_j)$, then the width of the projection over $\boldsymbol{e}_k$ is equal to:

$$\frac{1}{|\hat{a}_{ik}| \cdot \text{wid}(\boldsymbol{x}_k)} \left(\sum_{j=0, j \neq k}^{m} |\hat{a}_{ij}| \cdot \text{wid}(\boldsymbol{x}_j) \right) \tag{3}$$

Thus, we propose as pivoting heuristic, to select the element $|\hat{a}_{ik}|$ such that $|\hat{a}_{ik}| \cdot \text{wid}(\boldsymbol{x}_k)$ is maximized.

Example 2 *If we use the Gauss-Jordan elimination method, using as pivoting rule the value that maximizes the product $|a_{ij}| \cdot wid(\boldsymbol{x}_j)$, in the example we obtain the following preconditioner:*

$$\begin{pmatrix} 0.067 & -0.113 & 0.056 \\ 0.098 & -0.013 & 0.105 \\ -0.066 & -0.008 & -0.075 \end{pmatrix}$$

Consequently, for P.A we obtain:

$$\begin{pmatrix} 0 & 0.050 & 0 & 0.025 & 1 \\ 0 & 0.505 & 1 & -1.428 & 0 \\ 1 & -0.586 & 0 & -0.361 & 0 \end{pmatrix}$$

Once we perform algorithm 1, we obtain an additional contraction compared to just pivoting the cell with the largest absolute value. We obtain additional contraction on: $\boldsymbol{x}_1 \leftarrow [0.297, 2.880]$ ~~$[0.078, =]$~~ and $x_5 \leftarrow [-0.319, -0.025]$ ~~$[-0.352, -0.009]$~~.

4.2 Linear-based preconditioning

Despite offering good projections over $\boldsymbol{x}$, the Gauss-Jordan-based preconditioning methods rarely lead to optimal contractions. In this section, we describe two methods that directly focus on optimizing the projection over variables.

4.2.1 Minimizing the size of the interval projection

First, we attempt to construct a preconditioning *vector* $p = (p_1, p_2, ..., p_m)$ such that the projection of the system $p.A.x = 0$ over the interval $\boldsymbol{x}_k$ has a minimum size. The values of the vector $\hat{a} = p.A$ are computed:

$$\hat{a}_j = \sum_{i=1}^{m} p_i \cdot a_{ij}, \quad \forall j = 1, \ldots, n,$$

where a_{ij} are the coefficients of matrix A. Thus, taking into account the interval projection size (2), a preconditioning vector p for minimizing this size (related to a variable x_k), can be generated by solving the following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1,\ j \neq k}^{n} |\hat{a}_j| \cdot wid(\boldsymbol{x}_j) \\ & \hat{a}_k = 1 \\ \text{s.t.} \quad & \hat{a}_j = \sum_{i=1}^{m} p_i \cdot a_{ij}, \quad \forall j = 1, \ldots, n \end{aligned} \tag{4}$$

Note that by adding the constraint $\hat{a}_k = 1$ we can remove the quotient $|\hat{a}_{ik}|$ of formula (2) from the objective function. For constructing the preconditioning matrix P, we have to solve the linear program for each variable x_k *that we want to contract* and to include the precondition vectors as rows in P.

In order to deal with the absolute value inside the objective function, we replace $|\hat{a}_j|$ by auxiliary variables u_j. Then, we add the constraints $u_j \geq \hat{a}_j$ and $u_j \geq -\hat{a}_j$ and we solve the equivalent linear program by using the simplex algorithm.

Example 3 *For contracting $\boldsymbol{x}_1$ of Example 1, we would generate the following linear program:*

$$
\begin{array}{lll}
\textit{minimize} & 4.45u_1 + 3.98u_2 + 7.07u_3 + 3.57u_4 + 3.72u_5 & \\
& \hat{a}_1 = 1 & \\
& u_r \geq s_r & ; r = 1..5 \\
& u_r \geq -s_r & ; r = 1..5 \\
\textit{s.t.} & \hat{a}_1 = 7.31p_1 + 1.01p_2 - 6.72p_3 & \\
& \hat{a}_2 = -6.95p_1 - 3.03p_2 + 1.34p_3 & \\
& \hat{a}_3 = -5.28p_1 - 5.77p_2 - 5.25p_3 & \\
& \hat{a}_4 = 4.90p_1 + 8.12p_2 + 9.96p_3 & \\
& \hat{a}_5 = 0.08p_1 + 9.43p_2 + 1.09p_3, &
\end{array}
$$

with optimal solution $p^* = (-0.066, -0.008, 0.075)$. By solving the linear problems related to the other variables we would obtain the following preconditioner P:

$$
\begin{pmatrix}
-0.066 & -0.008 & 0.075 \\
0.127 & 0.006 & -0.068 \\
0.098 & -0.013 & 0.105 \\
-0.023 & 0.011 & -0.098 \\
0.067 & -0.113 & 0.056
\end{pmatrix}
$$

Finally, $P.A$ is:

$$
\begin{pmatrix}
1 & -0.586 & 0 & -0.361 & 0 \\
-1.400 & 1 & 0.354 & 0 & 0 \\
0 & 0.505 & 1 & -1.428 & 0 \\
-0.495 & 0 & -0.574 & 1 & 0 \\
0 & 0.050 & 0 & 0.025 & 1
\end{pmatrix}
$$

Compared to the pivoting heuristic in Example 2, the preconditioner obtained by solving linear programs offers an additional contraction on: $\boldsymbol{x}_2 \leftarrow [0.478, 4.400]$ ~~$[=, 4.463]$~~ and $\boldsymbol{x}_5 \leftarrow [-0.316, -0.025]$ ~~$[-0.319, =]$~~.

4.2.2 Minimizing/maximizing the upper/lower bound of the interval projection

Extending the idea of the previous section, now we attempt to construct preconditioning vectors $p = (p_1, p_2, ..., p_m)$ such that the projection minimize (resp. maximize) the upper (resp. lower) bound of the projection of the system $p.A.x = 0$ over the interval x_k. Considering that $\hat{a} = p.A$, the upper bound of the projection of $\hat{a}.x = 0$ over the interval $\boldsymbol{x}_k$ is:

$$
\overline{-\frac{1}{\hat{a}_k}\left(\sum_{j,j\neq k}^{n} \hat{a}_j \boldsymbol{x}_j\right)} \tag{5}
$$

Minimizing (5) is equivalent to maximize $\sum_{j,j\neq k}^{n} \underline{\hat{a}_j \boldsymbol{x}_j}$ with $\hat{a}_k = 1$. We can replace $\underline{\hat{a}_j \boldsymbol{x}_j}$ by auxiliary variables w_j and two inequalities: $w_j \leq \hat{a}_{ij}\underline{x_j}$ and $w_j \leq \hat{a}_{ij}\overline{x_j}$. Finally, we obtain the following linear program equivalent to minimize (5):

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1,\ j\neq k}^{n} w_j \\ \text{s.t.} \quad & \hat{a}_k = 1 \\ & \hat{a}_j = \sum_{i=1}^{m} p_i \cdot a_{ij} \quad \forall j = 1,\ldots,n \\ & w_j \leq \hat{a}_j \underline{x_j} \quad \forall j = 1,\ldots,n;\ \ j \neq k \\ & w_j \leq \hat{a}_j \overline{x_j} \quad \forall j = 1,\ldots,n;\ \ j \neq k \end{aligned} \tag{6}$$

An opposite and analogous reasoning can be performed in order to obtain a linear problem for maximizing the lower bound of the projection of $p.A.x = 0$ over $\boldsymbol{x}_k$.

Then, solving the linear programs results in obtaining preconditioning vectors p. By means of Gauss-Seidel projections, each of these vectors is capable of improving one of the bounds of an interval $\boldsymbol{x}_k$. The obtained vectors p can be included in a preconditioning matrix P (duplicated vectors can be discarded).

Example 4 *By solving the two linear programs for each variable in Example 1, we obtain the following preconditioning matrix (recall that each row corresponds to a solution vector p of a linear program):*

$$\begin{pmatrix} -0.066 & -0.008 & 0.075 \\ 0.127 & 0.006 & -0.068 \\ 0.098 & -0.013 & 0.105 \\ -0.023 & 0.011 & -0.098 \\ 0.067 & -0.113 & 0.056 \\ -0.069 & 0.009 & -0.073 \\ 0.061 & -0.113 & 0.062 \end{pmatrix}$$

The first five rows were obtained by minimizing upper bounds of projections, while the last two were obtained by maximizing lower bounds of projections. The missing rows correspond to duplicated ones. Finally, we obtain the following matrix P.A:

$$\begin{pmatrix} 1 & -0.586 & 0 & -0.361 & 0 \\ -1.400 & 1 & 0.354 & 0 & 0 \\ 0 & 0.505 & 1 & -1.428 & 0 \\ -0.495 & 0 & -0.574 & 1 & 0 \\ 0 & 0.050 & 0 & 0.025 & 1 \\ 0 & -0.353 & -0.699 & 1 & 0 \\ 0.083 & 0 & -0.003 & 0 & 1 \end{pmatrix}$$

Compared to the preconditioner obtained in Section 4.2.1, the preconditioner in this section offers additional contraction on $\boldsymbol{x}_5 \leftarrow [-0.245, -0.025]$ ~~$[-0.316, =]$~~.

Proposition 1 *Let p be an optimal solution of the linear program (6). Then, by using the system p.A.x and Gauss-Seidel, we can improve optimally the upper bound of the interval* $\boldsymbol{x}_k$.

Proof The optimal upper bound of a interval domain $\boldsymbol{x}_k$ is equivalent to the maximum value of x_k subject to the constraint system $A.x$, i.e.,

$$\begin{aligned} \text{maximize} \quad & x_k \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} \cdot x_j = 0, \quad \forall i = 1, \ldots, m \\ & \underline{x_j} \leq x_j \leq \underline{x_j}, \quad \forall i = 1, \ldots, m \end{aligned} \tag{7}$$

We consider first the dual problem of (7). Let $\pi \in \mathbb{R}^m$ be the vector associated to the constraints, $\ell \in \mathbb{R}^n$ be the vector associated to the bound constraints $\underline{x_j} \leq x_j$, and $u \in \mathbb{R}^n$ be the vector associated to the bound constraints $x_j \leq \overline{x_j}$. Thus, the dual problem of (7) can be stated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} 0 \cdot \pi + \sum_{j=1,\ j\neq k}^{n} \underline{x_j} \cdot \ell_j - \sum_{j=1,\ j\neq k}^{n} \overline{x_j} \cdot u_j \\ \text{s.t.} \quad & \sum_{i=1}^{m} a_{ij} \cdot \pi + \ell_j - u_j = 0, \quad \forall j = 1, \ldots, n; \ \ j \neq k \\ & \sum_{i=1}^{m} a_{ik} \cdot \pi = 1 \\ & \ell, u \leq 0, \pi \text{ free} \end{aligned}$$

By defining $p_i = \pi_i$ and $\hat{a}_j = \sum_{i=1}^{m} p_i \cdot a_{ij}$, and also developing the previous linear program, we obtain:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1,\ j\neq k}^{n} (\overline{x_j} \cdot u_j - \underline{x_j} \cdot \ell_j) \\ \text{s.t.} \quad & \hat{a}_j + \ell_j - u_j = 0, \quad \forall j = 1, \ldots, n; \ \ j \neq k \\ & \sum_{i=1}^{m} p_i \cdot a_{ij} = \hat{a}_j, \quad \forall j = 1, \ldots, n; \ \ j \neq k \\ & \hat{a}_k = 1 \\ & \ell, u \leq 0, \pi \text{ free} \end{aligned}$$

Let $w_j = \overline{x_j} \cdot u_j - \underline{x_j} \cdot \ell_j$, $\forall j \neq k$. Note that if the first constraint is multiplied by $-\overline{x_j}$ we obtain:

$$\begin{aligned} -\overline{x_j} \cdot \hat{a}_j + \overline{x_j} \cdot u_j &= \overline{x_j} \cdot \ell_j \quad / \text{ adding } -\underline{x_j} \cdot \ell_j \\ -\overline{x_j} \cdot \hat{a}_j + w_j &= \overline{x_j} \cdot \ell_j - \underline{x_j} \cdot \ell_j \\ w_j &= \overline{x_j} \cdot \hat{a}_j + \ell_j(\overline{x_j} - \underline{x_j}), \end{aligned}$$

as $\ell_j \leq 0$, we can deduce that $w_j \leq \overline{x_j} \cdot \hat{a}_j$. Using the same procedure, but multiplying but $\underline{x_j}$ instead, it can be deduced that $w_j \leq \underline{x_j} \cdot \hat{a}_j$. Finally, we reach the same linear program stated in (6).

As finding *the best* preconditioning vector p for projecting over the upper bound of $\boldsymbol{x}_k$ is equivalent to the dual linear problem of finding an optimal upper bound of $\boldsymbol{x}_k$, then, according to the duality theorem, the value of the optimal solutions is the same. In other words, by using the preconditioned system $p.A.x$ and the Gauss-Seidel procedure, we can improve optimally the upper bound of $\boldsymbol{x}_k$. □

Proposition 1 can be extended to lower bounds in a straightforward way. It is important to highlight that an equivalent proposition was derived in [9] by directly using duality theory of linear programming.

5 Experiments

In order to validate our approach, several sets of benchmark instances were generated by a random linear system generator[1]. The generator constructs rectangular linear systems $A.x = b$ with n variables and m constraints ($n > m$). Each constraint i has the following structure:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \tag{8}$$

where a_{ij} corresponds to a real value between -10 and 10. Without loss of generality, we have set b to the null vector (i.e., x equal to a null vector is always a solution of the problem). Additionally, for all the experiments performed, the number of variables have been fixed to $n = 20$. The number of constraints varies from 12 to 19. For each value of m, 20 systems have been generated. Additionally, each variable domain has been set initially to a random interval, with a minimum lower bound of -50 and a maximum upper bound of 50. Each of these interval are forced to contain the 0 in order to avoid empty solutions/manifolds.

All the strategies explained in the previous sections have been incorporated into Ibex [16], a C++ state-of-the-art library for constraint processing over real numbers.

5.1 Contracting power

Figure 1 reports a comparison between the different strategies. The plot on the left side shows, on problems with different number of constraints, the average relative width, w.r.t. the input domain width, reached by the most contracted intervals after contraction (e.g., 1.0 means no contraction). The right plot shows the average relative perimeter, w.r.t. the input box perimeter, reached by the boxes after contraction.

All the strategies perform the Gauss-Seidel procedure for contracting $\boldsymbol{x}$ on the system $P.A.x = 0$. The strategy GAUSS MAX constructs the preconditioning matrix P by using the Gauss elimination method with the maximum heuristic, while the strategy GAUSS MAX-DIAM constructs P by using the heuristic that takes into account the box $\boldsymbol{x}$. The strategy LP MIN-SIZE constructs P by solving the linear programs (4) that minimize the size of the projection intervals. On the other hand, the strategy LP OPT constructs P by solving the linear programs (6) that maximize and minimize the bounds of the variable domains.

[1] `https://github.com/vareyesr/linear-generator`

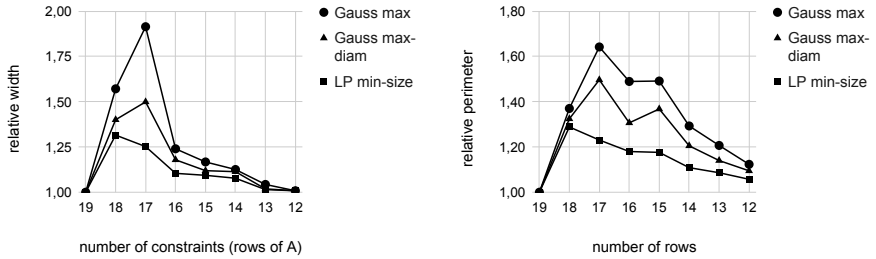


Fig. 1: Average relative contraction in function of the number of constraints in the linear systems. (left) Relative width of the domain of the most contracted variable; (right) relative perimeter after contraction.

From the figure we can see that the linear-based preconditioners are the ones that contract the most. The largest difference on the minimum diameter are obtained when the number of constraints is 14 and 15. On the other hand, the largest differences in the perimeter are obtained when the number of constraints is between 14 and 16.

Note that when the number of rows is 19, all the strategies obtain the same contraction. The reason is because in this case, all the preconditioners behaves as A^{-1} as the number of rows in A is almost equivalent to the number of columns. The same case is obtained when the number of constraints is low. In this case it seems that the system approaches the *global hull-consistency*. Global hull-consistency occurs when each domain bound is part of a solution.

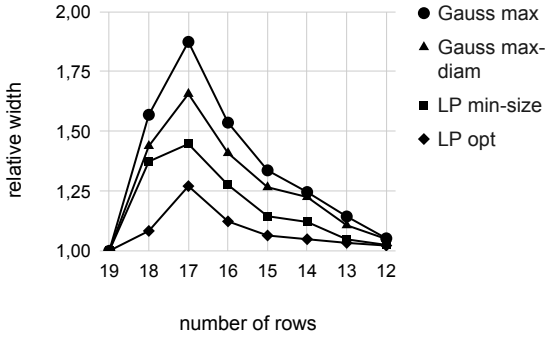
Note that the contraction of LP OPT is optimal (i.e., it is equivalent to perform 2n calls to simplex). On the other hand, the contraction performed by LP MIN-SIZE is sub-optimal, this is due to this strategy uses only one preconditioning vector p for improving both bounds of each interval (unlike LP OPT that uses one vector p for improving each bound).

Additionally, we can see that by including the variable domains in the Gauss pivoting heuristics (GAUSS MAX-DIAM), we reach a significant better contraction related to its counterpart.

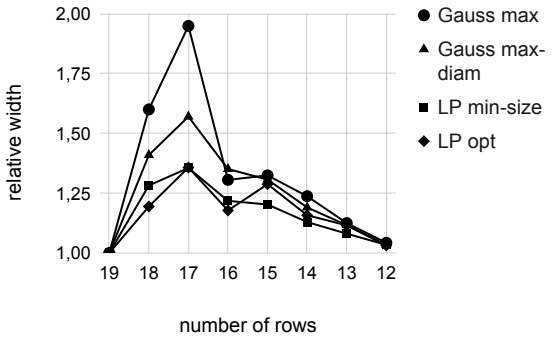
5.2 Sustainability

In a second series of experiments we evaluate the *sustainability* of the approaches. That is, we want to know how long in the search we could use the same preconditioner P without losing too much effectiveness in contraction.

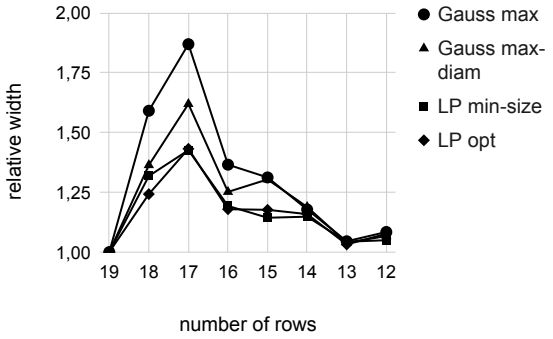
Thus, the experiment consists in first generating a preconditioning matrix P using the same set of benchmarks and *initial box* $\boldsymbol{x}$ as the previous experiment. Then, we arbitrarily and randomly reduce this box to a fraction of its original width (50%, 10% and 1%). Plots in Figure 2 reports the contraction performed by the strategies (on the most contracted variable) by using the *reduced box*. In this way, we simulate, in a certain way, what happens in an iteration of a B&B solver after some subdivisions and domain reductions of the initial box $\boldsymbol{x}$.



(a)



(b)



(c)

Fig. 2: Relative width of the domain of the most contracted variable on linear systems with different number of constraints. Each strategy uses the initial box $\boldsymbol{x}$ for generating the preconditioning matrix P. Then, the contraction is performed on a randomly generated box $\boldsymbol{x}' \in \boldsymbol{x}$ with (a) 50% (b) 10% and (c) 1% of the width of $\boldsymbol{x}_0$.

We also included the strategy OBBT in the plots. OBBT gives us a reference for the optimal contraction that can be reached (by performing 2n calls to the simplex algorithm). In the plots we can see that, as the widths decrease, the strategies move away from the optimal contraction given by OBBT. Note also that the linear-based strategies remain below the Gauss-based ones. Just as the previous experiment, the Gauss-based strategy that takes into account the domains of the variables is better than its counterpart, even when the size of the box is small.

When the width of the reduced box is 50% of the original width, we can see that the contraction performed by LP OPT is still the best among all the strategies. However, when the width is small (10% or 1% of the original width), LP MIN-SIZE is more effective than LP OPT. We think that as the preconditioning vectors p generated by the strategy LP MIN-SIZE focus on improving both bounds of a variable *at the same time*, they probably are more adaptable to changes of the interval domain bounds.

6 Conclusions

In this work we propose three methods for generating preconditioning matrices for underdetermined linear systems $A.x = b$. These preconditioners can be used for improving the contraction of iterative methods such as the Gauss-Seidel algorithm. The first one generates the preconditioner by using a Gauss-Jordan elimination method that takes into account the width of the intervals for selecting the next coefficient in A to pivot. In this way, we are selecting the row i maximizing $a_{ik} \cdot \text{wid}(\boldsymbol{x}_k)$ for contracting $\boldsymbol{x}_k$ instead of simply selecting the row maximizing a_{ik} as a previous approach. Additionally, we have presented two preconditioners generated by solving linear programs. They are focused on minimizing the interval size and optimizing the interval bounds respectively.

Experiments show promising results. On the one hand, by using the preconditioner based on Gauss-Jordan elimination we obtain a better contraction than using its counterpart which selects the maximum coefficient of A for pivoting. On the other hand, the preconditioners generated by solving linear programs outperform the ones based on Gauss-Jordan elimination.

We also show that, by using the preconditioner focused on optimizing the interval bounds of a box $\boldsymbol{x}$, we reach an optimal contraction of this box. In addition, when a smaller box $\boldsymbol{x}' \subset \boldsymbol{x}$ is contracted, although the preconditioner does not offer an optimal contraction, it is still better than the Gauss-Jordan based strategies.

As a future work we will integrate the preconditioning methods into a global optimization solver. To avoid the costly n/2n calls to the simplex algorithms for generating the preconditioners based on linear programming, we plan to design a mechanism for updating P only when it is needed, e.g., when some *relevant* coefficients in A or some *relevant* bounds of variable domains suffer significant changes. Finally, we plan to optimize the time efficiency of the Gauss-Seidel method when only a few values are modified.

References

1. M. Locatelli and F. Schoen, *Global optimization: theory, algorithms, and applications.* SIAM, 2013.

2. I. Araya, G. Trombettoni, and B. Neveu, "A contractor based on convex interval taylor," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2012, pp. 1–16.
3. C. S. Adjiman, S. Dallwig, C. A. Floudas, and A. Neumaier, "A global optimization method, αbb, for general twice-differentiable constrained nlps—i. theoretical advances," *Computers & Chemical Engineering*, vol. 22, no. 9, pp. 1137–1158, 1998.
4. R. Misener and C. A. Floudas, "Antigone: algorithms for continuous/integer global optimization of nonlinear equations," *Journal of Global Optimization*, vol. 59, no. 2-3, pp. 503–526, 2014.
5. P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter, "Branching and bounds tightening techniques for non-convex minlp," *Optimization Methods & Software*, vol. 24, no. 4-5, pp. 597–634, 2009.
6. I. Nowak and S. Vigerske, "Lago: a (heuristic) branch and cut algorithm for nonconvex minlps," *Central European Journal of Operations Research*, vol. 16, no. 2, pp. 127–138, 2008.
7. T. Achterberg, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.
8. G. Trombettoni, A. Ignacio, B. Neveu, and G. Chabert, "Inner regions and interval linearizations for global optimization," in *AAAI 2011*, 2011.
9. A. M. Gleixner, T. Berthold, B. Müller, and S. Weltge, "Three enhancements for optimization-based bound tightening," *Journal of Global Optimization*, vol. 67, no. 4, pp. 731–757, 2017.
10. H. Niki, T. Kohno, and M. Morimoto, "The preconditioned gauss–seidel method faster than the sor method," *Journal of Computational and Applied Mathematics*, vol. 219, no. 1, pp. 59–71, 2008.
11. E. Hansen and G. W. Walster, "Solving overdetermined systems of interval linear equations," *Reliable computing*, vol. 12, no. 3, pp. 239–243, 2006.
12. M. Ceberio and L. Granvilliers, "Solving nonlinear equations by abstraction, gaussian elimination, and interval methods," in *International Workshop on Frontiers of Combining Systems*. Springer, 2002, pp. 117–131.
13. G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," in *Linear Algebra*. Springer, 1971, pp. 134–151.
14. J. Horáček and M. Hladík, "Subsquares approach–a simple scheme for solving overdetermined interval linear systems," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 613–622.
15. F. Domes and A. Neumaier, "Rigorous filtering using linear relaxations," *Journal of Global Optimization*, vol. 53, no. 3, pp. 441–473, 2012.
16. G. Chabert and L. Jaulin, "Contractor Programming," *Artificial Intelligence*, vol. 173, pp. 1079–1100, 2009.

A revised monotonicity-based method for computing sharp image enclosures of functions.

Ignacio Araya[1,a)], Victor Reyes[2,b)] and Alen Figueroa[1,c)]

[1]*Pontificia Universidad Católica de Valparaíso, Escuela de Ingeniería Informática, Chile.*
[2]*Universidad Diego Portales, Escuela de Informática y Telecomunicaciones Chile.*

[a)]Corresponding author: ignacio.araya@pucv.cl
[b)]victor.reyes@udp.cl
[c)]alen.figueroa@mail.pucv.cl

Abstract. The computation of sharp interval image enclosures of functions over bounded variable domains is in the heart of interval-based branch and bound optimization (and constraint satisfaction) solvers. Generally, computing sharper enclosures allows solvers to filter non feasible and non-optimal regions in a more effective way, improving thus the performance of the whole algorithm.

Interval arithmetics extends to intervals arithemtics operators, such as $+, -, *$, sin, cos, etc. Then, we can directly use these operators for computing image enclosures of real functions over bounded domains (i.e., natural interval evaluations). On the other hand, it is well-known that when a function f is monotonic w.r.t. some variable(s) in a given domain, we can compute sharper images of f on this domain than by using natural interval evaluations.

In this work, we propose a more general monotonicity-based method that may be applied even if the function is non monotonic w.r.t. its variables. The method combines basic interval-based filtering techniques with a straightforward analysis of function derivatives. First, by performing filtering with partial derivatives, we detect sub-intervals in the domain where the function *certainly* increase or decrease. Then, depending on the case, we can know in which sub-domains in the interval should be the value maximizing (resp. minimizing) the function. Finally, we use the natural interval evaluation on the sub-domains maximizing f (resp. minimizing f) for computing the upper bound (resp. lower bound) of the enclosure. We show that this method is equivalent to computing an enclosure by using the traditional monotonicity-based method when f is monotonic, however, it may be much more effective when f is not.

INTRODUCTION

The computation of sharp interval image enclosures is in the heart of interval based methods [1]. When solving optimization problems, interval branch-and-bound methods test on every node if constraints $f(x) \leq 0$ are satisfied, i.e., if the enclosure of f contains values in $(-\infty, 0]$, otherwise the node may be discarded. Also, constraint propagation algorithms, used at each node of the search tree to filter variable domains, can be improved when they use better methods for computing enclosures. For instance, Mohc [2], a constraint propagation algorithm, is based on montonicity-based computations of the enclosure; the filtering method proposed in [3] uses a a test of existence based on monotonicity for filtering the initial bounds of the variable domains. In this work, we propose a new method based on monotonicity for computing sharp image enclosures of functions.

First recall basic material about interval arithmetics [1, 4] to introduce the interval extensions useful in our work. An **interval** $\boldsymbol{x}_i = [\underline{x}_i, \overline{x}_i]$ defines the set of reals x_i s.t. $\underline{x}_i \leq x_i \leq \overline{x}_i$, where $\underline{x}_i$ and $\overline{x}_i$ are floating-point numbers. The **hull** of a set of intervals is defined as $\text{hull}(\boldsymbol{x}_1, ..., \boldsymbol{x}_n) = \left[\min_{i\in\{1..n\}}(\underline{x_i}), \max_{i\in\{1..n\}}(\overline{x_i})\right]$. A **box** $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ represents the Cartesian product of intervals $\boldsymbol{x}_1 \times \ldots \times \boldsymbol{x}_n$. Interval arithmetic defines the extension of unary and binary operators, for instance, $\boldsymbol{x}_1 + \boldsymbol{x}_2 = [\underline{x}_1 + \underline{x}_2, \overline{x}_1 + \overline{x}_2]$, $\log(\boldsymbol{x}_1) = [\log(\underline{x_1}), \log(\overline{x_1})]$.

A function $f : \mathbb{IR}^n \to \mathbb{IR}$ is **factorable** if it can be computed in a finite number of simple steps, using unary and binary operators. $\boldsymbol{f} : \mathbb{IR}^n \to \mathbb{IR}$ is said to be an **extension** of a real factorable function f to intervals if it computes an enclosure of the image range, i.e., $\forall \boldsymbol{x} \in \mathbb{IR}^n, \boldsymbol{f}(\boldsymbol{x}) \supseteq \{f(x), x \in \boldsymbol{x}\}$. An **optimal enclosure** is the sharpest interval containing the image of $f(x)$ over $\boldsymbol{x}$.

There are several kinds of extensions. The **natural extension** $\boldsymbol{f}_n$ corresponds to mapping a real n-dimensional function f to intervals by using interval arithmetic. For instance, if we compute the image of the real function:

$$f(x_1, x_2) = x_1^2 - 5x_1 - x_2^3 + 2x_2^2 + 14x_2 \tag{1}$$

by applying the natural extension over the box $\boldsymbol{x} = [0, 2] \times [-2, 1]$, we obtain: $[0, 2]^2 - 5 \cdot [0, 2] - [-2, 1]^3 + 2 \cdot [-2, 1]^2 + 14 \cdot [-2, 1] = [-39, 34]$. In general, the natural extension does not compute an optimal enclosure due to the *dependence problem* [5], i.e., interval operators consider each occurrence of a variable as a *different variable*. We can reduce the number of occurrences factorizing some variables, for example, expression (1) can be re-written as $f(x_1, x_2) = x_1(x_1 - 5) + x_2(14 + 2x_2 - x_2^2)$. By applying the natural extension over the factorized expression we obtain a sharper interval: $\boldsymbol{f}(\boldsymbol{x}) = [-42, 16]$.

The **monotonicity-based extension** $\boldsymbol{f}_m$ first computes lower and upper bounds of the partial derivatives of f to identify the *monotonic variables*[1]. Each variable x_i is fixed to a value x_i^+ that maximizes (resp. a value x_i^- that minimizes) the evaluation of f (as x_i are monotonic variables, x_i^+ and x_i^- are bounds of the interval domain). Then, an upper bound (resp. lower bound) of the image of f is computed by using the natural extension: $\boldsymbol{f}_m(\boldsymbol{x}) = \left[\boldsymbol{f}_n(x_1^-, ...x_n^-, \boldsymbol{x}^0), \boldsymbol{f}_n(x_1^+, ..., x_n^+, \boldsymbol{x}^0)\right]$, where $\boldsymbol{x}^0$ is an interval vector of non-detected monotonic variables. By considering real values instead of intervals for some variables, we can reach sharper enclosures by using the natural extension [5]. Consider, for example, the expression (1). We note that f is decreasing w.r.t. x_1 ($\frac{\partial f}{\partial x_1} \in [-5, -1]$). By applying the monotonicity-based extension (on the factorized expression) we obtain a sharper image than using the natural extension: $\boldsymbol{f}_m(\boldsymbol{x}) = [-38, 16] \subset [-42, 16]$.

Filtering algorithms (or **contractors**), are methods that removes values from the bound of a box that do not satisfy one or several constraints. If we require to filter the domain of a variable with only one constraint, a straightforward method consists on isolating the variable that we want to filter and compute the image of the generated projection function. For instance, consider the equation $x^3 + 3x = 14$, with domain $\boldsymbol{x} = [0, 3]$. If we want to filter the domain of x, we can isolate each occurrence of x and compute enclosures for the projection functions: $\boldsymbol{x} \leftarrow \boldsymbol{x} \cap (-3\boldsymbol{x} + 14)^{1/3}$ and $\boldsymbol{x} \leftarrow \boldsymbol{x} \cap \frac{-\boldsymbol{x}^3 + 14}{3}$. After performing the projections once, we reach $\boldsymbol{x} = [1.70, 2.42]$. After performing the projections several times, $\boldsymbol{x}$ converges to the solution of the equation $\boldsymbol{x} = [2, 2]$. This is not always the case and the projection generally converge to sub-optimal domains. The algorithm HC4-Revise [6][2] is a well-known contractor that filters the domains of variables without explicitly generating the projection functions but directly working on the expression tree related to the constraint.

PROPOSAL

In this work, we propose $\boldsymbol{f}_G$, a generalized version of the monotonicity-based extension. The method is based on simple monotonicity analysis for computing sharp function enclosures when variables are not detected monotonic. It works on three steps:

1. For each domain $\boldsymbol{x}_i$ we identify sub-domains where f is monotonically increasing or decreasing w.r.t. x_i;
2. by analyzing the sub-domains we generate a box $\boldsymbol{x}^+ = \boldsymbol{x}_1^+ \times ... \times \boldsymbol{x}_n^+ \subseteq \boldsymbol{x}$ containing the vector maximizing f. Respectively, we generate the box $\boldsymbol{x}^- \subseteq \boldsymbol{x}$ containing the vector minimizing f;
3. we compute the enclosure by using the natural extension on each box, i.e., $\boldsymbol{f}_G(\boldsymbol{x}) = [\boldsymbol{f}_n(\boldsymbol{x}^-), \boldsymbol{f}_n(\boldsymbol{x}^+)]$

The method *is equivalent* to the monotonicity-based extension when all variables are monotonic, however it may find sharper enclosures when they are not. By applying this method on the previous example we can obtain the optimal enclosure, i.e., $\boldsymbol{f}_G(\boldsymbol{x}) = [-19.19, 15] \subset [-38, 16]$ which is much better than the traditional monotonicity-based extension. In the following, we explain each step of the method in more detail.

Identifying Increasing and Decreasing Intervals. Let $\boldsymbol{x}_i$ be the domain of variable x_i. In order to find intervals in $\boldsymbol{x}_i^+ \subseteq \boldsymbol{x}_i$, such that f is increasing w.r.t. x_i for any value in $\boldsymbol{x}_i^+$, we use derivatives and a basic contractor (HC4-revise).

Formally, we want to find an interval $\boldsymbol{x}_i^+$, such that $\frac{\partial f}{\partial x_i}(x) > 0$ for any $x \in \boldsymbol{x}_1 \times ... \times \boldsymbol{x}_i^+ \times ... \times \boldsymbol{x}_n \in \boldsymbol{x}$. This problem can be solved by using a contractor for filtering the box $\boldsymbol{x}$ with the negation of the constraint. Let be $\boldsymbol{x}_c$ the

[1] monotonic variables are all of the variables x_i such that f is monotonically increasing (or decreasing) w.r.t. x_i.

[2] also known as FBBT (feasibility-based bounds tightening) by the global optimization community [7].

box resulting when $\boldsymbol{x}$ is contracted by using $\frac{\partial f}{\partial x_i}(x) \leq 0$, then *any* vector x' in the discarded region $\boldsymbol{x}' = \boldsymbol{x} - \boldsymbol{x}_c$ satisfies the constraint $\frac{\partial f}{\partial x_i}(x') > 0$.

Algorithm 1 shows a procedure for finding increasing and decreasing regions inside a box $\boldsymbol{x}$, related to a function f. At each *foreach* iteration, the algorithm selects a variable x_i and, by analyzing the interval $\boldsymbol{x}'_i$ contracted by partial derivatives, it can ensure that the interval at the left (and/or right) of $\boldsymbol{x}'_i$ is increasing (i.e., $l_i = +1$) or decreasing (i.e., $l_i = -1$). If f is increasing (resp. decreasing) in the entire interval $\boldsymbol{x}_i$, then $I_i = True$ (resp. $D_i = True$).

```
1  procedure get_ID_intervals (f, x, x_ini, ϵ): x', l, r, I, D
2      x' ← x_ini; l ← (0, ..., 0), r ← (0, ..., 0); I ← (False, ..., False); D ← (False, ..., False);
3      foreach i ∈ {1..n} do
4          x ← x_ini;
5          do
6              x'_i ← contract(∂f/∂x_i (x) ≤ 0, x);
7              if x'_i is empty then I_i = True; break    // f is increasing w.r.t. x_i;
8              if x'_i > x_i then l_i = +1;    // f is increasing on the left side of x'_i, i.e., [x_i, x'_i];
9              if x̄'_i < x̄_i then r_i = +1:   // f is increasing on the right side of x̄'_i, i.e., [x̄'_i, x̄_i];
10             x ← x_1 × ... × x'_i × ... × x_n;
11             x'_i ← contract(∂f/∂x_i (x) ≥ 0, x);
12             (...) // Analogous analysis for detecting decreasing regions inside x_i
13         while x_i is reduced more than ϵ;
14     return x', l, r, I, D
```

Algorithm 1: Algorithm for finding increasing and decreasing regions in the domains of variables. Interval $\boldsymbol{x}_i$ may be reduced at each do-while iteration by the contractions. Thus, the parameter ϵ defines the minimum required reduction to continue looping.

For instance, if an initial interval $\boldsymbol{x}_i$ is contracted on the right by using the constraint $\frac{\partial f}{\partial x_i}(x) \leq 0$ (i.e., line 9) and on the left by using the constraint $\frac{\partial f}{\partial x_i}(x) \geq 0$, we can ensure that the function is: (1) decreasing when x_i belong to the interval at the left of the contracted interval $\boldsymbol{x}'_i$ and (2) increasing when x_i belongs to the interval at the right of the interval $\boldsymbol{x}'_i$ (see Fig. 1). Note that, paradoxically, we cannot say anything about the values inside $\boldsymbol{x}'_i$.

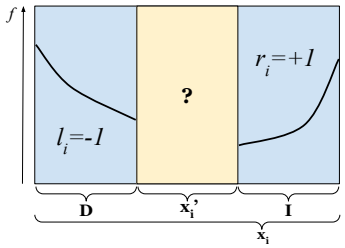


FIGURE 1. Example of applying Algorithm 1 for analyzing monotonicity of f w.r.t. a variable x_i on the domain $\boldsymbol{x}_i$.

Generating sub-domains minimizing and maximizing f. Once increasing and decreasing regions are detected for each interval, we analyze the different cases and, for each of them, we identify in which sub-domain(s) should *surely* be the maximizer and minimizer of f. For instance, in Fig. 1, the value of x_i maximizing f (i.e., the maximizer) *necessarily* has to belong to $\{\underline{x_i}\} \cup \{\overline{x_i}\} \cup \boldsymbol{x}'_i$. On the other hand, the value of x_i minimizing f (i.e., the minimizer) has to belong to $\boldsymbol{x}'_i$. Table 1 summarizes the different cases and their related sub-domains.

Then, by combining the maximizer (resp. minimizer) sub-domains of each variable, we construct the sub-space $\mathcal{X}^+ \in \mathbb{R}$ (resp. $\mathcal{X}^- \in \mathbb{R}$) containing the vector $x \in \boldsymbol{x}$ maximizing (resp. minimizing) f. For instance, given that $x = \{x_1, x_2\}$, if the maximizer of x_1 belongs to $[-1, 0] \cup \{1\}$ and the maximizer of x_2 is -1, then the vector x maximizing f belongs to $\mathcal{X}^+ = [-1, 0] \times [-1, -1] \cup [1, 1] \times [-1, -1]$.

TABLE 1. Sub-domains where the maximizer and minimizer of f should belong to depending on the case.

case	minimizer belongs to:	maximizer belongs to:
$I_i = True$	$\{\underline{x_i}\}$	$\{\overline{x_i}\}$
$D_i = True$	$\{\overline{x_i}\}$	$\{\underline{x_i}\}$
$r_i = +1$ $(l_i = 0)$	$\boldsymbol{x}'_i$	$\boldsymbol{x}'_i \cup \{\overline{x_i}\}$
$r_i = -1$ $(l_i = 0)$	$\boldsymbol{x}'_i \cup \{\overline{x_i}\}$	$\boldsymbol{x}'_i$
$l_i = +1$ $(r_i = 0)$	$\boldsymbol{x}'_i \cup \{\underline{x_i}\}$	$\boldsymbol{x}'_i$
$l_i = -1$ $(r_i = 0)$	$\boldsymbol{x}'_i$	$\boldsymbol{x}'_i \cup \{\underline{x_i}\}$
$l_i = +1, r_i = -1$	$\boldsymbol{x}'_i \cup \{\underline{x_i}, \overline{x_i}\}$	$\boldsymbol{x}'_i$
$l_i = -1, r_i = +1$	$\boldsymbol{x}'_i$	$\boldsymbol{x}'_i \cup \{\underline{x_i}, \overline{x_i}\}$
$l_i = +1, r_i = +1$	$\boldsymbol{x}'_i \cup \{\underline{x_i}\}$	$\boldsymbol{x}'_i \cup \{\overline{x_i}\}$
$l_i = -1, r_i = -1$	$\boldsymbol{x}'_i \cup \{\overline{x_i}\}$	$\boldsymbol{x}'_i \cup \{\underline{x_i}\}$
$l_i = 0, r_i = 0$	$\boldsymbol{x}_i$	$\boldsymbol{x}_i$

Computation of the enclosure. For computing an upper bound (resp. lower bound) for the enclosure of f, we compute enclosures of f over each box of the sub-space $\mathcal{X}^+$ (resp. $\mathcal{X}^-$) by using the natural extension. The upper bound correspond to the maximum upper bound of the enclosures, i.e., considering that $\mathcal{X}^+ = \{\boldsymbol{x}^{+(1)}, ..., \boldsymbol{x}^{+(k)}\}$, the upper bound of the enclosure of f is computed: $\max_{i=1..k} \overline{\boldsymbol{f}_n(\boldsymbol{x}^{+(i)})}$. The lower bound is computed analogously. By applying this method on expression (1), we obtain an optimal enclosure: $\boldsymbol{f}_G(\boldsymbol{x}) = [-19.19, 15]$ (we have to perform 4 computation with the natural extension). A non-combinatorial variant would consist in *hulling* the sub-domains before combining them for constructing $\mathcal{X}^+$. In this case we have to perform only one natural evaluation for each bound, reaching an enclosure of $\boldsymbol{f}_{G'}(\boldsymbol{x}) = [-19.19, 30]$. Fig.2 shows an example of the method on an univariate function.

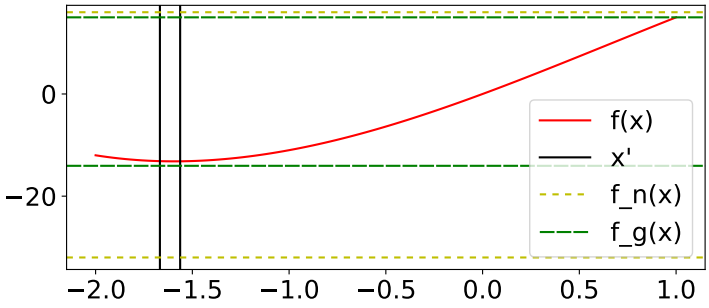


FIGURE 2. Computation of the enclosure of $f(x) = x(-x^2 + 2x + 14)$, with $\boldsymbol{x} = [-2, 1]$. Although the contraction of $\boldsymbol{x}'$ is not optimal, the enclosure found by the proposed method significantly outperform the natural evaluation and it is close to optimal.

We plan to validate the method by incorporating it into an interval-based optimization solver (IbexOpt [8]). The method will be used: (1) for testing feasibility of boxes; (2) as a component of constraint propagation contractor; and (3) for computing enclosures inside a linear relaxation method.

REFERENCES

[1] R. Moore, "Interval analysis, vol. 60," 1966.

[2] I. Araya, G. Trombettoni, B. Neveu, *et al.*, "Exploiting monotonicity in interval constraint propagation.," in *AAAI*, 2010.

[3] L. Granvilliers, "A new interval contractor based on optimality conditions for bound constrained global optimization," in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 90–97, IEEE, 2018.
[4] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*. Springer, 2001.
[5] I. Araya and V. Reyes, "Interval branch-and-bound algorithms for optimization and constraint satisfaction: a survey and prospects," *Journal of Global Optimization*, vol. 65, no. 4, pp. 837–866, 2016.
[6] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget, "Revising hull and box consistency," in *Int. Conf. on Logic Programming*, Citeseer, 1999.
[7] L. Liberti, *Writing Global Optimization Software*, pp. 211–262. Boston, MA: Springer US, 2006.
[8] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert, "Inner regions and interval linearizations for global optimization," in *AAAI Conference on Artificial Intelligence*, pp. 99–104, 2011.